Databases II
Workshop No. 3 — Concurrency, Parallelism, and Distributed Databases

Team Members:
Santiago Sanchez Moya – 20211020032
Juan Daniel Rodriguez Hurtado – 20211020152
Faider Camilo Trujillo Olaya – 20192020136
Computer Engineering Program
School of Engineering
Universidad Distrital Francisco José de Caldas
Professor: Eng. Carlos Andrés Sierra, M.Sc.

Full-time Adjunct Professor

Semester 2025-III

Contents

1. Introduction

This workshop represents the starting point of the course project for *Databases II.* The project focuses on designing and implementing a real-world, data-intensive application for a property rental platform that connects **hosts** who offer accommodations with **guests** seeking short-term stays.

The main goal of the project is to provide a secure, reliable, and efficient system that supports essential operations such as property listings, bookings, payments, and user management. The platform aims to simplify the accommodation process by enabling guests to easily discover and book properties, while ensuring that hosts can manage their listings, track earnings, and receive payouts automatically after completed bookings.

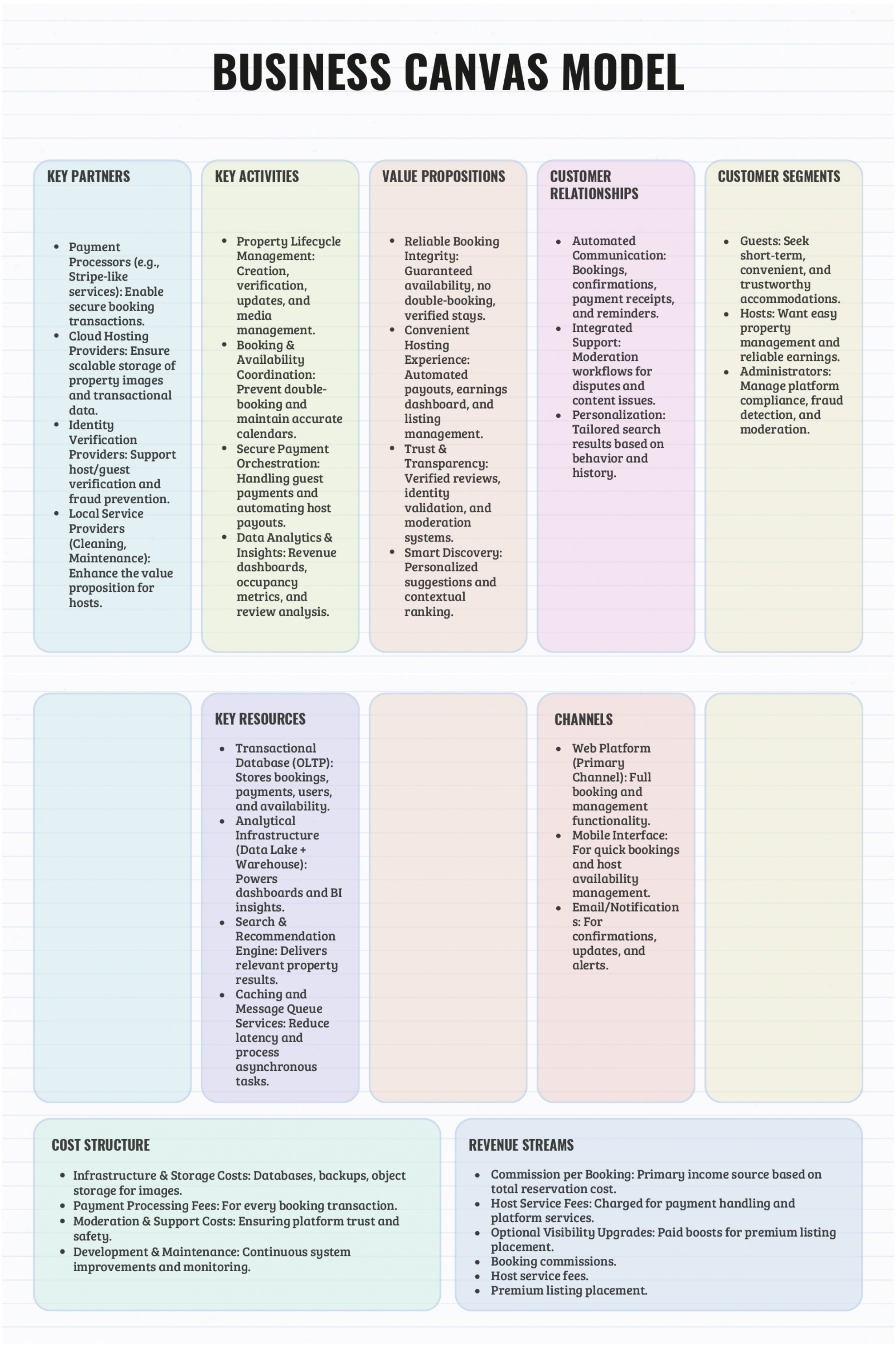
The system addresses the need for transparency, trust, and automation in the property rental market, reducing manual intervention and potential errors in booking and payout management. The core users include:

- **Guests:** Individuals searching for short-term stays.
- **Hosts:** Property owners or managers listing accommodations.
- **Administrators:** System operators ensuring platform compliance, content moderation, and security.

The primary objectives of this project are:

- To model the data requirements and relationships for the property rental domain.
- To define user stories and functional requirements aligned with business needs.
- To design the initial database architecture as the foundation for system implementation.

2. Business Canvas Model



BUSINESS CANVAS MODEL

KEY PARTNERS

- Payment Processors (e.g., Stripe-like services): Enable secure booking transactions.
- Cloud Hosting Providers: Ensure scalable storage of property images and transactional data.
- Identity Verification Providers: Support host/guest verification and fraud prevention.
- Local Service Providers (Cleaning, Maintenance): Enhance the value proposition for hosts.

KEY ACTIVITIES

- Property Lifecycle Management: Creation, verification, updates, and media management.
- Booking & Availability Coordination: Prevent double-booking and maintain accurate calendars.
- Secure Payment Orchestration: Handling guest payments and automating host payouts.
- Data Analytics & Insights: Revenue dashboards, occupancy metrics, and review analysis.

VALUE PROPOSITIONS

- Reliable Booking Integrity: Guaranteed availability, no double-booking, verified stays.
- Convenient Hosting Experience: Automated payouts, earnings dashboard, and listing management.
- Trust & Transparency: Verified reviews, identity validation, and moderation systems.
- Smart Discovery: Personalized suggestions and contextual ranking.

CUSTOMER RELATIONSHIPS

- Automated Communication: Bookings, confirmations, payment receipts, and reminders.
- Integrated Support: Moderation workflows for disputes and content issues.
- Personalization: Tailored search results based on behavior and history.

CUSTOMER SEGMENTS

- Guests: Seek short-term, convenient, and trustworthy accommodations.
- Hosts: Want easy property management and reliable earnings.
- Administrators: Manage platform compliance, fraud detection, and moderation.

KEY RESOURCES

- Transactional Database (OLTP): Stores bookings, payments, users, and availability.
- Analytical Infrastructure (Data Lake + Warehouse): Powers dashboards and BI insights.
- Search & Recommendation Engine: Delivers relevant property results.
- Caching and Message Queue Services: Reduce latency and process asynchronous tasks.

CHANNELS

- Web Platform (Primary Channel): Full booking and management functionality.
- Mobile Interface: For quick bookings and host availability management.
- Email/Notifications: For confirmations, updates, and alerts.

COST STRUCTURE

- Infrastructure & Storage Costs: Databases, backups, object storage for images.
- Payment Processing Fees: For every booking transaction.
- Moderation & Support Costs: Ensuring platform trust and safety.
- Development & Maintenance: Continuous system improvements and monitoring.

REVENUE STREAMS

- Commission per Booking: Primary income source based on total reservation cost.
- Host Service Fees: Charged for payment handling and platform services.
- Optional Visibility Upgrades: Paid boosts for premium listing placement.
- Booking commissions.
- Host service fees.
- Premium listing placement.

Figure 1: Business Canvas Model.

2.1. Improvements

We significantly improved readability with larger fonts and clear structure, and refined all definitions to be specific and directly related to our property rental platform.

3. Requirements Documentation

3.1. Functional Requirements (FR)

FR1. **User Registration & Authentication**

- FR1.1 Users must be able to register as guest or host.
- FR1.2 Users must be able to login/logout with secure password handling.
- FR1.3 Hosts must be able to verify their properties (upload info, photos).

FR2. **Property Management (Hosts)**

- FR2.1 Hosts must be able to create, update, and delete property listings.
- FR2.2 Hosts must manage availability calendars for their properties.
- FR2.3 Hosts must be able to view bookings and revenue history.

FR3. **Property Discovery (Guests)**

- FR3.1 Guests must filter results by price range, amenities, and availability.
- FR3.2 Guests must be able to view property details, reviews, and host profile.

FR4. **Booking & Reservations**

- FR4.1 Guests must be able to book available properties for selected dates.
- FR4.2 Users must not be able to double book for the same property/date.
- FR4.3 Users must receive booking confirmations.

FR5. **Payments**

- FR5.1 Guests must be able to pay for bookings using a payment gateway.
- FR5.2 Hosts must receive payouts after bookings are completed.

FR6. **Reviews & Ratings**

- FR6.1 Guests must be able to leave reviews and ratings after a stay.
- FR6.2 Hosts must be able to respond to reviews.

FR7. **Admin Functions**

- FR7.1 Admins must manage users (ban, verify, reset).
- FR7.2 Admins must manage reported properties and reviews.

3.2. Non-Functional Requirements (NFR)

NFR1. **Performance**

NFR1.1 The system must support a peak load of 100 concurrent users generating search queries. With an estimated rate of 3 queries per user per minute, the database must sustain approximately 300 read operations per minute against the property listings table during peak periods.
Justification: The user concurrency analysis defines the read load on the system. The database and caching layers must be optimized to handle this high frequency of read requests for property discovery.

NFR1.2 The booking system must prevent "double-booking" scenarios under the peak load of 100 concurrent users. The data integrity mechanisms must ensure that conflicting writes for the same property and dates are handled atomically, even when multiple users attempt to book simultaneously.
Justification: The combination of concurrent users and the business-critical nature of reservations creates a high-risk scenario for data conflicts. The system must prioritize transactional integrity under this write concurrency.

NFR1.3 The system must provide hosts with near real-time access to their booking history and revenue data. This requires efficient aggregation of booking and payment records on a per-host basis, updated with minimal latency from completed transactions.
Justification: Hosts require frequent access to their financial and booking data to manage their business. The system must support these personalized, aggregated views with high data freshness.

NFR2. **Scalability**

NFR2.1 The database must efficiently manage a property inventory starting at 1,000 listings and growing at a rate of 50 new properties per month. The schema must support this linear growth without degradation in the performance of core listing creation, update, and search operations.
Justification: Based on the projected growth of the platform's core inventory. The system's design must accommodate this steady influx of new property data.

NFR2.2 The system must be designed to process and store a minimum of 500 new booking transactions per month, with the capability to handle peak loads of 1,000 transactions per month. This includes all associated data writes for availability updates and payment records.
Justification: Derived from the projected booking volume and seasonal peaks. The architecture must handle this transaction rate while maintaining data integrity and system responsiveness.

NFR2.3 The system must store and serve a high volume of unstructured data, specifically property and user profile images. The architecture must decouple this binary large object (BLOB) storage from the transactional database to maintain performance.

Justification: A property rental platform is media-intensive. The frequency of image uploads and downloads necessitates a dedicated, scalable storage solution separate from the structured database.

NFR3. **Availability**

- NFR3.1 Access to the core transactional database (handling users, properties, bookings, and payments) must be maintained to prevent loss of business. The system must be designed to minimize unplanned downtime that would prevent booking operations.
  Justification: The platform's primary revenue stream is directly tied to the availability of booking and payment functions. This data is business-critical.
- NFR3.2 The backup and recovery strategy must be capable of restoring the database to any specific point in time, ensuring a maximum data loss tolerance of up to one hour for financial transactions (payments and bookings).
  Justification: Although the upper tolerance limit is one hour, the financial impact of losing even a single hour of transaction data is considered highly undesirable. Therefore, the system must target a significantly lower effective Recovery Point Objective (RPO), aiming to minimize real data loss to only a few minutes under normal operating conditions.
- NFR3.3 The synchronization mechanism between the write database and the read database must maintain a replication latency of less than 1 millisecond, ensuring that updates committed to the write database are propagated to the read database almost instantaneously.
  Justification: A replication delay below 1 millisecond is necessary to maintain near-real-time consistency between the write and read databases. Such minimal latency reduces the likelihood of stale reads, prevents inconsistencies during high-concurrency operations, and ensures that read replicas accurately reflect the most recent transactional state. This level of synchronization is critical for guaranteeing system availability, enabling timely failover, and preserving data integrity when applications rely on the read database for low-latency queries.

NFR4. **Maintainability**

- NFR6.1 The data model must enforce referential integrity across all core entity types (e.g., User, Property, Booking, Payment, Review) and their defined relationships, as specified in the ERD. This ensures that all bookings are linked to valid users and properties, and all payments are linked to valid bookings.
  Justification: The complex nature of a rental platform requires a highly relational data model. The integrity of the entire system depends on the consistency of these relationships.
- NFR6.2 To support business intelligence (e.g., host revenue dashboards, platform analytics) while maintaining optimal performance for core operations, the system must implement two distinct database instances: a write-optimized database for all transactional operations (bookings, payments, user updates) and a read-optimized database for analytical queries and

reporting. Data must be automatically replicated from the write database to the read database to ensure consistency. All reporting and analytical queries must be directed exclusively to the read database.
Justification: The growing volume of transactional data (500-1,000 bookings/month) and the computational intensity of analytical queries require physical separation of workloads to prevent reporting operations from degrading the performance of critical booking and search transactions.

NFR5. **Security**

- NFR4.1 All client-server communication must use HTTPS/TLS 1.2 or higher.
  Justification: Required for PCI DSS compliance when processing payments.
- NFR4.2 The system must perform input validation on all user forms and API endpoints (100%).
  Justification: Prevents SQL injection, XSS, and other common web vulnerabilities.

NFR6. **Usability**

- NFR5.1 The web interface must fully load within 3 seconds on a 10 Mbps internet connection.
  Justification: Matches Google Lighthouse's performance benchmarks for responsive websites.
- NFR5.2 A user must be able to complete the full booking process in five clicks or fewer from the search results page.
  Justification: Reduces interaction friction and improves conversion, consistent with Nielsen usability principles.

3.3. Requirements Definition Approach

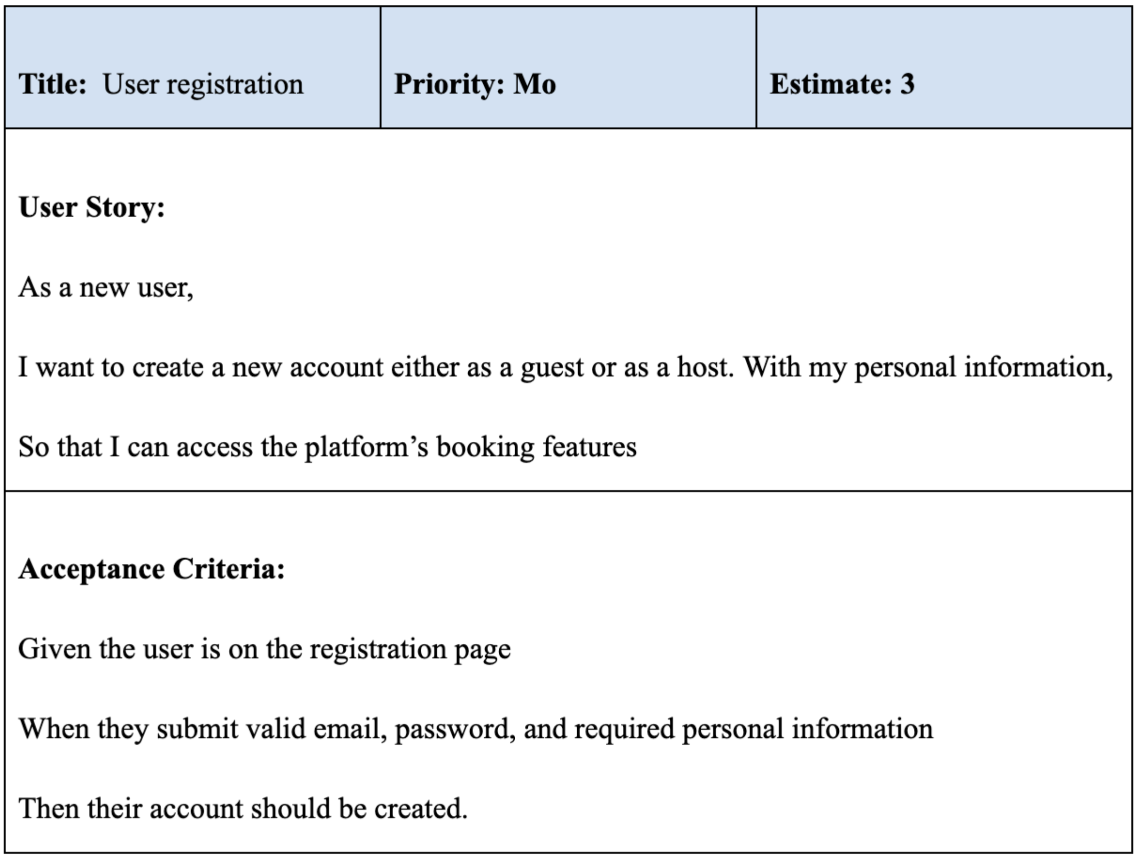
For the functional requirements, we initially defined them from a user-centered perspective to capture a holistic view of platform interactions. Once the global vision was established, we refined and reduced their scope to match the project's achievable range. Conversely, the non-functional requirements were redefined to transform generic expectations into measurable, testable, and technically justified performance and quality targets.

3.4. Improvements

We replaced assumption-based justifications with specific system analysis, eliminating premature references to specific technologies and focusing on conceptual performance and scalability goals.

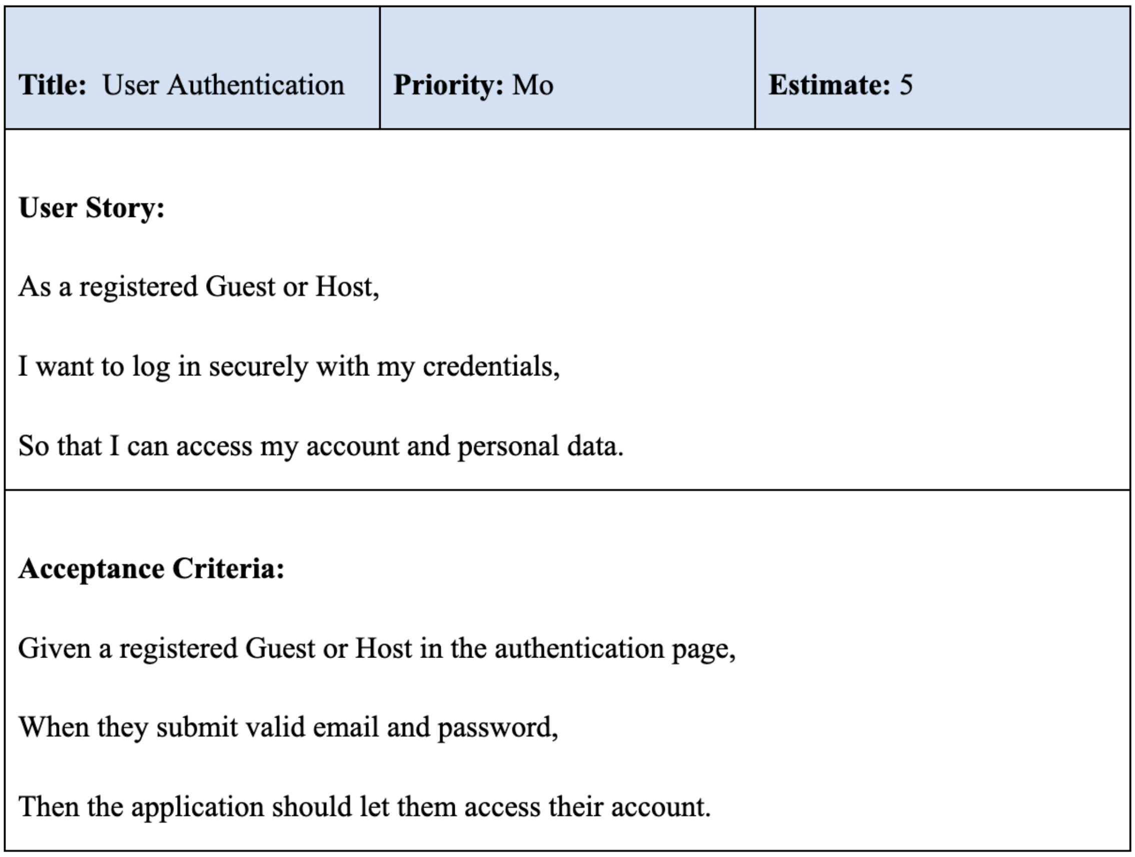
4. User Stories

This section presents the main user stories for the Guest, Host, and Admin roles. To estimate development effort, we used **Planning Poker**—an agile technique that combines expert judgment and team consensus to assign relative sizes to each task. User stories were prioritized following the **MoSCoW** method.



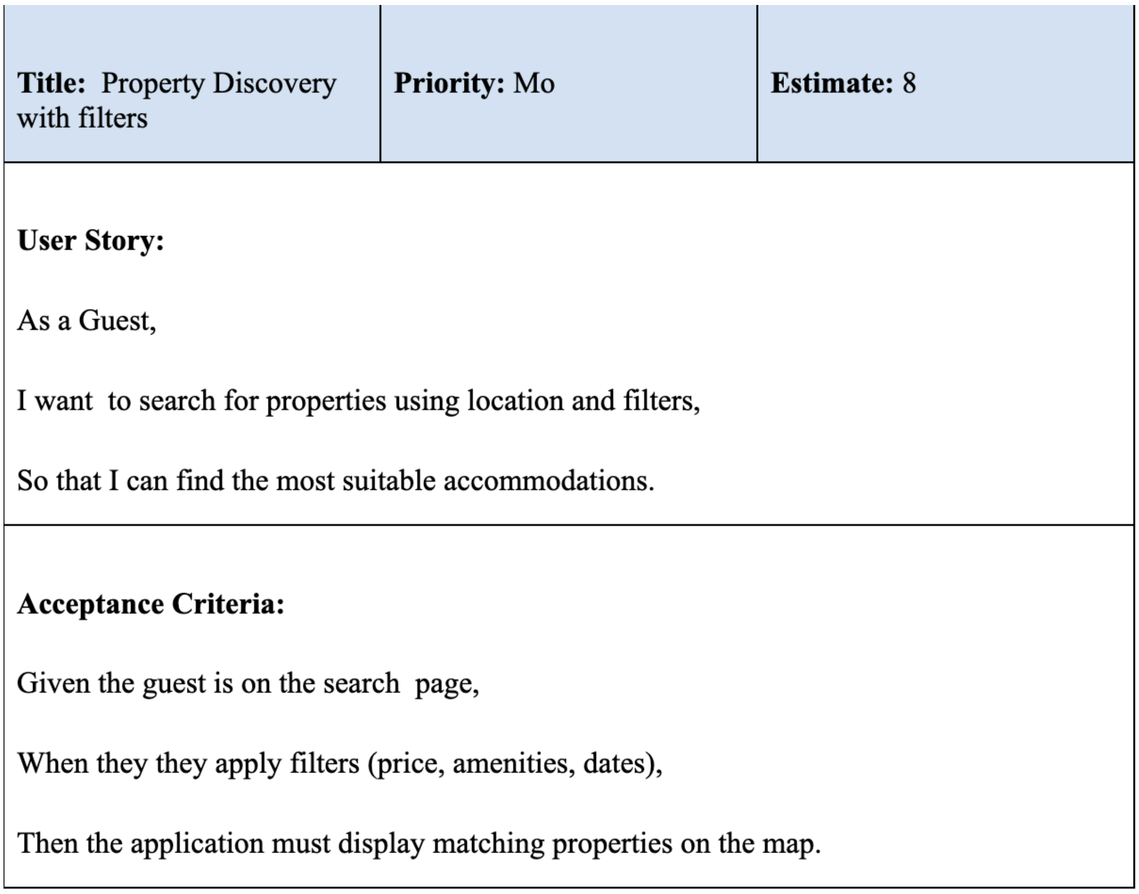
Title: User registration	**Priority: Mo**	**Estimate: 3**
User Story: As a new user, I want to create a new account either as a guest or as a host. With my personal information, So that I can access the platform's booking features		
Acceptance Criteria: Given the user is on the registration page When they submit valid email, password, and required personal information Then their account should be created.		

Figure 2: User Registration



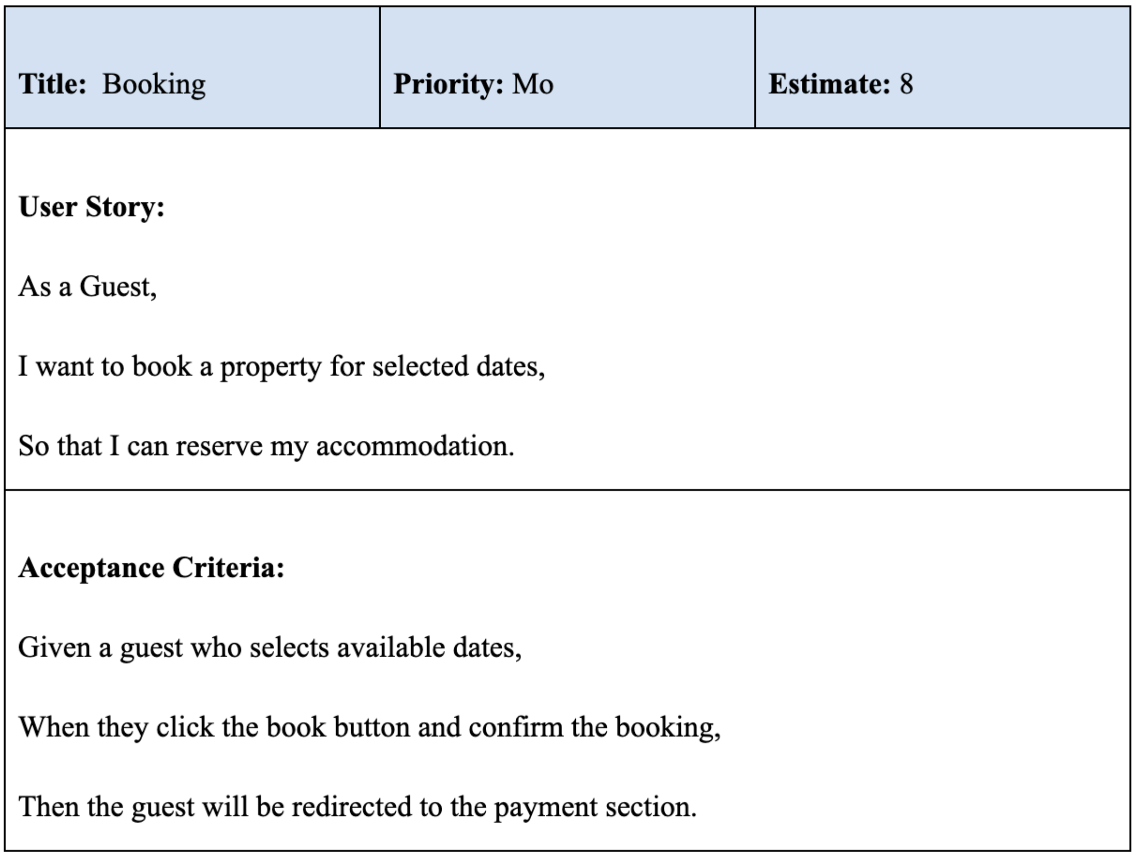
Title: User Authentication	**Priority:** Mo	**Estimate:** 5
User Story: As a registered Guest or Host, I want to log in securely with my credentials, So that I can access my account and personal data.		
Acceptance Criteria: Given a registered Guest or Host in the authentication page, When they submit valid email and password, Then the application should let them access their account.		

Figure 3: User Authentication



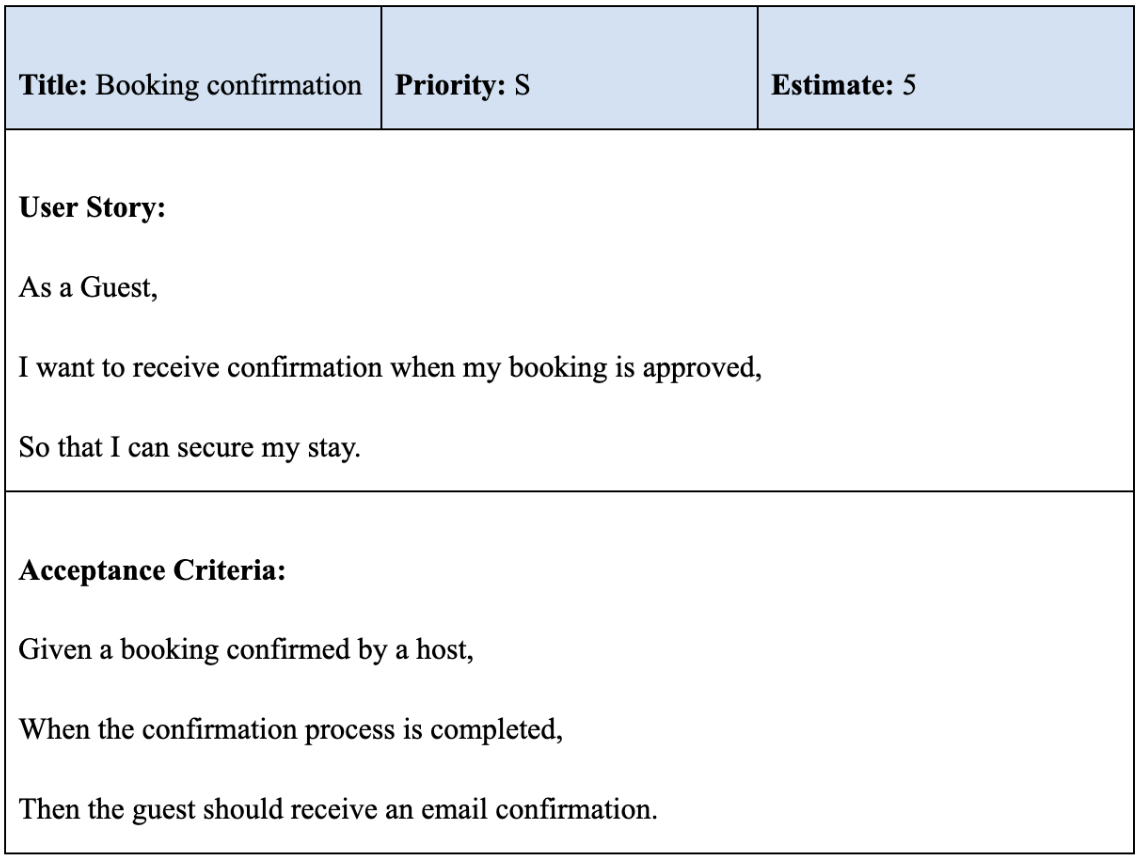
Title: Property Discovery with filters	**Priority:** Mo	**Estimate:** 8
User Story: As a Guest, I want to search for properties using location and filters, So that I can find the most suitable accommodations.		
Acceptance Criteria: Given the guest is on the search page, When they they apply filters (price, amenities, dates), Then the application must display matching properties on the map.		

Figure 4: Property Discovery with filters



Title: Booking	**Priority:** Mo	**Estimate:** 8
User Story: As a Guest, I want to book a property for selected dates, So that I can reserve my accommodation.		
Acceptance Criteria: Given a guest who selects available dates, When they click the book button and confirm the booking, Then the guest will be redirected to the payment section.		

Figure 5: Booking



Title: Booking confirmation	**Priority:** S	**Estimate:** 5

User Story:

As a Guest,

I want to receive confirmation when my booking is approved,

So that I can secure my stay.

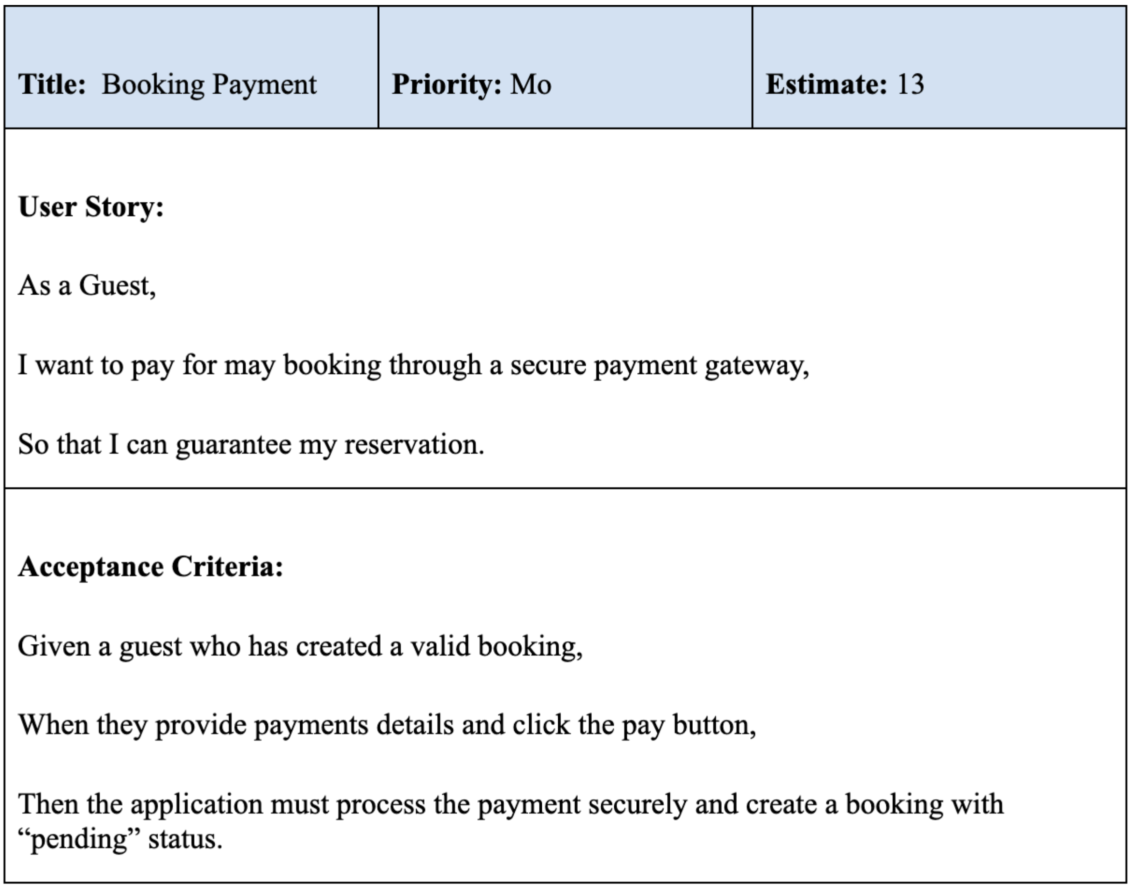
Acceptance Criteria:

Given a booking confirmed by a host,

When the confirmation process is completed,

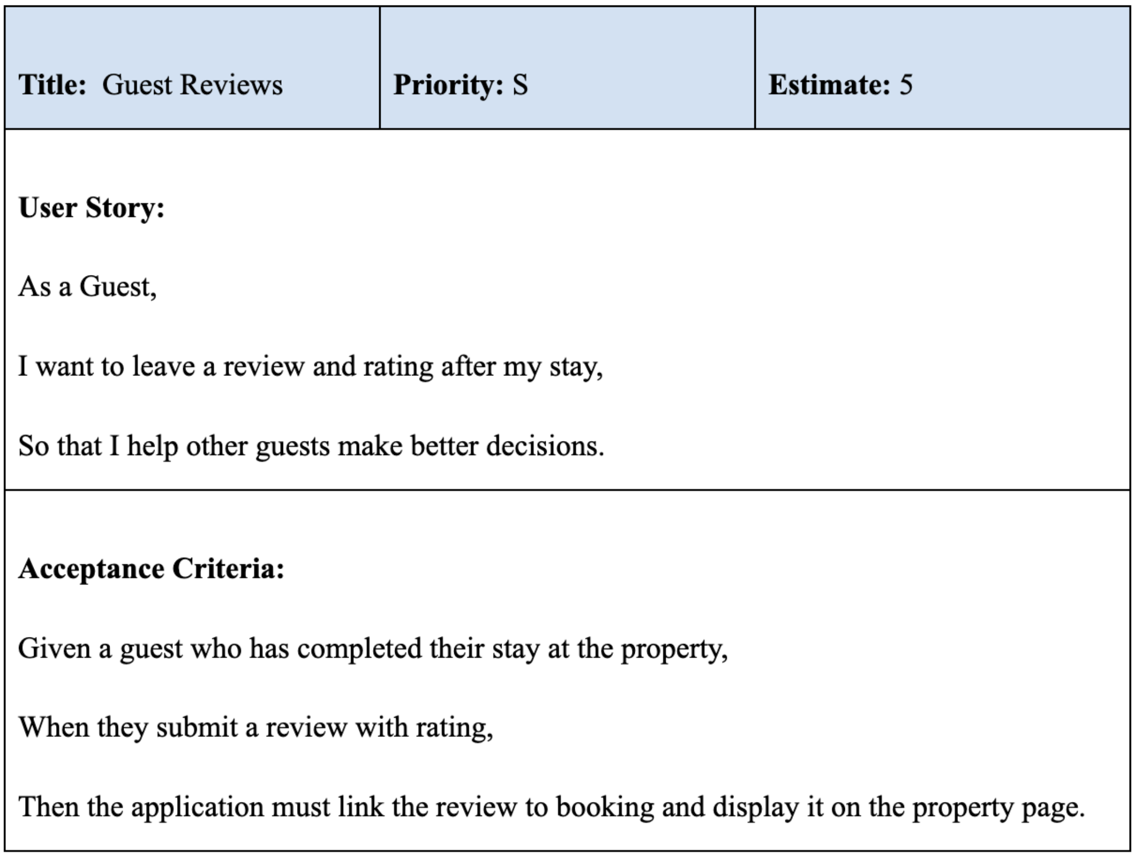
Then the guest should receive an email confirmation.

Figure 6: Booking confirmation



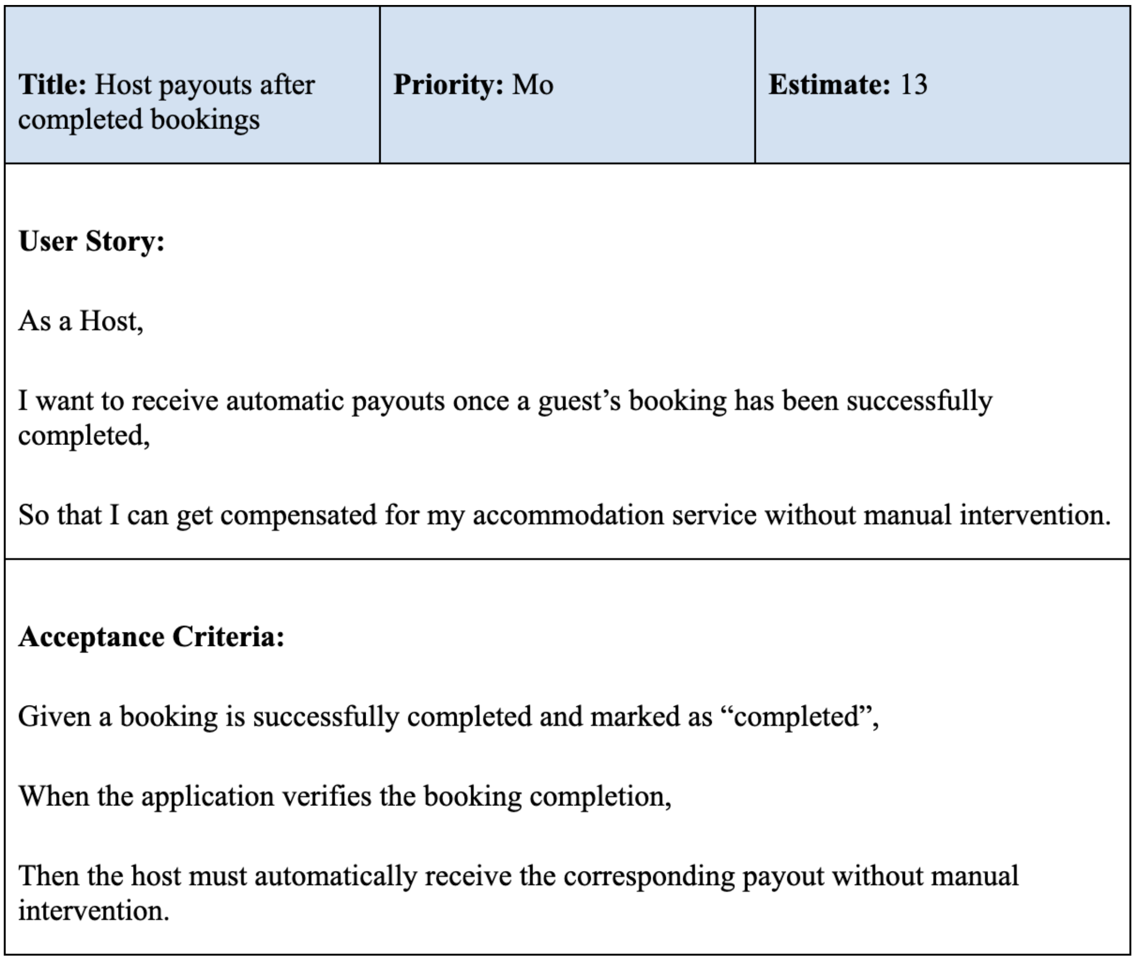
Title: Booking Payment	**Priority:** Mo	**Estimate:** 13
User Story: As a Guest, I want to pay for may booking through a secure payment gateway, So that I can guarantee my reservation.		
Acceptance Criteria: Given a guest who has created a valid booking, When they provide payments details and click the pay button, Then the application must process the payment securely and create a booking with "pending" status.		

Figure 7: Booking Payment



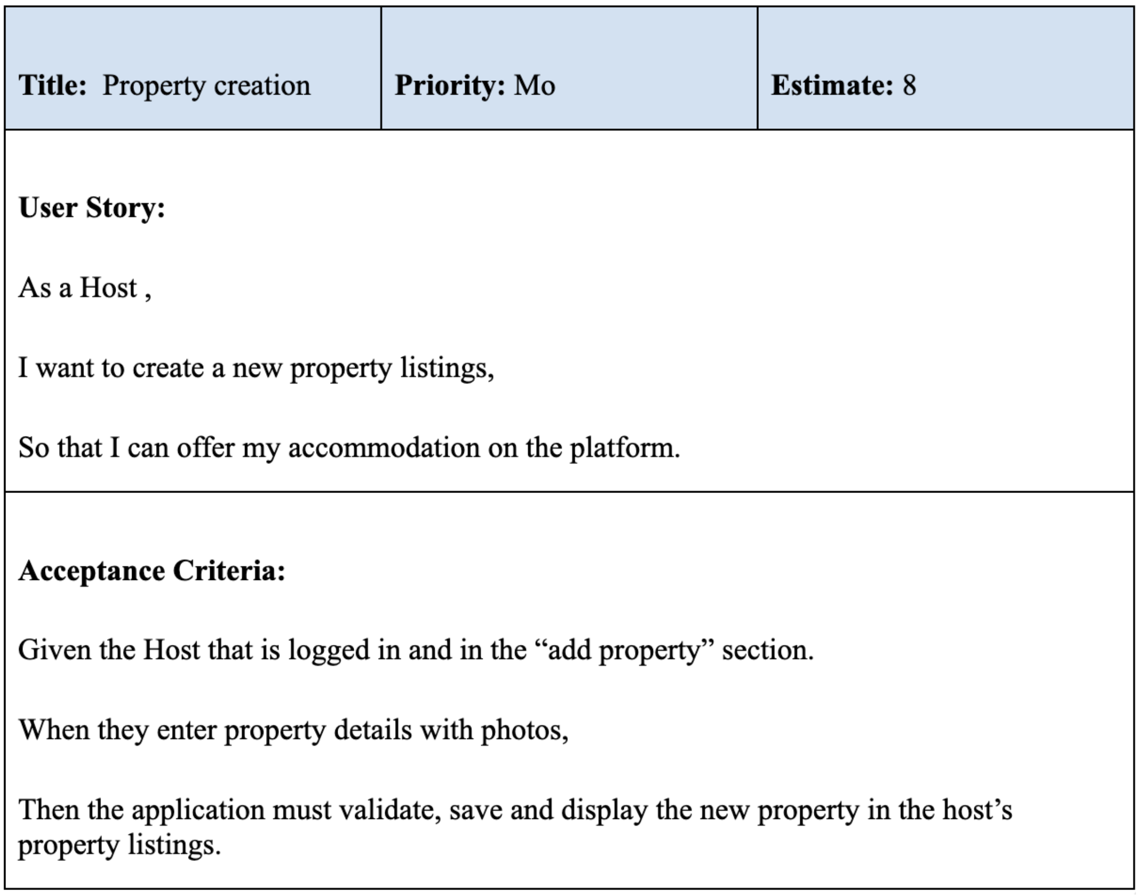
Title: Guest Reviews	**Priority:** S	**Estimate:** 5
User Story: As a Guest, I want to leave a review and rating after my stay, So that I help other guests make better decisions.		
Acceptance Criteria: Given a guest who has completed their stay at the property, When they submit a review with rating, Then the application must link the review to booking and display it on the property page.		

Figure 8: Guest Reviews



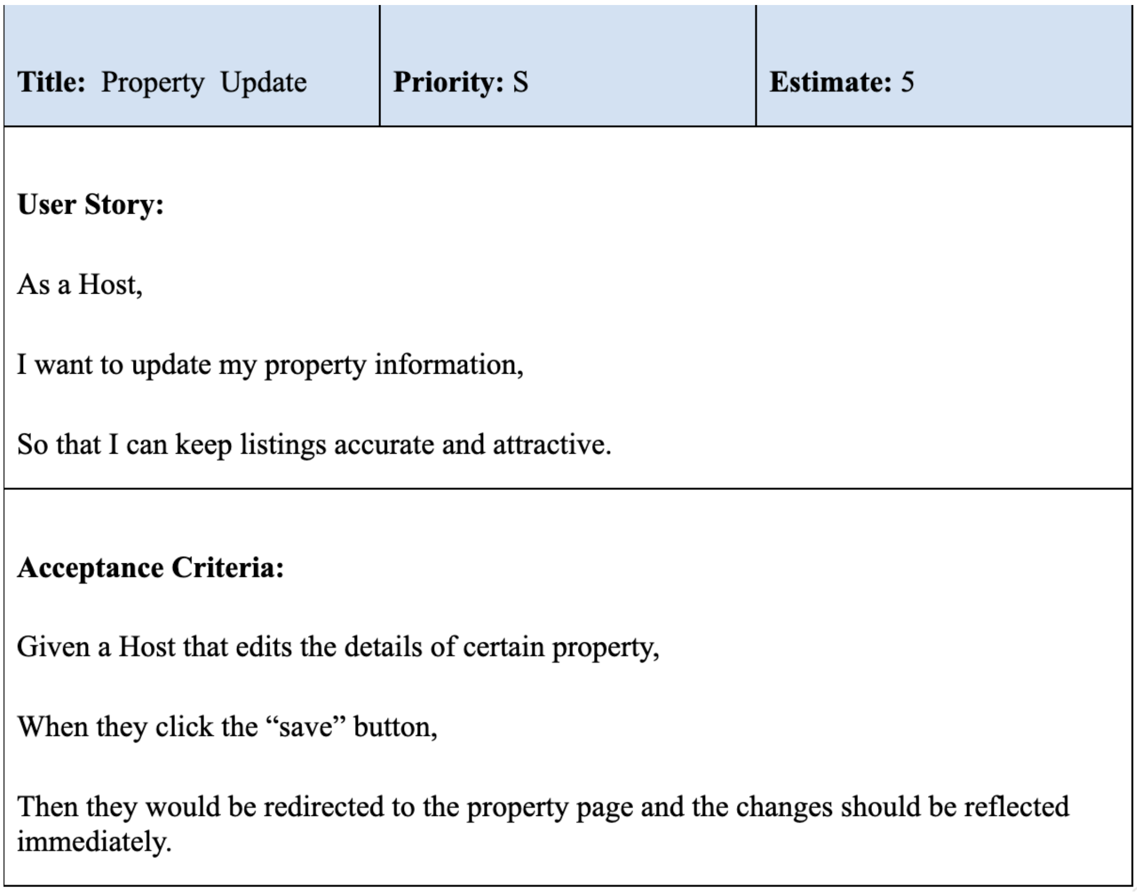
Title: Host payouts after completed bookings	**Priority:** Mo	**Estimate:** 13
User Story: As a Host, I want to receive automatic payouts once a guest's booking has been successfully completed, So that I can get compensated for my accommodation service without manual intervention.		
Acceptance Criteria: Given a booking is successfully completed and marked as "completed", When the application verifies the booking completion, Then the host must automatically receive the corresponding payout without manual intervention.		

Figure 9: Host payouts after completed bookings



Title: Property creation	**Priority:** Mo	**Estimate:** 8
User Story: As a Host , I want to create a new property listings, So that I can offer my accommodation on the platform.		
Acceptance Criteria: Given the Host that is logged in and in the “add property” section. When they enter property details with photos, Then the application must validate, save and display the new property in the host’s property listings.		

Figure 10: Property creation



Title: Property Update	**Priority:** S	**Estimate:** 5

User Story:

As a Host,

I want to update my property information,

So that I can keep listings accurate and attractive.

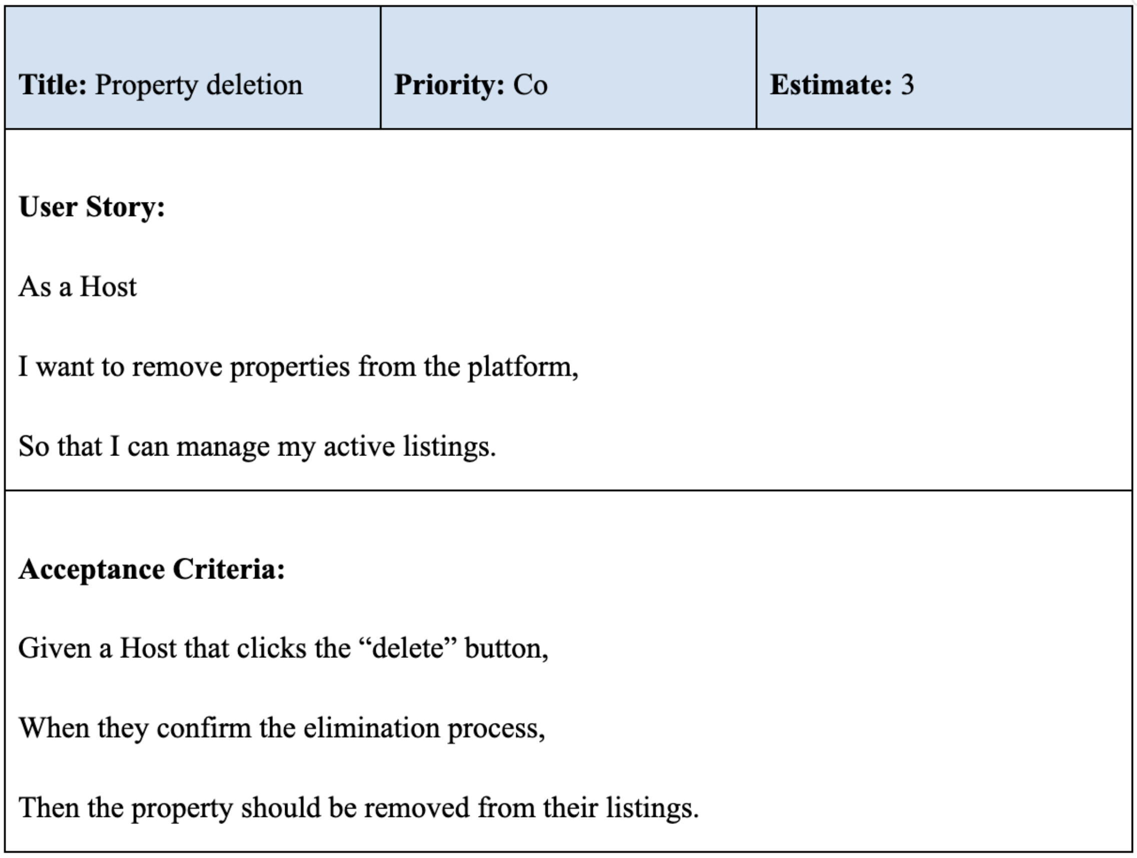
Acceptance Criteria:

Given a Host that edits the details of certain property,

When they click the “save” button,

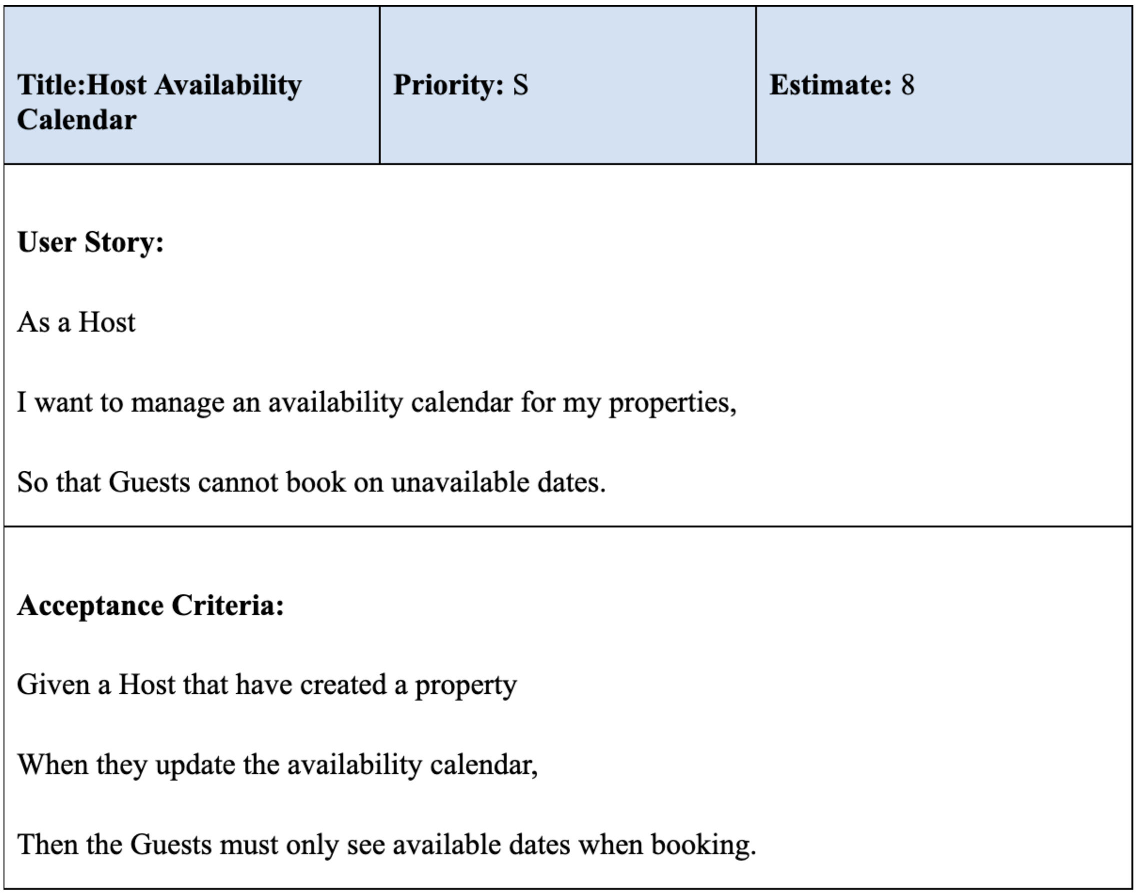
Then they would be redirected to the property page and the changes should be reflected immediately.

Figure 11: Property Update



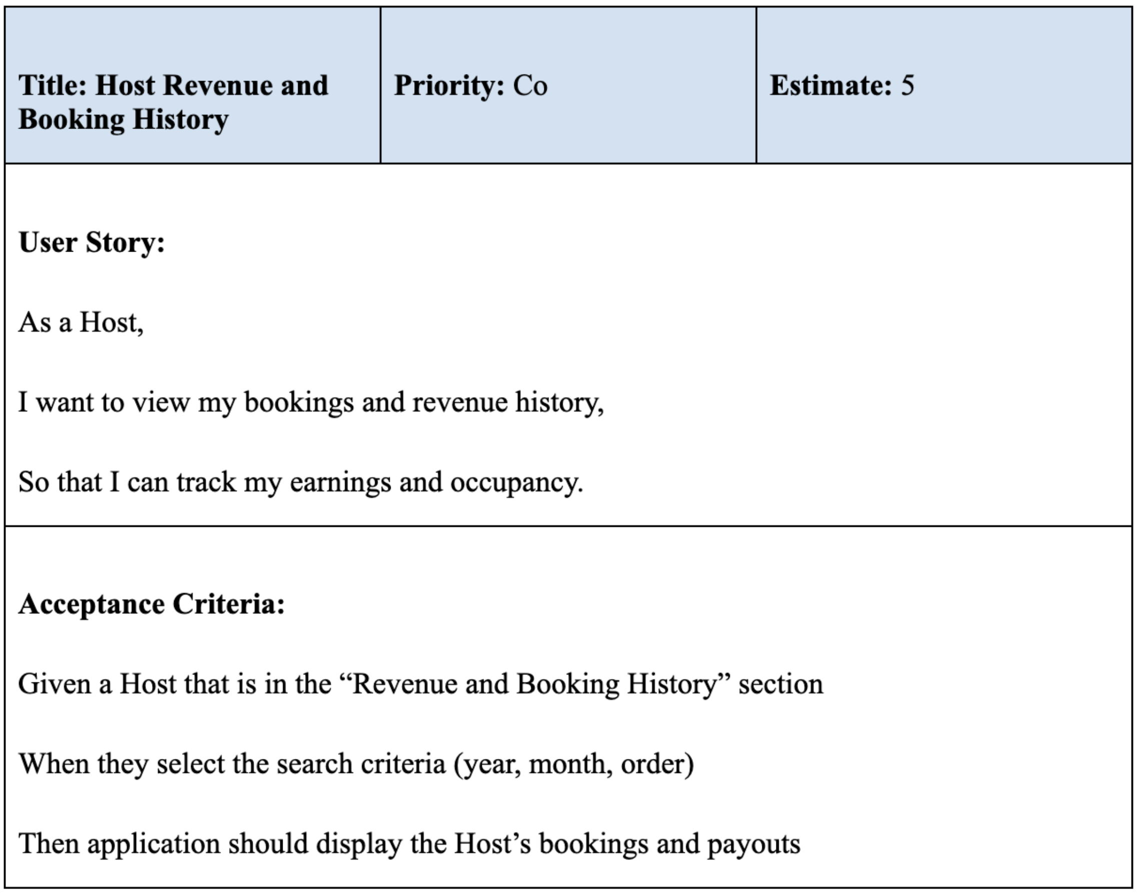
Title: Property deletion	**Priority:** Co	**Estimate:** 3
User Story: As a Host I want to remove properties from the platform, So that I can manage my active listings.		
Acceptance Criteria: Given a Host that clicks the “delete” button, When they confirm the elimination process, Then the property should be removed from their listings.		

Figure 12: Property deletion



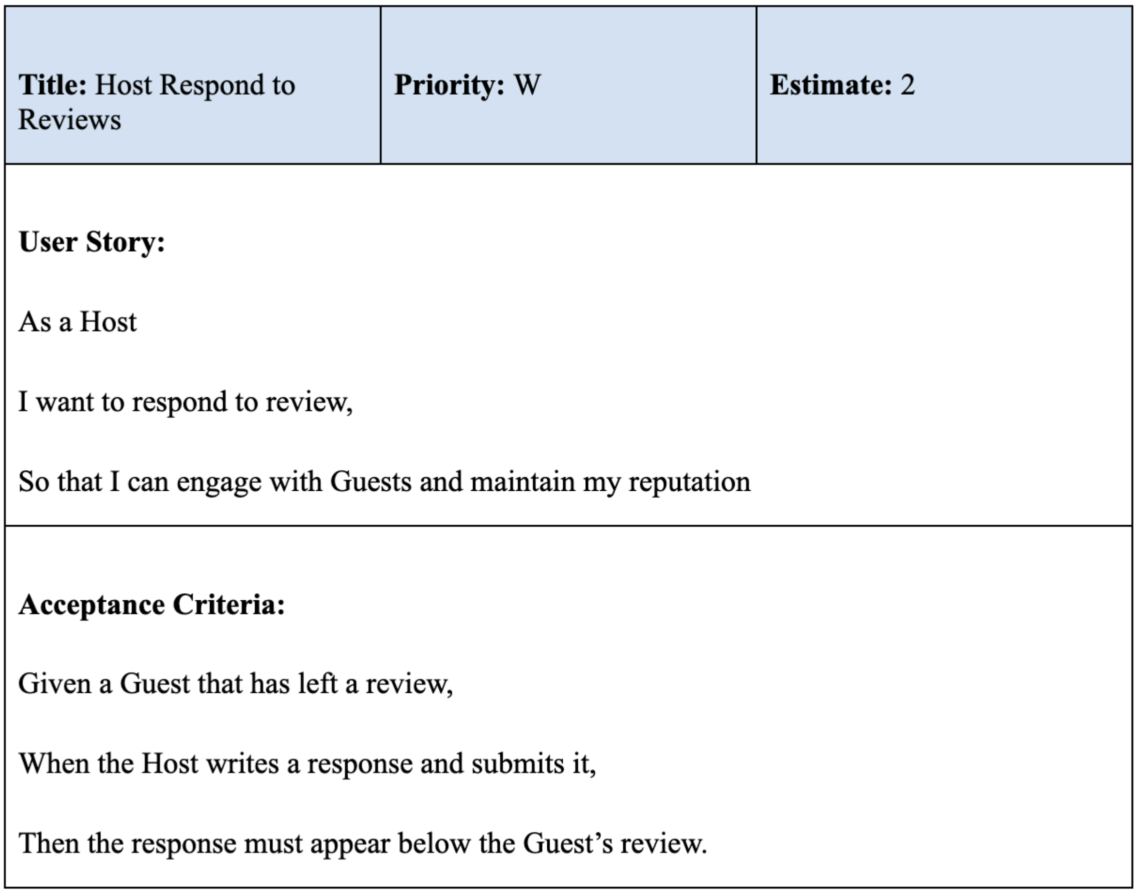
Title:Host Availability Calendar	**Priority:** S	**Estimate:** 8
User Story: As a Host I want to manage an availability calendar for my properties, So that Guests cannot book on unavailable dates.		
Acceptance Criteria: Given a Host that have created a property When they update the availability calendar, Then the Guests must only see available dates when booking.		

Figure 13: Host Availability Calendar



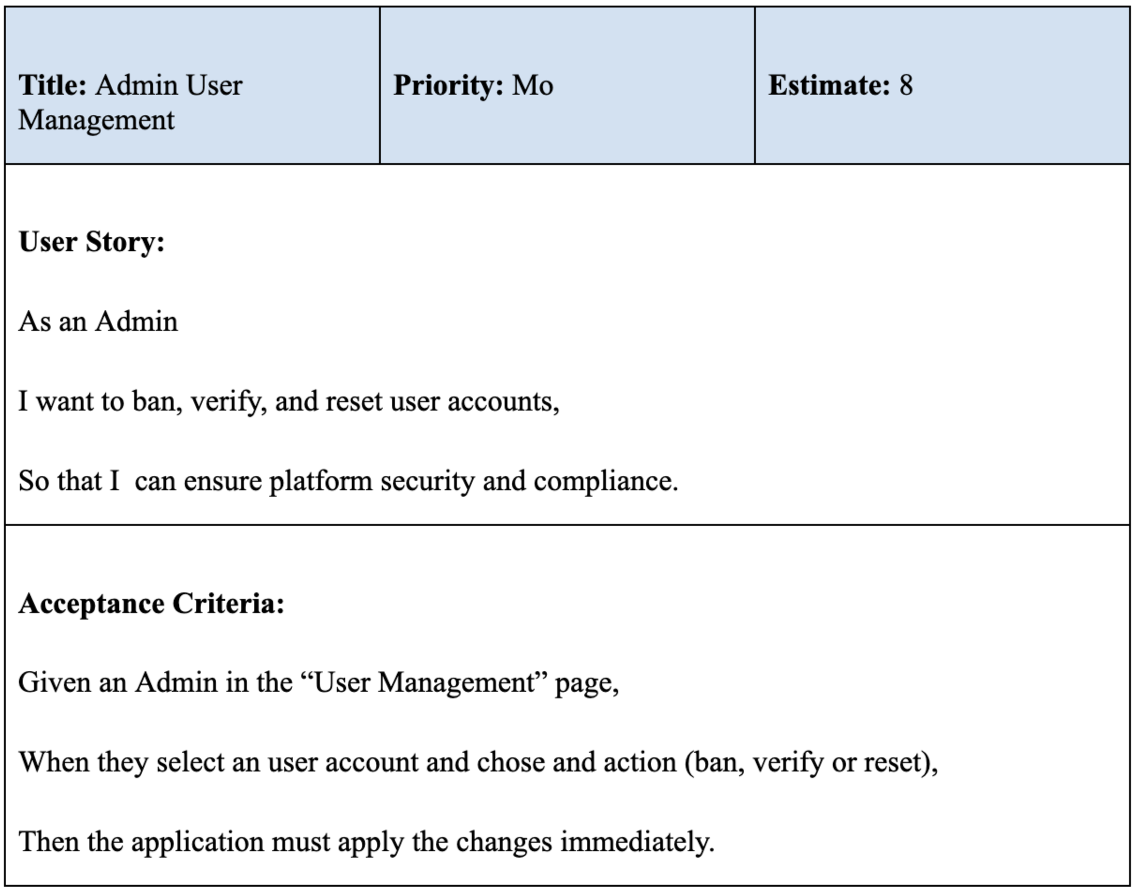
Title: Host Revenue and Booking History	**Priority:** Co	**Estimate:** 5
User Story: As a Host, I want to view my bookings and revenue history, So that I can track my earnings and occupancy.		
Acceptance Criteria: Given a Host that is in the “Revenue and Booking History” section When they select the search criteria (year, month, order) Then application should display the Host’s bookings and payouts		

Figure 14: Host Revenue and Booking History



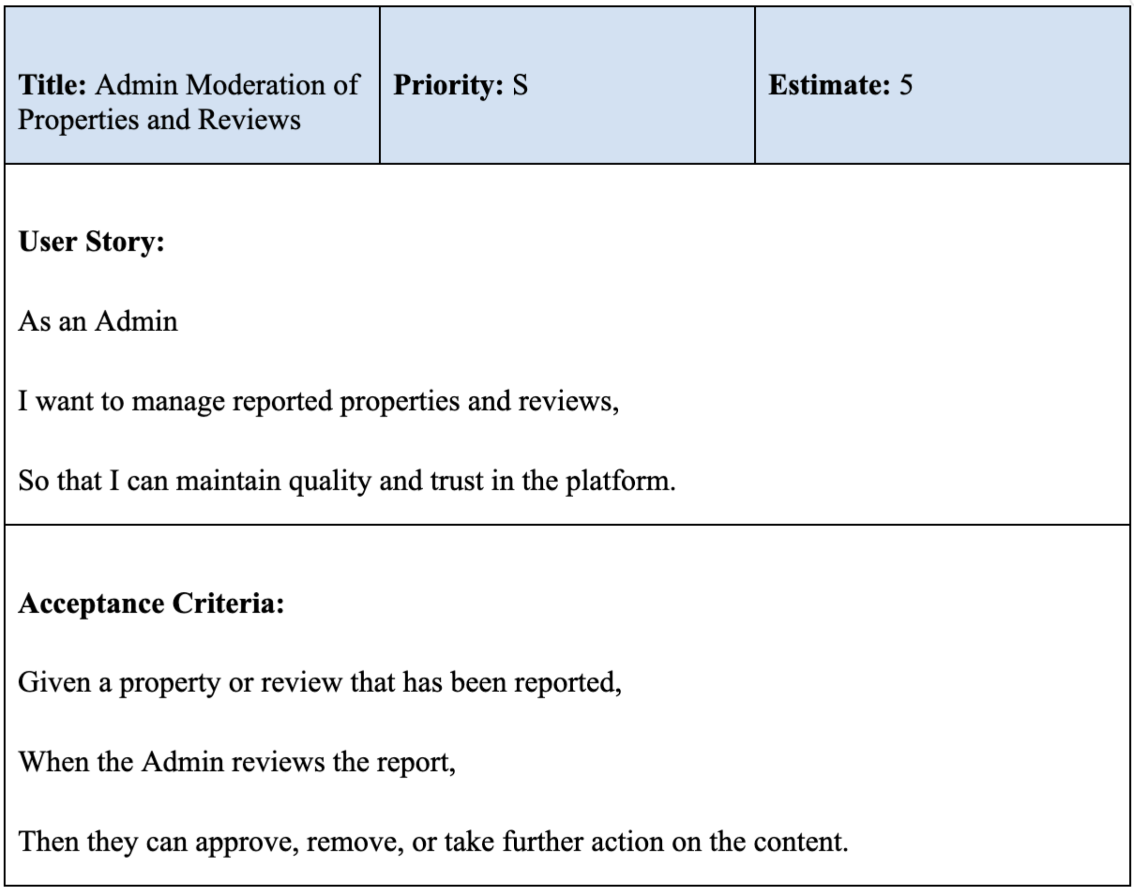
Title: Host Respond to Reviews	**Priority:** W	**Estimate:** 2
User Story: As a Host I want to respond to review, So that I can engage with Guests and maintain my reputation		
Acceptance Criteria: Given a Guest that has left a review, When the Host writes a response and submits it, Then the response must appear below the Guest's review.		

Figure 15: Host Respond to Reviews



Title: Admin User Management	**Priority:** Mo	**Estimate:** 8
User Story: As an Admin I want to ban, verify, and reset user accounts, So that I can ensure platform security and compliance.		
Acceptance Criteria: Given an Admin in the “User Management” page, When they select an user account and chose and action (ban, verify or reset), Then the application must apply the changes immediately.		

Figure 16: Admin User Management



Title: Admin Moderation of Properties and Reviews	**Priority:** S	**Estimate:** 5
User Story: As an Admin I want to manage reported properties and reviews, So that I can maintain quality and trust in the platform.		
Acceptance Criteria: Given a property or review that has been reported, When the Admin reviews the report, Then they can approve, remove, or take further action on the content.		

Figure 17: Admin Moderation of Properties and Reviews

4.1. Improvements

We completely restructured the user stories following the correct "As a [role]/I want to [goal]/So that [benefit]" format with third-person acceptance criteria, eliminating ambiguity and improving traceability.

5. Initial Database Architecture

5.1. High-level Architecture

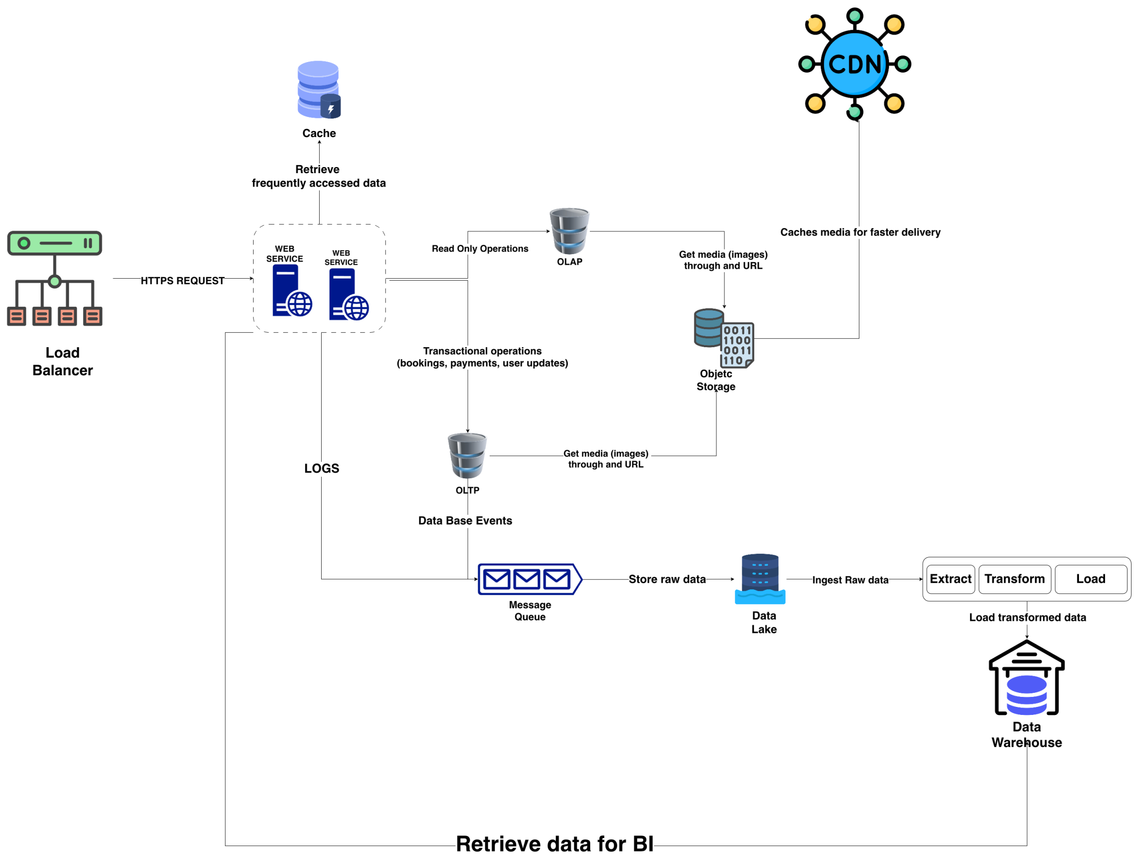


Figure 18: High-level Architecture.

5.2. Architecture Overview

The system is structured into four distinct layers, each specialized for specific data processing patterns and operational requirements:

5.2.1 A. Presentation & Distribution Layer

Purpose: Manage user request distribution, static content delivery, and initial request routing.

Components:

- **Content Delivery Network (CDN):** Distributes static assets globally to reduce latency and offload origin servers
- **Load Balancer:** Distributes incoming traffic across multiple application instances for high availability and scalability

Why this layer must exist: Global user distribution requires efficient content delivery and traffic management to ensure consistent performance and fault tolerance across geographical regions.

5.2.2 B. Operational Layer (OLTP – Transactional)

Purpose: Handle real-time, user-driven operations with strict consistency requirements.

Components:

- **Application Services:** Execute business logic for bookings, authentication, property management, and user interactions
- **Transactional Database:** Maintains normalized, ACID-compliant data for core business operations
- **Object Storage:** Stores unstructured binary data including property images, user photos, and document attachments

Why this layer must exist: Transactional integrity is critical for financial operations and inventory management, while binary assets require specialized storage with cost-effective scaling.

5.2.3 C. Analytical Layer (OLAP – Read-Optimized)

Purpose: Support business intelligence, reporting, and data analytics without impacting operational performance.

Components:

- **Read Replica:** Serves read-heavy queries for search and listing operations
- **Data Lake:** Ingests and stores raw, unstructured data from various sources including application logs and user events
- **Data Warehouse:** Maintains aggregated, historical data optimized for analytical queries
- **ETL Pipeline:** Transforms and loads raw data into structured formats suitable for analysis

Why this layer must exist: Analytical processing requires different data models and access patterns than transactional systems; separating these workloads prevents performance degradation of core business operations.

5.2.4 D. Support Layer (Performance & Asynchronous Processing)

Purpose: Enhance system performance, decouple components, and handle background processing.

Components:

- **Cache:** Stores frequently accessed data to reduce database load and improve response times
- **Message Queue:** Manages asynchronous task processing and event-driven communication between services

Why this layer must exist: Performance optimization and workload decoupling are essential for maintaining responsive user experiences under variable load conditions.

5.3. Data Flow Specification

1. **User Request Flow:**
   - User request → CDN (static assets) / Load Balancer (dynamic content) → Application Services
2. **Transactional Operations:**
   - Application Services → Transactional Database (writes) + Object Storage (binary data)
3. **Data Serving & Caching:**
   - Read operations → Cache (hot data) / Read Replica (complex queries)
   - Cache miss → Transactional Database or Read Replica → Populate Cache
4. **Asynchronous Processing:**
   - Application Services → Message Queue (events) → Background processors → Updates to databases
5. **Analytical Processing:**
   - Transactional Database + Data Lake → ETL Pipeline → Data Warehouse → Business Intelligence tools
6. **Content Delivery:**
   - Object Storage → CDN edge locations → Global users with low-latency access

5.4. Architectural Decision Justification

5.5. Data Lifecycle Management

The architecture supports complete data lifecycle management:

- **Ingestion:** Real-time through Application Services, batch through ETL processes
- **Storage:** Multi-tiered storage based on access patterns and data characteristics
- **Processing:** Real-time for user operations, batch for analytics, asynchronous for background tasks
- **Distribution:** Global content delivery through CDN, API-based data access for applications
- **Archival:** Automated data movement from operational stores to analytical and long-term storage

Architectural Decision	**Justification**
Four-Layer Separation	Clear separation of concerns between presentation, operations, analytics, and support functions enables specialized optimization and independent scaling
CDN + Load Balancer	Addresses global user distribution and ensures high availability through intelligent traffic routing and static content caching at edge locations
Object Storage Integration	Provides cost-effective, scalable storage for binary assets while maintaining separation from structured transactional data
OLTP/OLAP Separation	Prevents analytical query workloads from impacting critical booking transactions and user-facing operations
Read Replica Implementation	Offloads read-intensive search and listing operations from the primary transactional database
Caching Strategy	Reduces database load and improves response times for frequently accessed data patterns
Message-Based Decoupling	Enables asynchronous processing of non-critical path operations without blocking user interactions
ETL Pipeline Architecture	Supports business intelligence requirements while maintaining data consistency and historical tracking

5.6. Improvements

We completely redesigned the architecture to focus on data flows, main components, and system boundaries, removing front-end details and specific technologies that were not relevant at this stage.

5.7. Entity-Relationship Diagram

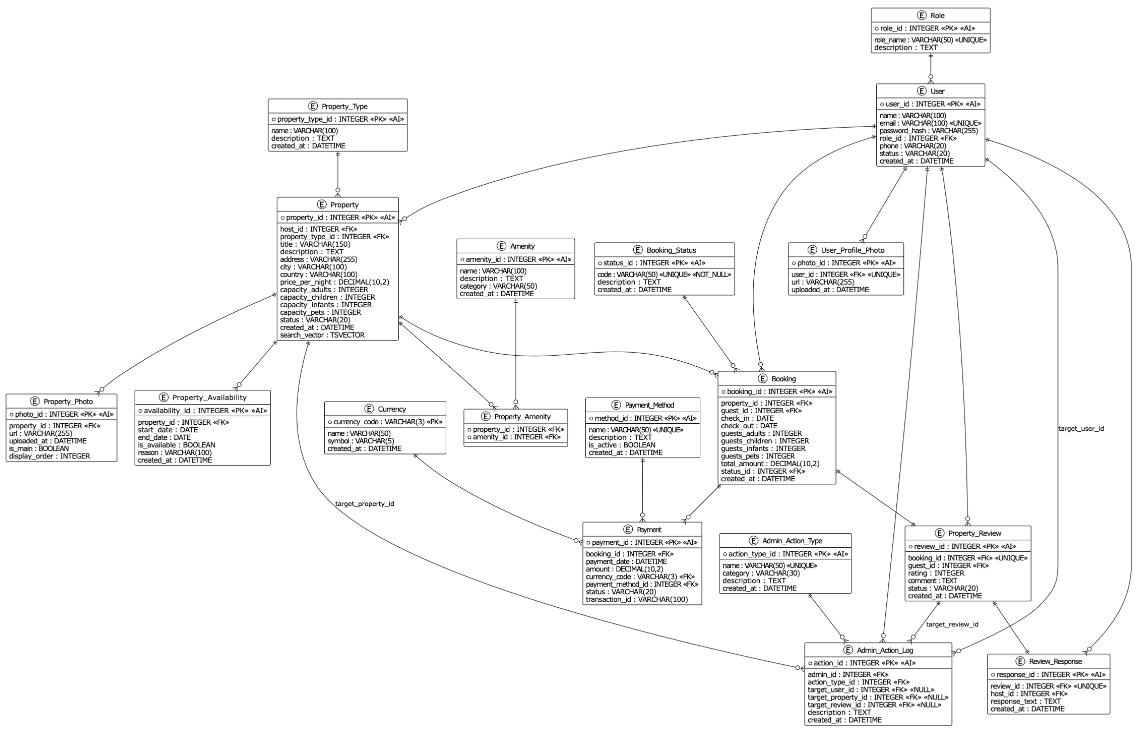


Figure 19: Entity-Relationship (ER) diagram of the database.

The Entity–Relationship (ER) diagram represents the logical structure of the system's database, which supports the main business processes of the platform: user management, property management,listing and discovery, booking and payments, and post-stay reviews. The model is thought to avoid redundancy and ensure referential integrity through foreign keys. Each group of entities is described and justified below.

User Management

- **User**: Central entity representing any system participant (guest, host, or admin). It stores essential identification data (name, email, hashed password, phone, and status). The field `role_id` determines access privileges and enables modular user management.
- **User_Profile_Photo**: Keeps the URL of the user's profile image in a separate table to maintain file management decoupled from the core user data. The one-to-one relationship ensures each user has a single active profile image, which improves clarity and supports potential media service integrations.
- **Role**: Defines user roles such as *guest*, *host*, or *admin*. This table allows role scalability (e.g., adding "superadmin" or "support") without altering user logic.
- **Relationships**:
  - **User–Role:** Many-to-One. Each user is assigned exactly one role, while each role can be shared by many users.

  - **User–User_Profile_Photo:** One-to-One. Each user has one photo entry, ensuring consistent linking of media data.

- **Justification:** This modular separation improves data normalization, reduces duplication, and allows flexibility for authentication and authorization mechanisms.

Property Management

- **Property**: Represents a listing created by a host. It includes details such as title, description, location, price, and capacity attributes (adults, children, infants, pets). The field `host_id` connects it to the user who owns the listing.
- **Property_Type**: Categorizes properties (e.g., Apartment, Cabin, Villa) and supports filtering in search functionalities.
- **Property_Photo**: Manages property images independently to allow multiple photos per listing. The inclusion of `is_main` and `display_order` supports image prioritization and user interface consistency.
- **Property_Availability**: Records periods when a property is available or blocked. This enables hosts to manually restrict dates, complementing availability derived automatically from bookings.
- **Amenity** and **Property_Amenity**: Define a many-to-many relationship between properties and their available features (WiFi, kitchen, parking, etc.). The join table `Property_Amenity` ensures flexibility and avoids data repetition.
- **Relationships**:
  - **Property–User:** Many-to-One (each host can own multiple properties).
  - **Property–Property_Type:** Many-to-One.
  - **Property–Property_Photo:** One-to-Many.
  - **Property–Property_Availability:** One-to-Many.
  - **Property–Amenity:** Many-to-Many (via Property_Amenity).
- **Justification:** This structure minimizes redundancy, allows flexible management of property metadata, and supports scalability for new property categories or features. The inclusion of availability records and full-text search (`search_vector`) enhances booking efficiency and search performance.

Booking and Payments

- **Booking**: Represents a reservation made by a guest for a specific property, including check-in/out dates, number of guests, and total amount. The foreign key `status_id` links to a dynamic list of states (pending, confirmed, cancelled, completed), improving system extensibility.
- **Booking_Status**: Enumerates booking lifecycle states to maintain referential consistency and allow standardized status handling.

- **Payment**: Stores financial transactions associated with bookings. It records the amount, currency, payment method, transaction identifier, and payment status.
- **Payment_Method** and **Currency**: Define controlled vocabularies for available payment methods and currencies, supporting multi-currency operations and compliance with financial APIs.
- **Relationships**:
  - **Booking–Property:** Many-to-One (each booking belongs to one property).
  - **Booking–User:** Many-to-One (guest makes bookings).
  - **Booking–Booking_Status:** Many-to-One.
  - **Booking–Payment:** One-to-One (each booking generates one payment record).
  - **Payment–Currency:** Many-to-One.
  - **Payment–Payment_Method:** Many-to-One.
- **Justification:** This structure ensures atomicity of financial transactions and clear traceability between reservations and payments. It supports both real-time processing and future extensions for refunds or payout automation.

Reviews and Responses

- **Property_Review**: Allows guests to leave a review after a completed booking, including rating and comment fields. The relation to `Booking` guarantees that only verified stays generate reviews, improving credibility.
- **Review_Response**: Enables hosts to respond to reviews, maintaining a transparent communication channel between both parties.
- **Relationships**:
  - **Property_Review–Booking:** One-to-One (each booking can generate one review).
  - **Property_Review–User:** Many-to-One (guest authoring the review).
  - **Review_Response–Property_Review:** One-to-One.
  - **Review_Response–User:** Many-to-One (host posting the response).
- **Justification:** The review structure enforces referential integrity, prevents duplicate evaluations per stay, and aligns with user experience goals of post-stay feedback and trust-building between guests and hosts.

In summary, the ER model ensures a normalized, scalable, and maintainable schema aligned with the application's functional requirements. Each relationship was carefully defined to balance data consistency, system performance, and future extensibility for features such as payouts, notifications, and administrative auditing.

5.8. Improvements

We recreated the entity-relationship diagram using professional diagramming tools, improving readability with clear labels, proper spacing, and well-defined logical relationships.

6. Information Requirements

The information required by the system can be grouped according to the frequency of generation, the volume of data, and the type of outputs produced. This ensures that storage, processing, and retrieval mechanisms support both operational and analytical workloads

6.1. Expected Outputs and Data Specifications

6.1.1 Operational Outputs

Booking Confirmations

- **Structure:** Report containing property details, guest information, booking dates, total amount, and confirmation code
- **Data Generation & Combination:**
  - **Volume:** 50-100 documents daily
  - **Variety:** Structured data from Booking, Property, User, and Property_Type tables
  - **Frequency:** Real-time generation per booking
  - **Data Sources:** `Booking` + `Property` + `User` + `Property_Type` → Combined via JOIN on property_id and user_id

Payment Receipts

- **Structure:** Transaction summary with amount, payment method, timestamp, booking reference
- **Data Generation & Combination:**
  - **Volume:** 1:1 with booking confirmations
  - **Variety:** Financial data from Payment, Booking, and Currency tables
  - **Frequency:** Synchronous with booking completion
  - **Data Sources:** `Payment` + `Booking` + `Currency` + `Payment_Method` → Combined via booking_id foreign key

6.1.2 Analytical Outputs

Revenue Dashboards for Hosts

- **Structure:** Time-series charts showing earnings, occupancy rates, booking trends

- **Data Aggregation & Combination:**
  - **Volume:** 5-10MB of aggregated data daily
  - **Variety:** Historical data from Payment, Booking, and Property tables
  - **Frequency:** Daily batch processing
  - **Data Aggregation:** `Payment.amount` + `Booking.status_id` + `Property.host_id` → Aggregated by date and host dimensions using GROUP BY

Guest Behavior Analytics

- **Structure:** User segmentation reports, search pattern analysis, conversion metrics
- **Data Generation & Combination:**
  - **Volume:** 100KB per user historical data
  - **Variety:** User behavior data from Booking, Property_Review, and search logs
  - **Frequency:** Batch processing every 6 hours
  - **Data Combination:** `User.user_id` + `Booking` + `Property_Review.rating` → Combined for pattern analysis using user-based aggregations

6.1.3 Administrative Outputs

Moderation Reports

- **Structure:** Tabular data with moderation actions, timestamps, resolution status
- **Data Generation & Combination:**
  - **Volume:** 2-5MB of log data daily
  - **Variety:** Administrative data from Admin_Action_Log, User, and Property_Review tables
  - **Frequency:** Event-driven generation
  - **Data Sources:** `Admin_Action_Log` + `User` + `Property_Review` → Combined via target_user_id and target_review_id relationships

6.1.4 Recommendation Outputs

Personalized Property Suggestions

- **Structure:** Ranked property listings with relevance scores
- **Data Combination & Generation:**
  - **Volume:** 500-1,000 recommendation requests per minute
  - **Variety:** Multi-source data from Property, Property_Amenity, Amenity, and Property_Review tables

- **Frequency:** Real-time generation per user request
- **Data Combination:** `Property` + `Property_Amenity` + `Amenity` + `Property_Review.rating` → Combined for real-time scoring using weighted averages

Table 1: Information Requirements Summary

Output	Data Operations & Source Tables	Volume/Frequency	Data Variety
Booking Confirmation	**Combine:** `Booking` + `Property` + `User` + `Property_Type`	50-100 daily Real-time	Structured (4+ tables)
Revenue Dashboard	**Aggregate:** `Payment` + `Booking` + `Property` (`host_id`)	5-10MB daily Batch	Historical + Financial
Guest Analytics	**Combine:** `User` + `Booking` + `Property_Review`	100KB/user 6-hour batches	Structured + Behavioral
Recommendations	**Combine:** `Property` + `Property_Amenity` + `Amenity` + `Property_Review`	500-1000/min Real-time	Multi-source contextual
Moderation Reports	**Combine:** `Admin_Action_Log` + `User` + `Property_Review`	2-5MB daily Event-driven	Mixed structured content

6.2. Improvements

We reoriented this section to focus on the volume, variety, and frequency of required information, clearly describing expected outputs and specifying what data needs to be generated and combined.

7. Query Proposals

The system employs a hybrid database approach where **PostgreSQL** handles structured, transactional data with ACID compliance, while **Redis** provides high-performance caching, real-time features, and session management. This combination leverages the strengths of both SQL and NoSQL paradigms to deliver optimal performance and scalability.

7.1. 1. Guest Information

7.1.1 User Profiles

PostgreSQL (SQL - Source of Truth):

```
SELECT
    u.user_id,
    u.name,
    u.email,
    u.status,
    u.created_at
FROM
    User u
JOIN
    Role r ON u.role_id = r.role_id
WHERE
    r.role_name = 'guest'
    AND u.status = 'active'
ORDER BY
    u.created_at DESC;
```

Purpose: Retrieves recently registered and active guest users. **Insight:** Supports user monitoring and personalized onboarding.

Redis (NoSQL - Performance Layer):

```
-- Cache user profile for fast session access
SET user:123:profile '{"name":"John Doe","email":"john@email.
  com","status":"active"}'
EXPIRE user:123:profile 3600

-- Retrieve cached profile
GET user:123:profile
```

Purpose: Provides sub-millisecond access to user data for authentication and session management. **Complementarity:** PostgreSQL stores the authoritative user data, while Redis caches frequently accessed profiles to reduce database load.

—

7.1.2 Property Listings

PostgreSQL (SQL - Complex Queries):

```
SELECT
    p.property_id,
    p.title,
    p.price_per_night,
    p.city,
    p.country
FROM
    Property p
WHERE
    p.city = 'Bogot '
    AND p.price_per_night BETWEEN 100000 AND 250000
    AND p.status = 'active';
```

Purpose: Retrieves available properties within a given price range and location. **Insight:** Powers search and filter functionality in guest discovery.

Redis (NoSQL - Real-time Search):

```
-- Store property IDs in sorted sets by price for fast range
   queries
ZADD listings:bogota 100000 "property:123" 150000 "property
   :456" 200000 "property:789"

-- Query properties by price range
ZRANGEBYSCORE listings:bogota 100000 250000

-- Cache full property details
HSET property:123 title "Luxury Apartment" city "Bogot "
   price 150000
```

Purpose: Enables real-time property discovery with minimal latency. **Complementarity:** PostgreSQL handles complex filtering and joins, while Redis provides instant access to pre-computed search results.

—

7.1.3 Booking Records

PostgreSQL (SQL - Transactional Integrity):

```
SELECT
    b.booking_id,
    u.name AS guest_name,
    p.title AS property_title,
    b.check_in,
    b.check_out,
    bs.code AS booking_status
FROM
    Booking b
JOIN
    User u ON b.guest_id = u.user_id
JOIN
    Property p ON b.property_id = p.property_id
JOIN
    Booking_Status bs ON b.status_id = bs.status_id
WHERE
    bs.code = 'CONFIRMED';
```

Purpose: Tracks confirmed bookings with associated guest and property details. **Insight:** Enables occupancy analysis and performance metrics.

Redis (NoSQL - Session State):

```
-- Store active booking session for fast dashboard access
HMSET booking:current:1234
    guest_name "John Doe"
    property_title "Beach Villa"
    check_in "2024-03-15"
    check_out "2024-03-20"
EXPIRE booking:current:1234 86400
```

```
-- Retrieve booking session
HGETALL booking:current:1234
```

Purpose: Maintains real-time booking state for user dashboards and quick access. **Complementarity:** PostgreSQL ensures booking data consistency, while Redis provides instant access to current user sessions.

—

7.1.4 Payment Data

PostgreSQL (SQL - Financial Records):

```
SELECT
    p.payment_id,
    p.booking_id,
    p.amount,
    pm.name AS method,
    p.status,
    p.payment_date
FROM
    Payment p
JOIN
    Payment_Method pm ON p.payment_method_id = pm.method_id
WHERE
    p.payment_date >= NOW() - INTERVAL '30 days';
```

Purpose: Fetches recent payments for revenue tracking and reconciliation. **Insight:** Supports financial audits and refund management.

Redis (NoSQL - Real-time Analytics):

```
-- Real-time revenue tracking
INCRBYFLOAT revenue:daily:2024-03-15 150.50
INCRBYFLOAT revenue:monthly:2024-03 150.50
INCRBYFLOAT revenue:host:42 150.50

-- Get real-time dashboard metrics
GET revenue:daily:2024-03-15
ZREVRANGE revenue:hosts 0 4 WITHSCORES  -- Top 5 hosts by
   revenue
```

Purpose: Provides instant analytics and revenue tracking without database queries. **Complementarity:** PostgreSQL maintains authoritative financial records, while Redis enables real-time business intelligence.

—

7.1.5 Guest Reviews

PostgreSQL (SQL - Structured Reviews):

```
SELECT
    r.review_id,
    u.name AS guest,
    p.title AS property,
    r.rating,
    r.comment
FROM
    Property_Review r
JOIN
    Booking b ON r.booking_id = b.booking_id
JOIN
    User u ON r.guest_id = u.user_id
JOIN
    Property p ON b.property_id = p.property_id
WHERE
    r.rating >= 4
    AND r.status = 'approved';
```

Purpose: Identifies top-rated stays and guest experiences. **Insight:** Enhances trust and supports property ranking algorithms.

Redis (NoSQL - Review Aggregations):

```
-- Store property rating aggregates for fast ranking
ZADD property:ratings 4.8 "property:123" 4.5 "property:456"
   4.2 "property:789"

-- Increment rating counters when new reviews arrive
HINCRBY property:123:reviews total_ratings 1
HINCRBY property:123:reviews sum_ratings 5

-- Calculate average rating
local total = redis.call('HGET', 'property:123:reviews', '
   total_ratings')
local sum = redis.call('HGET', 'property:123:reviews', '
   sum_ratings')
return sum / total
```

Purpose: Maintains real-time review statistics for property ranking. **Complementarity:** PostgreSQL stores detailed review data, while Redis provides instant access to aggregated ratings.

7.2. 2. Host Information

7.2.1 Property Data

PostgreSQL (SQL - Property Management):

```
SELECT
    property_id,
    title,
    price_per_night,
    status
FROM
    Property
WHERE
    host_id = 42;
```

Purpose: Displays all properties managed by a specific host. **Insight:** Enables property management and performance tracking.

Redis (NoSQL - Host Session):

```
-- Cache host's property list for dashboard
SADD host:42:properties "property:123" "property:456" "
   property:789"
EXPIRE host:42:properties 3600

-- Quick property count for host
SCARD host:42:properties
```

Purpose: Provides instant access to host property lists. **Complementarity:** PostgreSQL manages property data, while Redis caches host-specific views.

—

7.2.2 Availability Calendar

PostgreSQL (SQL - Availability Management):

```
SELECT
    property_id,
    start_date,
    end_date,
    is_available
FROM
    Property_Availability
WHERE
    property_id = 42;
```

Purpose: Shows availability and blocked dates for host's property. **Insight:** Prevents double-booking and aids scheduling.

Redis (NoSQL - Real-time Availability):

```
-- Store available dates in sorted sets for fast conflict
   checking
SADD property:42:available_dates "2024-03-15" "2024-03-16" "
   2024-03-17"

-- Block dates when booking is made
SREM property:42:available_dates "2024-03-15"

-- Check instant availability
SISMEMBER property:42:available_dates "2024-03-15"
```

Purpose: Provides real-time availability checks with minimal latency. **Complementarity:** PostgreSQL maintains the authoritative availability schedule, while Redis enables instant conflict detection.

—

7.2.3 Revenue and Booking History

PostgreSQL (SQL - Financial Reporting):

```sql
SELECT
    u.user_id AS host_id,
    SUM(pay.amount) AS total_revenue,
    COUNT(b.booking_id) AS total_bookings
FROM
    Booking b
JOIN
    Property p ON b.property_id = p.property_id
JOIN
    User u ON p.host_id = u.user_id
JOIN
    Payment pay ON b.booking_id = pay.booking_id
WHERE
    u.user_id = 42
GROUP BY
    u.user_id;
```

Purpose: Calculates host's total earnings and booking count. **Insight:** Supports payout reports and dashboard summaries.

Redis (NoSQL - Real-time Metrics):

```
-- Real-time host revenue tracking
INCRBYFLOAT host:42:revenue:current_month 250.75
INCRBYFLOAT host:42:bookings:current_month 1

-- Get host performance metrics
GET host:42:revenue:current_month
GET host:42:bookings:current_month
```

Purpose: Provides instant revenue and booking metrics for host dashboards. **Complementarity:** PostgreSQL ensures accurate financial reporting, while Redis delivers real-time performance data.

7.3. 3. System-Generated Information

7.3.1 Recommendations

PostgreSQL (SQL - Recommendation Logic):

```sql
SELECT
    p.property_id,
    p.title
FROM
    Property p
WHERE
    p.city = 'Medell n'
ORDER BY
    p.price_per_night ASC
LIMIT 5;
```

Purpose: Returns five recommended properties based on location and pricing. **Insight:** Supports personalized property discovery.

Redis (NoSQL - Recommendation Caching):

```
-- Cache personalized recommendations for users
LPUSH recommendations:user:123 "property:456" "property:789" "
   property:101"
EXPIRE recommendations:user:123 1800  -- 30 minutes

-- Get cached recommendations
LRANGE recommendations:user:123 0 -1

-- Store trending properties
ZINCRBY trending:properties 1 "property:456"
ZREVRANGE trending:properties 0 9  -- Top 10 trending
   properties
```

Purpose: Delivers instant property recommendations without database queries. **Complementarity:** PostgreSQL computes recommendation logic, while Redis serves pre-computed results instantly.

—

7.3.2 Notifications

Redis (NoSQL - Real-time Messaging):

```
-- Publish real-time notifications
PUBLISH notifications:user:123 '{"type":"booking_confirmed","
   message":"Your booking #456 is confirmed!"}'

-- Store notification history
LPUSH user:123:notifications '{"type":"booking_confirmed","
   timestamp":"2024-03-15T10:30:00Z"}'
LTRIM user:123:notifications 0 49  -- Keep last 50
   notifications

-- Real-time online users tracking
SADD online:users "user:123"
EXPIRE online:users 300  -- Refresh every 5 minutes
```

Purpose: Enables real-time notifications and user presence tracking. **Complementarity:** While PostgreSQL stores notification history, Redis handles real-time delivery and presence management.

7.4. 4. Administrator Information

7.4.1 User Management Data

PostgreSQL Query:

```
SELECT
    u.user_id,
    u.name,
    r.role_name,
    u.status
FROM
```

```
    User u
JOIN
    Role r ON u.role_id = r.role_id
WHERE
    r.role_name IN ('host', 'guest');
```

Purpose: Retrieves user roles and their active statuses. **Insight:** Supports administrative user management.

7.4.2 Analytics and Reports

PostgreSQL Query:

```
SELECT
    COUNT(DISTINCT u.user_id) AS total_users,
    COUNT(DISTINCT b.booking_id) AS total_bookings,
    SUM(p.amount) AS total_revenue
FROM
    Payment p
JOIN
    Booking b ON p.booking_id = b.booking_id
JOIN
    User u ON b.guest_id = u.user_id;
```

Purpose: Generates key performance indicators for the platform. **Insight:** Enables data-driven decision-making and business analysis.

Redis Query:

```
GET dashboard:active_users
```

Purpose: Retrieves real-time count of currently active users.

7.4.3 Audit Logs

PostgreSQL (SQL - Comprehensive Audit Trail):

```
SELECT
    aal.action_id,
    a.name AS admin_name,
    u.name AS target_user_name,
    p.title AS target_property_title,
    r.comment AS target_review_text,
    at.name AS action_type,
    aal.description AS reason,
    aal.created_at
FROM Admin_Action_Log aal
JOIN User a ON aal.admin_id = a.user_id
JOIN Admin_Action_Type at ON aal.action_type_id = at.
   action_type_id
LEFT JOIN User u ON aal.target_user_id = u.user_id
LEFT JOIN Property p ON aal.target_property_id = p.property_id
LEFT JOIN Property_Review r ON aal.target_review_id = r.
   review_id
WHERE aal.created_at >= NOW() - INTERVAL '30 days'
```

```
ORDER BY aal.created_at DESC;
```

Purpose: Provides comprehensive audit trail of all administrative actions. **Insight:** Ensures accountability and compliance with data protection regulations.

Redis (NoSQL - Real-time Audit Monitoring):

```
-- Track real-time admin activity for security monitoring
INCR audit:actions:today
ZADD audit:recent_actions 1710500000 "admin:42:deleted_review
   :789"
ZREMRANGEBYSCORE audit:recent_actions 0 1710400000  -- Clean
   old entries

-- Alert on suspicious activity patterns
INCR admin:42:actions:last_hour
EXPIRE admin:42:actions:last_hour 3600

-- Real-time admin dashboard metrics
ZREVRANGE audit:active_admins 0 9 WITHSCORES  -- Top 10 active
    admins
```

Purpose: Enables real-time security monitoring and anomaly detection. **Complementarity:** PostgreSQL stores the complete audit history, while Redis provides real-time monitoring and alerting capabilities.

7.5. Database Paradigm Synergy

The hybrid approach demonstrates how SQL and NoSQL databases complement each other:

- **PostgreSQL excels at:** Complex joins, transactional integrity, structured data, ACID compliance, and comprehensive audit trails
- **Redis excels at:** Real-time operations, caching, session management, high-throughput reads, and real-time monitoring
- **Combined benefit:** PostgreSQL ensures data consistency and auditability while Redis delivers performance at scale and real-time insights

This architecture supports the platform's requirements for both data integrity (bookings, payments, audit logs) and real-time performance (search, notifications, analytics, security monitoring).

7.6. Improvements

We expanded the query proposals to include complete examples from both database paradigms, demonstrating how SQL and NoSQL complement each other in our design.

8. Concurrency Analysis

8.1. Possible concurrent access to data

- **Property Booking:** Multiple users trying to book the same property for overlapping dates, causing a race condition.

  a) Both transactions check availability and find the property available.

  b) Both proceed to create booking records.

  c) Result: Double booking, property over-committed.

- **Property Availability Updates:** When the status of a property changes on a date, the availability of the property must be updated with minimal latency. For example, when a host updates the availability of a property (e.g., blocks a date) at the same time a guest is trying to book, we might have a race condition.

  a) Host updates property availability while guest is completing booking.

  b) Booking system reads stale availability data.

  c) Result: Booking conflicts with updated availability, requiring manual resolution.

- **User Account Updates:** Multiple devices or sessions updating the same user profile.

  a) User updates profile picture from mobile app while updating contact information from web browser.

  b) Both sessions read current profile state.

  c) Result: One update overwrites the other, causing data loss.

- **Reviews and Ratings:** Multiple reviews being submitted for the same property or user at the same time.

  a) Multiple review submissions for the same booking due to network retries or user impatience.

  b) System processes duplicate review requests.

  c) Result: Multiple reviews for single booking, violating one-review-per-booking rule.

- **Payment Processing:** Concurrent payment attempts for the same booking.

  a) User initiates multiple payment requests for same booking due to interface or network lag.

  b) Payment system processes multiple transactions.

  c) Result: Duplicate charges, requiring refund processing and customer support.

8.2. Data Access Conflicts

Property Booking – Proposed Concurrency Controls:

a) **High-Isolation Database Transactions:** It is advisable to perform the booking workflow under a high isolation level (e.g., *SERIALIZABLE*), as this level can help prevent phenomena such as non-repeatable reads or phantom rows during property availability checks.

b) **Pessimistic Locking:** A write lock on the property's availability interval may be employed during the booking process. This approach provides mutual exclusion, though it may also introduce lock contention under high concurrency conditions.

c) **Optimistic Concurrency Control:** Property availability can be validated both at the beginning of the workflow and again prior to committing the transaction. If the version or timestamp of the availability record has changed, the transaction may be aborted to avoid a lost-update anomaly.

d) **Conditional State-Based Updates:** An atomic conditional update (e.g., `UPDATE ... WHERE is_available = TRUE`) may be utilized to ensure that the write is only performed when the resource remains free, thereby mitigating race conditions without relying on explicit locks.

Property Availability Updates – Proposed Concurrency Controls:

a) **High-Isolation Database Transactions:** Similar to the booking workflow, transactions and locking mechanisms can be employed. The host's update may involve verifying the current availability state to ensure that, if a booking already exists, the host is not permitted to block that date (or, alternatively, that the booking operation fails if the host's block is considered authoritative).

   A similar locking mechanism (e.g., `SELECT FOR UPDATE`) may be utilized when the host attempts to modify availability. This approach allows any ongoing booking process for the same dates to either complete or abort before the host's update is applied.

User Account Updates – Proposed Concurrency Controls:

a) **Logical Locks via Advisory Locks:** For sensitive updates (e.g., changing email, password), take a logical lock on the user profile using Postgres advisory locks keyed by the user ID. This provides mutual exclusion while avoiding table-level or row-level blocking under normal usage.

b) **Distributed Locks or Edit Tokens (Redis):** For operations involving multiple steps or requiring intensive user interaction, a short-lived distributed lock (e.g., Redlock in Redis) can be used. This approach helps prevent two devices from simultaneously executing a complex update sequence, such as uploading a profile photo and modifying contact information at the same time.

   Alternatively, an "edit token" can be stored in Redis to indicate that the profile is currently being edited. Then, when the user wants to edit their profile, the Redis database is always validated first before allowing the edit, reducing the likelihood of conflicting updates.

Reviews and Ratings – Proposed Concurrency Controls:

a) **Booking-Scoped Conditional Inserts:** A conditional insertion strategy can be employed in which a review is only stored if no existing review is associated with the corresponding booking. This can be implemented through an atomic constraint (such as a unique index on booking_id) or via an insert-with-check operation. If a concurrent submission attempts to create an additional review, the database automatically rejects the duplicate, ensuring consistency across replicas and backend services.

b) **Short-Lived Locks:** For workflows that may trigger repeated review submissions within a very short timeframe, a temporary submission lock may be introduced in Redis and keyed by the corresponding booking identifier. When the first submission arrives, the lock is created and subsequent submissions are temporarily deferred for a brief delay interval. This delay provides the backend with enough time to process the initial request and verify whether a review has already been persisted for the booking.

   After the delay, each deferred request performs a validation step against the write-primary database to determine whether a review now exists. If a review has already been recorded, the deferred request is safely discarded to uphold the one-review-per-booking constraint. If no review is found, the request may proceed normally.

Payment Processing – Proposed Concurrency Controls:

a) **Idempotent Payment Requests:** An idempotency mechanism may be introduced so that repeated payment attempts associated with the same booking produce a single canonical result. This can be achieved by issuing an idempotency key for the payment operation and recording it alongside the transaction metadata. If concurrent or repeated attempts are submitted with the same idempotency key, the backend returns the previously generated response rather than initiating an additional charge, thereby preventing duplicate payments.

b) **Short-Lived Payment Locks:** For workflows that may trigger multiple payment submissions within a short interval—often due to network instability or user retries—a temporary payment lock may be placed in Redis and keyed by the booking identifier. When the initial payment request is received, the lock is created and subsequent requests are deferred for a brief delay interval. This delay allows the backend to complete the initial payment validation and determine whether a successful charge has already been recorded.

   After the delay, each deferred request performs a verification step against the write-primary database to confirm the current payment state. If a charge has already been processed for the booking, the deferred attempt is safely discarded to uphold the single-payment-per-booking requirement. If no charge exists, the deferred request may proceed with payment processing under normal conditions.

9. Parallel and Distributed Database Design

9.1. Distributed Database Design

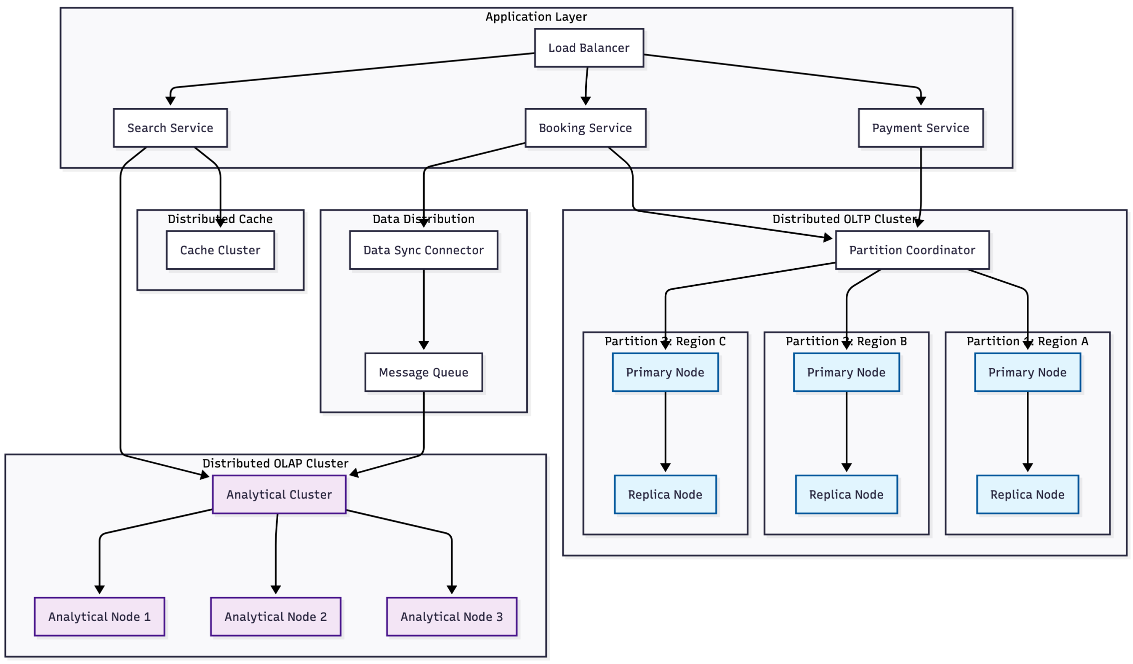


Figure 20: Distributed architecture.

9.2. Distributed architecture

The adoption of a distributed architecture is based on the scalability requirements identified during business analysis. The system must handle projected growth of 50 properties monthly and a transactional volume of 500 to 1,000 monthly bookings, with peaks of 300 read operations per minute. These demands justify the transition from a centralized architecture to a distributed model that allows progressive horizontal scaling.

The geographically segmented access pattern inherent to the vacation rental business favors a region-based distribution. Users predominantly interact with properties within their area of interest, enabling natural data partitioning that reduces latency and improves performance through access locality.

9.3. Distribution Strategy

The architecture implements a horizontal partitioning scheme using composite keys that combine regional identifiers with property identifiers. This strategy ensures that related entities remain in the same partition, minimizing costly distributed operations. Each partition operates as an independent unit, containing properties, bookings, and users from a specific region.

Within each regional partition, additional partitioning by temporal criteria is applied for fast-growing entities. This two-level approach enables efficient data lifecycle management and maintains optimized performance for active operations while automatically archiving historical data.

9.4. Parallelization and Optimization

The distributed design enables parallel query execution across multiple nodes. Search and query operations are automatically distributed among relevant partitions, where each node processes its local data segment. Partial results are consolidated in a coordinator node before delivering the final response to the user.

To separate conflicting workloads, specialized clusters are implemented: one optimized for transactional processing and another for analytical queries. This physical separation prevents complex reporting operations from affecting the performance of business-critical transactions.

9.5. Availability and Resilience

The distributed nature provides fault tolerance through synchronous replication within each partition. Regional fault containment ensures that disruptions in one geographical zone do not compromise global system availability. Consistency strategies are adapted to the operation type, employing strong consistency for critical transactions and eventual consistency for less sensitive read operations.

9.6. Operational Advantages

Economically, the distributed model offers better cost scalability compared to vertical hardware scaling. The ability to add capacity incrementally through new partitions proves more efficient than upgrading high-end servers. Operationally, it enables maintenance and updates without global interruptions, as operations can be performed on individual partitions while others remain active.

Phased implementation mitigates transition risks, starting with read replicas for immediate benefits and progressing toward full distribution as the strategy is validated with real data. This controlled approach progressively manages the inherent complexity of distributed systems.

9.7. Conclusion

The distributed architecture represents a balanced solution that addresses immediate performance requirements while establishing foundations for future growth. The combination of geographical partitioning, strategic replication, and workload separation creates a system that scales elegantly with business growth, ensuring consistent performance and high availability as market demands evolve.

10. Performance Improvement Strategies

To enhance system performance as data volume and user concurrency increase, the architecture incorporates several strategies based on parallelism, distribution, and workload isolation.

1. **Horizontal Scaling (Distributed Application Servers)**
   Multiple stateless application servers run in parallel and share workload through a load balancer.
   Performace Boost:
   Requests are processed concurrently, enabling the system to handle spikes in

traffic.
Trade-offs:

- Requires shared session storage (cache or tokens).
- Increases complexity in deployment and monitoring.

Basis for implementation:
User-facing operations such as search and booking need low latency even when traffic increases.

2. **Data Partitioning (Sharding / Table Partitioning)**
   Large tables—such as Bookings or Payments—are horizontally partitioned (by month, region, or property ID).
   Performace Boost:

   - Reduces index size → faster queries
   - Enables parallel scans
   - Minimizes lock contention on hot tables

   Trade-offs:

   - Cross-partition queries may become slower
   - Requires more DB administration → partition lifecycle, cleanup

   Basis for implementation:
   Booking and payment tables grow quickly; partitioning ensures predictable performance over time.

3. **Replication for Read Scalability (Read Replica & Failover)**
   A read-only replica offloads analytic and search queries from the primary database.
   Performace Boost:

   - Isolates read and write workloads
   - Ensures transactional DB is dedicated to booking integrity
   - Enables parallel read operations

   Trade-offs:

   - Replication delay (eventual consistency)
   - Requires monitoring of replica lag

   Basis for implementation:
   Search queries represent the majority of transactions; isolating them prevents booking failures.

4. **Distributed Caching (Redis Cluster))**
   Hot data—popular properties, availability, sessions—is stored in distributed in-memory nodes.
   Performace Boost:

- Sub-millisecond access
- Reduces load on DB
- Handles massive read concurrency

Trade-offs:

- Cache invalidation complexity
- Possible stale data if TTL is too long

Basis for implementation:
Property discovery is read-intensive; caching is the fastest way to scale reads.

5. **Event-Driven Architecture (Message Queues)**
Heavy tasks (emails, notifications, ETL triggers, recommendation jobs) run asynchronously.
Performace Boost:

- Frees application servers from blocking operations
- Improves response times
- Enables concurrent worker execution

Trade-offs:

- Requires monitoring of queues
- Eventual consistency (small delays)

Basis for implementation:
Booking and payment operations must remain fast; asynchronous tasks prevent bottlenecks.

11. References

References

[1] IBM. (2025, July 25). *What is a data architecture?*. IBM Think. Disponible en: `https://www.ibm.com/think/topics/data-architecture`

[2] Corporate Finance Institute. (s. f.). *Business Model Canvas Examples*. Disponible en: `https://corporatefinanceinstitute.com/resources/management/business-model-canvas-examples/`

[3] Autor(es). (Año). *Designing Machine Learning Systems: An Iterative Process for Production-Ready Applications*. Editorial/Editorial académica. *[Completar con autor y año cuando estén disponibles]*.

[4] Autor desconocido. (s. f.). *[Figura sobre arquitectura de datos]*. Substack. Disponible en: `https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Feb15f23f-a99f-406b-add4-eaffe0d1633d_2332x2696.png`

[5] YouTube. (2025, September 27). *Hotel Reservation (AirBnb, Booking.com) - System Design Interview Question* [Video]. YouTube. Disponible en: https://www.youtube.com/watch?v=m67Mjbx6DMY

[6] Bocconi University. (s. f.). *The rise of the sharing economy: Estimating the impact of Airbnb on the hotel industry.* IGIER Bocconi Working Paper No. 684. Disponible en: https://igier.unibocconi.eu/sites/default/files/media/publication/684.pdf

[7] Radical Storage. (s. f.). *Airbnb Statistics: Key Figures and Insights on the Global Market.* Disponible en: https://radicalstorage.com/travel/airbnb-statistics/

[8] Business of Apps. (s. f.). *Airbnb Statistics (2025).* Disponible en: https://www.businessofapps.com/data/airbnb-statistics

[9] Uber Engineering. (s. f.). *Inside Uber's Big Data Platform.* Disponible en: https://www.uber.com/en-CO/blog/uber-big-data-platform/

[10] Coronel, C., & Morris, S. (2014). *Database Systems: design, implementation, & management.* Cengage Learning.