

Final Project Report

Project Name: Hotel Booking Website

Course: Internet Programming

University: Institute of Technology of Cambodia

Lecturer: Mr. CHUN Thavorac

Team Members

No.	Student Name	Student ID
1	Sam Sokleap	P20230031
2	Kheang Ann	P20230027

Table of Contents

1. [Introduction](#)
 2. [System Design](#)
 3. [Technology Stack](#)
 4. [Features](#)
 5. [Third-Party Integration](#)
 6. [Security Implementation](#)
 7. [Business Logic](#)
 8. [Deployment](#)
 9. [Challenges & Solutions](#)
 10. [Future Plans](#)
 11. [Conclusion](#)
 12. [Appendix](#)
-

1. Introduction

1.1 Problem Statement

In Cambodia's growing tourism industry, many hotels face significant challenges in managing their online presence and booking processes. Our research identified four critical pain points that this system aims to address:

1.1.1 Discovery Problem (Source)

Many Cambodian travelers still rely on **phone calls and social media** (Facebook, Telegram) to find and book hotels during their vacations. This creates several issues:

- **Limited Exposure:** Hotels without websites miss potential customers who search online
- **Inconsistent Information:** Hotel details scattered across multiple social media posts

- **Time-Consuming Process:** Customers must contact each hotel individually to check availability
- **Language Barriers:** International tourists struggle with Khmer-only Facebook pages

1.1.2 Cost Problem (Price)

Most small to medium-sized hotels in Cambodia **cannot afford to build and maintain their own booking website:**

- **Development Costs:** Custom website development costs \$3,000-\$15,000+
- **Hosting Expenses:** Monthly server and domain costs add up over time
- **Technical Expertise:** Hotels lack IT staff to manage and update systems
- **Payment Integration:** Implementing secure payment gateways requires technical knowledge

Our Solution: Provide a **free platform** where hotels can list their properties, upload images, and receive bookings without any development or hosting costs.

1.1.3 Operations Problem (Management)

Handling bookings, payments, and customer communications creates operational overhead:

- **Manual Tracking:** Bookings tracked in notebooks or spreadsheets
- **Payment Collection:** Cash-only or bank transfer payments are difficult to track
- **Communication Gaps:** Missed calls and delayed responses lose customers
- **No Analytics:** Hotels lack data to optimize pricing and occupancy

Our Solution: Centralized dashboard handling **payment processing, booking management, and automated notifications** - reducing administrative burden on hotel staff.

1.1.4 Confidence Problem (Trust)

Both customers and hotels need assurance when transacting online:

- **Customer Concerns:** "Will the hotel honor my booking?" "Is my payment secure?"
- **Hotel Concerns:** "Will the customer actually show up?" "How do I verify payment?"
- **No Accountability:** Without a platform, disputes are difficult to resolve

Our Solution:

- **Secure Stripe payments** with verified transactions
- **Rating and review system** for accountability
- **Booking confirmation workflow** with status tracking
- **Platform acts as intermediary** for dispute resolution

1.2 Objectives

The primary objectives of the Hotel Booking Website project are:

1. **Develop a User-Friendly Booking Platform:** Create an intuitive web interface where guests can easily search for hotels, view room availability, and complete bookings with minimal friction.

2. **Implement Real-Time Room Availability Management:** Build a sophisticated availability algorithm that prevents double-bookings and accurately tracks room inventory across overlapping date ranges.
3. **Integrate Secure Payment Processing:** Implement Stripe payment gateway to handle secure credit card transactions with proper webhook handling and payment status tracking.
4. **Create a Comprehensive Admin Dashboard:** Develop a full-featured administrative interface for managing hotels, rooms, users, bookings, amenities, and viewing business analytics.
5. **Ensure System Security:** Implement robust authentication (including OAuth 2.0), authorization, and data protection mechanisms to safeguard user information and prevent unauthorized access.
6. **Build a Scalable Architecture:** Design a modular, maintainable system architecture that can scale to accommodate future growth and feature additions.
7. **Support Multi-Hotel Operations:** Enable the system to manage multiple hotels with individual configurations, room types, amenities, and pricing structures.
8. **Implement Rating and Review System:** Allow guests to provide feedback on their stays, helping future customers make informed decisions.

1.3 Scope

In Scope

The Hotel Booking Website includes the following functionalities:

User-Facing Features:

- User registration with email/password or Google OAuth 2.0
- User authentication and profile management
- Hotel search with filters (destination, dates, guest count)
- Room browsing with availability checking
- Multi-room booking capability
- Secure payment processing via Stripe
- Booking management (view, cancel)
- Hotel rating and review submission
- Password reset functionality

Admin Features:

- Admin authentication with role-based access control
- Dashboard with analytics and charts
- Hotel CRUD operations (Create, Read, Update, Delete)
- Room CRUD operations with image management
- User management (view, activate/deactivate)
- Booking management (approve, reject, view)
- Amenity management for hotels and rooms
- Bed type configuration
- Payment/bill monitoring

System Features:

- Automated booking expiration handling
- Real-time room availability calculation
- Image storage (local development, Cloudinary for production)
- Tax calculation (10% tax rate)
- Booking status lifecycle management

Out of Scope

The following features are not included in the current version:

- Responsive web applications (iOS/Android)
- Multi-language support
- Multi-currency support
- KHQR payment method
- Staff and hotel owner roles
- Room service management
- Loyalty/rewards program
- Inventory management for hotel supplies
- Chat/messaging system between guests and hotels

2. System Design

2.1 Project Structure

The project follows a monorepo structure with clearly separated frontend and backend codebases:

```
Hotel-Booking-System/
├── docker-compose.yml
├── .env                                # Env for Backend
└── README.md

└── backend/                             # NestJS Backend API
    ├── package.json
    └── src/
        ├── main.ts
        ├── app.module.ts
        ├── config/          # Database, Cloudinary, Upload configs
        │   ├── auth/         # Authentication & User management
        │   ├── hotels/        # Hotels CRUD
        │   ├── rooms/         # Rooms CRUD
        │   ├── booking/       # Booking management
        │   ├── payment/       # Stripe payment integration
        │   ├── ratings/        # Reviews & Ratings
        │   ├── amenities/      # Amenities management
        │   └── bed-types/      # Bed types configuration
        └── uploads/           # Local file storage (dev)
```

```

└── frontend/
    ├── user/                      # User-facing Vue.js App
    │   ├── package.json
    │   ├── src/
    │   │   ├── views/              # Page components
    │   │   ├── components/        # Reusable components
    │   │   ├── stores/             # Pinia state management
    │   │   ├── router/             # Vue Router
    │   │   └── utils/              # Utilities
    │   └── public/
    │       └── .env                # env for user frontend
    └── admin/                     # Admin Vue.js App
        ├── package.json
        ├── src/
        │   ├── views/              # Admin pages
        │   ├── components/         # Admin components
        │   ├── stores/              # State management
        │   ├── router/              # Admin routing
        │   └── composable/          # Reusable logic
        └── public/
            └── .env                # env for admin frontend

```

2.2 System Architecture

The Hotel Booking System follows a **three-tier architecture** with clear separation of concerns:

PRESNTATION LAYER (Vue.js 3)

- User Frontend: Vue Router, Pinia Store, Axios Client
- Admin Frontend: Vue Router, Pinia Store, Chart.js

APPLICATION LAYER (NestJS Backend API)

- Middleware: Helmet Security, CORS, JWT Auth, Validation Pipe
- Modules: Auth, Hotels, Rooms, Booking, Payment, Ratings, Amenities, BedTypes, Admin
- Scheduled Tasks: Payment Expiration Check (Cron)

DATA LAYER

- PostgreSQL (Neon DB) - Primary database
- Stripe - Payment gateway
- Cloudinary - Image CDN

EXTERNAL SERVICES

- Google OAuth 2.0 - Social authentication

Data Flow:

- User Frontend → (HTTPS/REST) → Backend
- Admin Frontend → (HTTPS/REST) → Backend

- Backend → PostgreSQL, Stripe, Cloudinary, Google OAuth

2.3 Architecture Patterns

The system employs several established software architecture patterns:

2.3.1 Backend Architecture Pattern: Modular MVC

The NestJS backend follows a **Modular MVC (Model-View-Controller)** pattern:

Layer	Responsibility	Implementation
Controllers	Handle HTTP requests, route to services	<code>*.controller.ts</code> files
Services	Business logic, data processing	<code>*.service.ts</code> files
Entities	Data models, database schema	<code>entities/*.entity.ts</code>
DTOs	Request/Response validation	<code>dto/*.dto.ts</code>
Guards	Authorization middleware	<code>guards/*.guard.ts</code>
Strategies	Authentication strategies	<code>strategies/*.strategy.ts</code>

2.3.2 Frontend Architecture Pattern: Component-Based with Composition API

The Vue.js frontends utilize:

Pattern	Purpose	Implementation
Composition API	Logic reuse, better TypeScript support	<code><script setup></code> syntax
Pinia State Management	Centralized state, reactive stores	<code>stores/*.ts</code>
Vue Router	Client-side routing, route guards	<code>router/index.ts</code>
Composables	Extractable, reusable logic	<code>composables/*.ts</code>

2.3.3 Database Design Pattern: Relational with TypeORM

- Entity-Relationship Model** with normalized tables
- TypeORM Repository Pattern** for data access
- Query Builder** for complex queries
- Auto-synchronization** for development (disabled in production)

2.4 Entity Relationship Diagram (ERD)

Entity Relationships:

Parent Entity	Relationship	Child Entity
USER	has many	USER_ROLE
ROLE	has many	USER_ROLE

Parent Entity	Relationship	Child Entity
USER	creates many	BOOKING
USER	makes many	PAYMENT
USER	writes many	RATING
BOOKING	contains many	BOOKING_ITEM
BOOKING	has one	PAYMENT
BOOKING	receives one	RATING
HOTEL	has many	ROOM
HOTEL	has many	HOTEL_AMENITY
HOTEL	receives many	RATING
ROOM	booked in many	BOOKING_ITEM
ROOM	has many	ROOM_AMENITY
ROOM	has many	ROOM_BED
AMENITY	assigned to many	HOTEL_AMENITY, ROOM_AMENITY
BED_TYPE	assigned to many	ROOM_BED

Entity Attributes:

```
USER {
    uuid id PK
    string email UK
    string password
    string firstName
    string lastName
    string provider
    string profileImage
    boolean isActive
    datetime createdAt
    datetime updatedAt
}
```

```
ROLE {
    uuid id PK
    string name UK
}
```

```
BOOKING {
    uuid id PK
    uuid userId FK
    decimal totalPrice
    enum status
    string guestPhone
```

```
    date guestDOB
    datetime createdAt
    datetime confirmedAt
    datetime paymentExpiresAt
    string rejectionReason
}

BOOKING_ITEM {
    uuid id PK
    uuid bookingId FK
    uuid roomId FK
    string roomName
    string hotelName
    date checkIn
    date checkOut
    decimal priceAtBooking
    int quantity
}

PAYMENT {
    uuid id PK
    uuid bookingId FK
    uuid userId FK
    decimal amount
    enum status
    string paymentMethod
    string stripePaymentId
    string stripeSessionId
    string transactionId
    datetime createdAt
}

HOTEL {
    uuid id PK
    string name UK
    text description
    string destination
    string location
    string email
    string phone
    decimal avgRating
    array images
    boolean isActive
    datetime createdAt
}

ROOM {
    uuid id PK
    uuid hotelId FK
    string name
    text description
    decimal price
    int available
    int maxOccupancy
}
```

```

    decimal discountPercentage
    array images
    boolean isActive
}

AMENITY {
    uuid id PK
    string name UK
    string icon
    enum category
}

BED_TYPE {
    uuid id PK
    string name UK
    string icon
}

RATING {
    uuid id PK
    uuid hotelId FK
    uuid userId FK
    uuid bookingId FK
    decimal overallScore
    int service
    int facilities
    int comfort
    int value
    int location
    text comment
    datetime createdAt
}

```

2.5 API Architecture

The backend exposes a **RESTful API** with the following module endpoints:

Module	Base URL	Description
Auth	/auth	User authentication, registration, OAuth
Admin	/admin	Admin-specific operations
Hotels	/hotels	Hotel CRUD and search
Rooms	/rooms	Room management and availability
Booking	/bookings	Booking lifecycle management
Payment	/payments	Stripe payment processing
Ratings	/ratings	Hotel reviews and ratings
Amenities	/amenities	Hotel/Room amenity management

Module	Base URL	Description
Bed Types	/bed-types	Bed configuration management

3. Technology Stack

3.1 Backend Technologies

Technology	Version	Purpose
NestJS	11.x	Backend framework with modular architecture
TypeScript	5.7.x	Type-safe JavaScript development
Node.js	20.x	JavaScript runtime environment
TypeORM	0.3.x	Object-Relational Mapping for database
PostgreSQL	16.x	Primary relational database
Passport	11.x	Authentication middleware
JWT	11.x	JSON Web Token authentication
bcrypt	6.x	Password hashing
Stripe SDK	20.x	Payment processing
class-validator	0.14.x	DTO validation
class-transformer	0.5.x	Object transformation
Helmet	8.x	HTTP security headers
Multer	2.x	File upload handling
Cloudinary SDK	1.41.x	Cloud image storage
@nestjs/schedule	6.x	Cron job scheduling

3.2 Frontend Technologies

Technology	Version	Purpose
Vue.js	3.5.x	Frontend framework with Composition API
TypeScript	5.9.x	Type-safe development
Vue Router	4.6.x	Client-side routing
Pinia	3.x	State management
Axios	1.13.x	HTTP client for API calls
Vite	7.x	Build tool and dev server
Chart.js	4.5.x	Analytics charts (Admin)

Technology	Version	Purpose
vue-chartjs	5.3.x	Chart.js Vue wrapper
VueDatePicker	12.x	Date selection component
Remixicon	4.8.x	Icon library
jwt-decode	4.x	JWT payload extraction

3.3 Development Tools

Tool	Purpose
Visual Studio Code	Primary code editor/IDE
GitHub	Version control and repository hosting
GitHub Copilot	AI-powered code completion and assistance
ESLint	Code linting and style enforcement
Prettier	Code formatting
Jest	Backend unit testing
Vitest	Frontend unit testing
Postman	API testing and documentation

3.4 DevOps & Deployment

Technology	Purpose
Render	Backend & Frontend hosting (Web Service + Static Sites)
Neon	PostgreSQL database hosting
Cloudinary	Image CDN and storage
GitHub Actions	CI/CD (Auto-deploy on push)
Docker	Local development containerization
Docker Compose	Multi-container development

4. Features

4.1 User Features

4.1.1 Authentication & Authorization

Feature	Description
---------	-------------

Feature	Description
User Registration	Create account with email/password validation (min 8 characters, uppercase, lowercase, number)
Email/Password Login	Secure login with bcrypt password verification
Google OAuth 2.0	One-click sign-in with Google account
JWT Session Management	Stateless authentication with 1-day token expiration
Password Reset	Email-based password recovery with secure tokens
Profile Management	Update personal information and profile picture
Account Deactivation Handling	Graceful handling when admin deactivates user account

4.1.2 Hotel Discovery

Feature	Description
Destination Search	Filter hotels by destination/location
Date-Based Search	Search with check-in and check-out dates
Guest Count Filter	Filter rooms by occupancy capacity
Hotel Listing	Browse all available hotels with images and ratings
Hotel Details	View hotel information, amenities, location, and contact
Room Browsing	View available room types with prices and amenities
Real-Time Availability	See actual room availability for selected dates

4.1.3 Booking System

Feature	Description
Multi-Room Selection	Add multiple rooms to a single booking
Price Calculation	Real-time price calculation with discount and tax (10%)
Booking Summary	Review booking details before confirmation
Booking Creation	Submit booking request with guest information
Booking History	View all past and current bookings
Booking Details	View individual booking information and status
Booking Cancellation	Cancel pending or confirmed bookings
Status Tracking	Monitor booking status (Pending → Confirmed → Completed)

4.1.4 Payment Processing

Feature	Description
Stripe Checkout	Secure payment via Stripe Checkout Session
Payment Status	Track payment status (Pending, Completed, Failed)
Payment Expiration	1-hour window to complete payment after admin approval
Automatic Failure Handling	Booking marked as failed if payment not completed in time

4.1.5 Rating & Review System

Feature	Description
Post-Stay Rating	Rate hotels after completed stays
Category Scores	Rate service, facilities, comfort, value, location (1-10)
Overall Score	Automatic calculation of overall rating (1-5 stars)
Written Review	Leave detailed comments about the stay
Review Visibility	View other guests' reviews on hotel pages

4.2 Admin Features

4.2.1 Dashboard & Analytics

Feature	Description
Overview Dashboard	Central view of system statistics
Booking Statistics	Charts showing booking trends and status distribution
Revenue Analytics	Payment and revenue visualization
User Statistics	User registration and activity metrics
Chart.js Visualizations	Interactive charts for data analysis

4.2.2 Hotel Management

Feature	Description
Hotel List	View all hotels with search and filter
Create Hotel	Add new hotels with details and images
Edit Hotel	Modify hotel information and amenities
Delete Hotel	Remove hotels (with booking validation)
Image Management	Upload, reorder, and remove hotel images

Feature	Description
Status Toggle	Activate/deactivate hotels

4.2.3 Room Management

Feature	Description
Room List	View all rooms grouped by hotel
Create Room	Add rooms with pricing, beds, and amenities
Edit Room	Modify room details and configuration
Delete Room	Remove rooms (with active booking check)
Bed Configuration	Configure bed types and quantities
Discount Management	Set percentage discounts on rooms
Inventory Control	Set available room count

4.2.4 User Management

Feature	Description
User List	View all registered users
User Details	View user profile and booking history
Activate User	Re-enable deactivated user accounts
Deactivate User	Disable user access (prevents login)
Role Assignment	Manage user roles (admin/user)

4.2.5 Booking Management

Feature	Description
Booking List	View all bookings with filters
Booking Details	View full booking information
Approve Booking	Confirm pending bookings (starts 1-hour payment timer)
Reject Booking	Reject bookings with reason
Cancel Booking	Cancel confirmed bookings
Status Filtering	Filter by booking status

4.2.6 Amenity Management

Feature	Description
----------------	--------------------

Feature	Description
Hotel Amenities	Manage hotel-level amenities (WiFi, Pool, Parking)
Room Amenities	Manage room-level amenities (AC, TV, Minibar)
Icon Selection	Choose icons for amenities
Category Management	Organize amenities by category

4.2.7 Bed Type Management

Feature	Description
Bed Type List	View all bed types
Create Bed Type	Add new bed types (Single, Double, King, etc.)
Edit Bed Type	Modify bed type name and icon
Delete Bed Type	Remove unused bed types

4.2.8 Bill/Payment Monitoring

Feature	Description
Payment History	View all payment transactions
Payment Status	Monitor payment statuses
Transaction Details	View Stripe transaction IDs
Revenue Tracking	Track completed payments

5. Third-Party Integration

5.1 Stripe Payment Gateway

Purpose: Secure online payment processing for booking payments

Integration Type: Stripe Checkout Sessions with Webhooks

5.1.1 Implementation Details

```
// Stripe Service Configuration
@Injectable()
export class StripeService {
  private stripe: Stripe;

  constructor(private configService: ConfigService) {
    this.stripe = new Stripe(configService.get('STRIPE_SECRET_KEY'));
  }
}
```

```
}
```

5.1.2 Payment Flow

Payment Flow Steps:

1. User clicks "Pay Now" on Frontend
2. Frontend sends POST /payments/stripe/checkout to Backend
3. Backend creates Checkout Session with Stripe
4. Stripe returns session URL to Backend
5. Backend returns checkout URL to Frontend
6. Frontend redirects User to Stripe Checkout Page
7. User completes payment on Stripe
8. Stripe sends Webhook (payment_intent.succeeded) to Backend
9. Backend updates Payment Status = COMPLETED
10. Backend updates Booking Status = COMPLETED
11. Stripe redirects User to Success Page

5.1.3 Key Features

Feature	Implementation
Checkout Sessions	Stripe-hosted payment page for security
Webhook Handling	Real-time payment status updates
Signature Verification	Validate webhook authenticity
Payment Intents	Stripe Payment Intents API for SCA compliance
Metadata	Store booking ID in payment metadata

5.1.4 Environment Variables

```
STRIPE_SECRET_KEY=sk_test_xxxxxx
STRIPE_WEBHOOK_SECRET=whsec_xxxxxx
STRIPE_SUCCESS_URL=your-frontend-url/payment/success
STRIPE_CANCEL_URL=your-frontend-url/payment/cancel
```

5.2 Google OAuth 2.0

Purpose: Social login allowing users to sign in with their Google accounts

5.2.1 Implementation Details

```
// Google Strategy Configuration
@Injectable()
export class GoogleStrategy extends PassportStrategy(Strategy, 'google') {
  constructor(private authService: AuthService) {
    super({
      clientID: process.env.GOOGLE_CLIENT_ID,
      clientSecret: process.env.GOOGLE_CLIENT_SECRET,
      callbackURL: process.env.GOOGLE_CALLBACK_URL,
      scope: ['email', 'profile'],
    });
  }
}
```

5.2.2 OAuth Flow

Google OAuth Flow Steps:

1. User clicks "Sign in with Google" on Frontend
2. Frontend redirects User to /auth/google
3. User sends GET /auth/google to Backend
4. Backend redirects User to Google Consent Screen
5. User authorizes the App on Google
6. Google sends Callback with auth code to Backend
7. Backend exchanges code for user info with Google
8. Google returns profile data to Backend
9. Backend creates or finds User in database
10. Backend generates JWT Token
11. Backend redirects User with JWT token
12. User is now logged in!

5.2.3 Environment Variables

```
FRONTEND_URL=your-frontend-url
GOOGLE_CLIENT_ID=xxxxxx.apps.googleusercontent.com
GOOGLE_CLIENT_SECRET=GOCSPX-xxxxxx
GOOGLE_CALLBACK_URL=your-backend-url/auth/google/redirect
```

5.3 Cloudinary (Cloud Image Storage)

Purpose: Cloud-based image storage for hotel, room, and profile images in production

5.3.1 Implementation Details

```
// Cloudinary Configuration
cloudinary.config({
  cloud_name: process.env.CLOUDINARY_CLOUD_NAME,
```

```

    api_key: process.env.CLOUDINARY_API_KEY,
    api_secret: process.env.CLOUDINARY_API_SECRET,
  });

// Cloudinary Storage for Multer
export function createCloudinaryStorage(folderName: string) {
  return new CloudinaryStorage({
    cloudinary: cloudinary,
    params: async (req, file) => ({
      folder: `hotel-booking/${folderName}`,
      allowed_formats: ['jpg', 'jpeg', 'png', 'webp'],
      transformation: [{ width: 1200, height: 800, crop: 'limit' }],
    }),
  });
}

```

5.3.2 Features

Feature	Description
Automatic Upload	Images uploaded directly to Cloudinary
Image Transformation	Automatic resizing (1200x800 max)
Format Support	JPG, JPEG, PNG, WEBP formats
Folder Organization	Separate folders for hotels, rooms, profiles
CDN Delivery	Global CDN for fast image loading
Cleanup	Automatic deletion of replaced images

5.3.3 Environment Variables

```

CLOUDINARY_CLOUD_NAME=your-cloud-name
CLOUDINARY_API_KEY=xxxxxx
CLOUDINARY_API_SECRET=xxxxxx

```

6. Security Implementation

6.1 Authentication Security

6.1.1 Password Security

Measure	Implementation
Hashing Algorithm	bcrypt with salt rounds (default: 10)
Password Requirements	Minimum 8 characters, uppercase, lowercase, number

Measure	Implementation
No Plain Text Storage	Passwords never stored in plain text
Password Reset Tokens	Secure random tokens with expiration

```
// Password Hashing
const saltRounds = 10;
const hashedPassword = await bcrypt.hash(password, saltRounds);

// Password Verification
const isValid = await bcrypt.compare(inputPassword, hashedPassword);
```

6.1.2 JWT Token Security

Measure	Implementation
Token Signing	HMAC-SHA256 with secret key
Token Expiration	1-day expiration period
Payload Minimization	Only essential user data in payload
Stateless Verification	No server-side session storage

```
// JWT Configuration
JwtModule.register({
  secret: process.env.JWT_SECRET,
  signOptions: { expiresIn: '1d' },
})

// Token Payload
{
  email: user.email,
  sub: user.id,
  firstName: user.firstName,
  lastName: user.lastName,
}
```

6.2 Authorization Security

6.2.1 Guards Implementation

Guard	Purpose	Code
JwtAuthGuard	Validates JWT tokens on protected routes	<code>@UseGuards(JwtAuthGuard)</code>
RolesGuard	Enforces role-based access control	<code>@UseGuards(RolesGuard)</code>

Guard	Purpose	Code
Public Decorator	Marks routes as public (no auth required)	<code>@Public()</code>

```
// JwtAuthGuard Implementation
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  canActivate(context: ExecutionContext) {
    const isPublic = this.reflector.getAllAndOverride('isPublic', [
      context.getHandler(),
      context.getClass(),
    ]);
    if (isPublic) return true;
    return super.canActivate(context);
  }
}

// RolesGuard Implementation
@Injectable()
export class RolesGuard implements CanActivate {
  canActivate(ctx: ExecutionContext): boolean {
    const required = this.reflector.getAllAndOverride<string[]>(ROLES_KEY, [
      ctx.getHandler(),
      ctx.getClass(),
    ]);
    if (!required) return true;

    const user = ctx.switchToHttp().getRequest().user;
    const roles = user.roles?.map((ur) => ur.role?.name) ?? [];
    return required.some((r) => roles.includes(r));
  }
}
```

6.3 HTTP Security Headers

6.3.1 Helmet.js Configuration

```
// Security Headers
app.use(
  helmet({
    crossOriginOpenerPolicy: { policy: 'same-origin-allow-popups' },
  }),
);
```

Headers applied by Helmet:

- X-DNS-Prefetch-Control
- X-Frame-Options

- X-Content-Type-Options
- X-XSS-Protection
- Strict-Transport-Security
- Referrer-Policy

6.4 CORS Configuration

```
app.enableCors({
  origin: corsOrigins, // Configurable via CORS_ORIGIN env
  methods: 'GET,HEAD,PUT,PATCH,POST,DELETE,OPTIONS',
  credentials: true,
  allowedHeaders: 'Content-Type, Authorization',
});
```

6.5 Input Validation

6.5.1 Global Validation Pipe

```
app.useGlobalPipes(
  new ValidationPipe({
    whitelist: true,           // Strip unknown properties
    transform: true,           // Auto-transform payloads to DTO types
  }),
);
```

6.5.2 DTO Validation Examples

```
// Registration Validation
export class UserRegisterDto {
  @IsEmail()
  email: string;

  @IsString()
  @MinLength(8)
  @Matches(/([A-Z]/, { message: 'Password must contain uppercase letter' })
  @Matches(/([a-z]/, { message: 'Password must contain lowercase letter' })
  @Matches(/(\d)/, { message: 'Password must contain number' })
  password: string;

  @IsString()
  @Length(1, 50)
  firstName: string;

  @IsString()
  @Length(1, 50)
  lastName: string;
}
```

```
// Booking Validation
export class CreateBookingDto {
  @IsArray()
  @ValidateNested({ each: true })
  @Type(() => RoomSelectionDto)
  roomSelections: RoomSelectionDto[];

  @IsOptional()
  @IsDateString()
  guestDateOfBirth?: string;
}
```

6.6 SQL Injection Prevention

TypeORM's parameterized queries prevent SQL injection:

```
// Safe - Parameterized Query
const user = await this.userRepository.findOne({
  where: { email: inputEmail }
});

// Safe - Query Builder with Parameters
const result = await this.bookingItemRepository
  .createQueryBuilder('bookingItem')
  .where('bookingItem.roomId = :roomId', { roomId: id })
  .andWhere('bookingItem.checkIn < :checkOut', { checkOut })
  .getMany();
```

6.7 Stripe Webhook Security

```
// Webhook Signature Verification
const signature = request.headers['stripe-signature'];
const event = this.stripe.webhooks.constructEvent(
  request.rawBody,
  signature,
  this.configService.get('STRIPE_WEBHOOK_SECRET'),
);
```

6.8 File Upload Security

```
// Allowed formats validation
params: async (req, file) => ({
  folder: `hotel-booking/${folderName}`,
  allowed_formats: ['jpg', 'jpeg', 'png', 'webp'],
  transformation: [{ width: 1200, height: 800, crop: 'limit' }],
})
```

6.9 Account Security

```
// Deactivated Account Check
if (!user.isActive) {
    throw new ForbiddenException(
        'Your account has been deactivated. Please contact the administrator.'
);
}
```

7. Business Logic

7.1 Booking Status Lifecycle

The booking system implements a comprehensive status lifecycle:

Booking Status Lifecycle:

Current State	Action/Trigger	Next State	Notes
(Start)	User Creates Booking	PENDING	Rooms reserved
PENDING	Admin Approves	CONFIRMED	1hr payment window starts
PENDING	Admin Rejects	CANCELLED	Rooms released
PENDING	User Cancels	CANCELLED	Rooms released
CONFIRMED	Payment Success	COMPLETED	Booking active, stay confirmed
CONFIRMED	Payment Timeout (1hr)	FAILED	Rooms released (auto by cron job)
CONFIRMED	User Cancels	CANCELLED	Rooms released
COMPLETED	-	(End)	-
CANCELLED	-	(End)	-
FAILED	-	(End)	-

Status Codes:

- **PENDING:** Awaiting admin review, rooms blocked
- **CONFIRMED:** Approved, 1-hour payment window
- **COMPLETED:** Payment received, booking active
- **CANCELLED:** Manually cancelled by user/admin
- **FAILED:** Payment timeout or system failure

Status Details

Status	Description	Room Impact	Next Actions
PENDING	Booking created, awaiting admin approval	Reserved (blocked)	Admin: Approve/Reject; User: Cancel
CONFIRMED	Admin approved, awaiting payment	Still reserved	User: Pay/Cancel; System: Auto-fail
COMPLETED	Payment successful, booking active	Booked	User: Rate after stay
CANCELLED	Cancelled by user or admin	Released	None
FAILED	Payment timeout (auto)	Released	User: Create new booking

7.2 Room Availability Algorithm

The system implements a sophisticated overlap detection algorithm:

```
// Core Availability Logic
async getBookedRoomCount(roomId: string, checkIn: Date, checkOut: Date): Promise<number> {
  const today = new Date();

  // Count rooms that are blocked for the given date range
  const overlappingBookings = await this.bookingItemRepository
    .createQueryBuilder('bookingItem')
    .innerJoin('bookingItem.booking', 'booking')
    .where('bookingItem.roomId = :roomId', { roomId })
    .andWhere(
      // Include PENDING, CONFIRMED, or active COMPLETED bookings
      '(booking.status IN (:...activeStatuses) OR ' +
      '(booking.status = :completed AND bookingItem.checkOut > :today))',
      {
        activeStatuses: ['pending', 'confirmed'],
        completed: 'completed',
        today: today,
      }
    )
    // Date overlap detection
    .andWhere('bookingItem.checkIn < :checkOut', { checkOut })
    .andWhere('bookingItem.checkOut > :checkIn', { checkIn })
    .getCount();

  return overlappingBookings;
}

async getAvailableRoomCount(roomId: string, checkIn: Date, checkOut: Date): Promise<number> {
  const room = await this.roomsRepository.findOne({ where: { id: roomId } });
  if (!room) return 0;

  const bookedCount = await this.getBookedRoomCount(roomId, checkIn, checkOut);
```

```

    return Math.max(0, room.available - bookedCount);
}

```

Date Overlap Formula

Two date ranges overlap when:

```
existingCheckIn < newCheckOut AND existingCheckOut > newCheckIn
```

Visual representation:

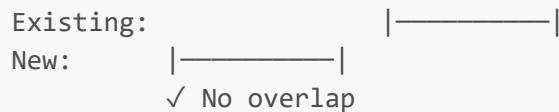
Case 1: Overlap (New booking overlaps existing)



Case 2: No Overlap (New booking starts after existing ends)



Case 3: No Overlap (New booking ends before existing starts)



7.3 Price Calculation

```

// Price Calculation Formula
const calculatePrice = (room, nights) => {
  // Step 1: Apply discount to base price
  const basePrice = Number(room.price);
  const discount = room.discountPercentage || 0;
  const discountedPrice = basePrice * (1 - discount / 100);

  // Step 2: Calculate room price for nights
  const roomPrice = discountedPrice * nights;

  // Step 3: Sum all rooms
  const subtotal = sumOfAllRoomPrices;

  // Step 4: Apply 10% tax
  const tax = subtotal * 0.10;

  // Step 5: Calculate total
  const total = subtotal + tax;
}

```

```
    return { subtotal, tax, total };
};
```

Example Calculation:

Room: Deluxe Suite
 Base Price: \$200/night
 Discount: 15%
 Nights: 3

Discounted Price: $\$200 \times (1 - 0.15) = \$170/\text{night}$
 Room Total: $\$170 \times 3 = \510
 Subtotal: \$510
 Tax (10%): \$51
 Total: \$561

7.4 Payment Expiration System

Automated task scheduler for payment monitoring:

```
@Injectable()
export class BookingTasksService {
  @Cron(CronExpression.EVERY_MINUTE)
  async handleExpiredPayments() {
    const now = new Date();

    // Find CONFIRMED bookings past payment deadline
    const expiredBookings = await this.bookingRepository.find({
      where: {
        status: BookingStatus.CONFIRMED,
        paymentExpiresAt: LessThan(now),
      },
    });

    for (const booking of expiredBookings) {
      // Check if payment was completed
      const payment = await this.paymentRepository.findOne({
        where: {
          bookingId: booking.id,
          status: PaymentStatus.COMPLETED,
        },
      });

      if (!payment) {
        // Mark as FAILED - rooms automatically released
        booking.status = BookingStatus.FAILED;
        booking.rejectionReason =
          'Payment not completed within 1 hour. Reserved rooms have been
          released.';
      }
    }
  }
}
```

```
        await this.bookingRepository.save(booking);
    }
}
}
}
```

7.5 Rating Calculation

```
// Calculate overall score from category ratings
private calculateOverallScore(dto: CreateRatingDto): number {
  const { service, facilities, comfort, value, location } = dto;

  // Average of 5 categories (scale 1-10)
  const avgCategoryScore = (service + facilities + comfort + value + location) / 5;

  // Convert to 1-5 scale
  const overallScore = Math.round((avgCategoryScore / 2) * 10) / 10;

  return overallScore;
}

// Update hotel average rating
async updateHotelRating(hotelId: string): Promise<void> {
  const ratings = await this.ratingRepository.find({ where: { hotelId } });

  const avgRating = ratings.length > 0
    ? ratings.reduce((sum, r) => sum + r.overallScore, 0) / ratings.length
    : 0;

  await this.hotelRepository.update(hotelId, { avgRating });
}
```

7.6 Booking Validation Rules

Validation	Rule	Error Message
Age Check	Guest must be 16+ years old	"Guest must be at least 16 years old to make a booking"
Date Validation	Check-out must be after check-in	"Check-out date must be after check-in date"
Availability	Requested quantity \leq available count	"Room only has X available for the selected dates"
Status Transition	Only valid status changes allowed	"Only pending bookings can be confirmed"

Validation	Rule	Error Message
Duplicate Rating	One rating per booking	"You have already rated this booking"
Duplicate Entries	Unique hotel/room names	"Hotel/Room with name already exists"

8. Deployment

8.1 Deployment Architecture

Deployment Overview:

Component	Platform	Type
User Frontend (Vue.js)	Render	Static Site
Admin Frontend (Vue.js)	Render	Static Site
Backend API (NestJS)	Render	Web Service
PostgreSQL	Neon DB	External Service
Payments	Stripe	External Service
Images	Cloudinary	External Service

CI/CD Pipeline:

1. Developer pushes code to GitHub Repository
2. Render triggers Auto-Build (npm install → npm build)
3. Render deploys updated services automatically

Data Flow:

- Users → User Frontend / Admin Frontend → Backend API → PostgreSQL, Stripe, Cloudinary

8.2 Backend Deployment (Render)

Platform: Render Web Service

Repository: <https://github.com/Sokleap-SAM/Hotel-Booking-System>

Branch: production

Setting	Value
Root Directory	backend
Build Command	npm install && npm run build
Start Command	npm run start:prod
Environment	Node

Setting	Value
Auto-Deploy	Yes (on Git push to production branch)

Backend URL: <https://hotel-booking-system-qjxx.onrender.com>

8.3 User Frontend Deployment (Render)

Platform: Render Static Site

Repository: <https://github.com/Sokleap-SAM/Hotel-Booking-System>

Branch: production

Setting	Value
Root Directory	frontend/user
Build Command	npm install && npm run build
Publish Directory	dist
Auto-Deploy	Yes (on Git push to production branch)

Rewrite Rule (for SPA routing):

- Source: `/*`
- Destination: `/index.html`
- Action: Rewrite

User Frontend URL: <https://cambook.onrender.com>

8.4 Admin Frontend Deployment (Render)

Platform: Render Static Site

Repository: <https://github.com/Sokleap-SAM/Hotel-Booking-System>

Branch: production

Setting	Value
Root Directory	frontend/admin
Build Command	npm install && npm run build
Publish Directory	dist
Auto-Deploy	Yes (on Git push to production branch)

Rewrite Rule (for SPA routing):

- Source: `/*`
- Destination: `/index.html`
- Action: Rewrite

Admin Frontend URL: <https://cambook-admin.onrender.com>

8.5 Database Deployment (Neon PostgreSQL)

Platform: Neon

Version: PostgreSQL 16

Setting	Value
Region	Same as Render backend
SSL Mode	Require
Connection Pooling	Enabled

8.6 Image Storage (Cloudinary)

Setting	Value
Provider	Cloudinary
Folder Structure	hotel-booking/hotels, hotel-booking/rooms, hotel-booking/profiles
Transformations	Auto-resize to 1200x800 max
Allowed Formats	jpg, jpeg, png, webp
CDN	Global delivery

8.7 Environment Variables

Backend (Render)

```
# Database (Neon PostgreSQL)
DATABASE_URL=postgresql://user:password@host/database?sslmode=require

# Authentication
JWT_SECRET=your-secure-jwt-secret-key

# Google OAuth 2.0
GOOGLE_CLIENT_ID=xxxxxx.apps.googleusercontent.com
GOOGLE_CLIENT_SECRET=GOCSPX-xxxxxx
GOOGLE_CALLBACK_URL=https://your-backend.onrender.com/auth/google/redirect
# Stripe Payment
STRIPE_SECRET_KEY=sk_live_xxxxxx
STRIPE_WEBHOOK_SECRET=whsec_xxxxxx
STRIPE_SUCCESS_URL=https://your-user-frontend.onrender.com/payment/success
STRIPE_CANCEL_URL=https://your-user-frontend.onrender.com/payment/cancel

# Cloudinary
CLOUDINARY_CLOUD_NAME=your-cloud-name
CLOUDINARY_API_KEY=xxxxxx
CLOUDINARY_API_SECRET=xxxxxx

# CORS
```

```
CORS_ORIGIN=https://your-user-frontend.onrender.com,https://your-admin-frontend.onrender.com

# Frontend URL (for OAuth redirect)
FRONTEND_URL=https://your-user-frontend.onrender.com
```

User Frontend (Render)

```
VITE_API_URL=https://your-backend.onrender.com
```

Admin Frontend (Render)

```
VITE_API_URL=https://your-backend.onrender.com
```

8.8 Local Development (Docker Compose)

For local development, Docker Compose is available:

```
# docker-compose.yml
services:
  backend:
    build: ./backend
    ports:
      - "3000:3000"
    depends_on:
      - postgres
    env_file:
      - .env

  postgres:
    image: postgres:16
    environment:
      POSTGRES_DB: ${DB_NAME}
      POSTGRES_USER: ${DB_USER}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

8.9 Deployment URLs Summary

Service	Environment	URL
Backend API	Production	[Place URL Here]
User Frontend	Production	[Place URL Here]
Admin Frontend	Production	[Place URL Here]
Database	Production	[Neon Connection String]
Cloudinary	Production	[Dashboard URL]
Backend API	Development	http://localhost:3000
User Frontend	Development	http://localhost:5173
Admin Frontend	Development	http://localhost:5174

9. Challenges & Solutions

9.1 Validation Handling

Challenge

Implementing comprehensive input validation across the entire application to prevent invalid data from entering the system while providing meaningful error messages to users.

Why Both Backend AND Frontend Validation?

We implemented validation on **both layers** for important reasons:

Layer	Purpose	Benefits
Frontend	User experience, instant feedback	Prevents unnecessary API calls, shows errors before submission
Backend	Security, data integrity	Protects against malicious requests, API consumers, bypassed frontend

Key Insight: Frontend validation can be bypassed (browser dev tools, direct API calls), so backend validation is the **security layer**. Frontend validation improves **user experience** by catching errors early.

Feature Example: User Registration Validation

Frontend Validation (Vue.js):

```
// User-facing form with immediate feedback
const validateRegistration = () => {
  const errors = {};

  // Email format check
  if (!form.email || !/^[^s@]+@[^\s@]+\.[^\s@]+$/ .test(form.email)) {
```

```

        errors.email = 'Please enter a valid email address';
    }

    // Password strength check
    if (form.password.length < 8) {
        errors.password = 'Password must be at least 8 characters';
    }
    if (!/[A-Z]/.test(form.password)) {
        errors.password = 'Password must contain an uppercase letter';
    }
    if (!/[a-z]/.test(form.password)) {
        errors.password = 'Password must contain a lowercase letter';
    }
    if (!/\d/.test(form.password)) {
        errors.password = 'Password must contain a number';
    }

    return { isValid: Object.keys(errors).length === 0, errors };
};

```

Backend Validation (NestJS DTO):

```

// Security layer - validates even if frontend is bypassed
export class UserRegisterDto {
    @IsEmail({}, { message: 'Please provide a valid email address' })
    email: string;

    @IsString()
    @MinLength(8, { message: 'Password must be at least 8 characters' })
    @Matches(/([A-Z])/, { message: 'Password must contain at least one uppercase
letter' })
    @Matches(/([a-z])/, { message: 'Password must contain at least one lowercase
letter' })
    @Matches(/\d/, { message: 'Password must contain at least one number' })
    password: string;

    @IsString()
    @Length(1, 50, { message: 'First name must be between 1 and 50 characters' })
    firstName: string;
}

```

Validation Flow:

Dual-Layer Validation Flow:

Step 1: User fills form

Step 2: Frontend Validation (Vue.js) - Instant feedback

- Check email format
- Check password strength

- Check required fields
- If invalid → Show error immediately to user
- If valid → Proceed to backend

Step 3: POST /auth/register to Backend

Step 4: Backend Validation (NestJS) - Security layer

- DTO decorators (class-validator)
- Type transformation
- Business rules validation
- If invalid → Return 400 Bad Request to Frontend
- If valid → Proceed to database

Step 5: Database Operation

- INSERT user into database
- Return success to Backend

Step 6: Response

- Backend returns 201 Created to Frontend
- Frontend shows success message to User

Other Validation Implementations

1. Global Validation Pipe (Backend):

```
// main.ts - Applied to ALL endpoints automatically
app.useGlobalPipes(
  new ValidationPipe({
    whitelist: true,          // Strip properties not in DTO
    transform: true,           // Auto-transform types
    forbidNonWhitelisted: false,
  }),
);
```

2. Nested Object Validation:

```
// Booking DTO with nested room selections
export class CreateBookingDto {
  @IsArray()
  @ValidateNested({ each: true })
  @Type(() => RoomSelectionDto)
  roomSelections: RoomSelectionDto[];

  @IsOptional()
  @IsDateString()
  guestDateOfBirth?: string;
}
```

3. Custom Business Logic Validation:

```
// Age validation in booking service
if (createBookingDto.guestDateOfBirth) {
    const age = calculateAge(createBookingDto.guestDateOfBirth);
    if (age < 16) {
        throw new BadRequestException(
            'Guest must be at least 16 years old to make a booking'
        );
    }
}
```

9.2 Redundant Handling (Duplicate Prevention)

Challenge

Preventing duplicate entries for critical data like hotel names, room names within hotels, user emails, and preventing double-booking of rooms.

Issues Faced

- Case-insensitive duplicate checking (e.g., "Hotel ABC" vs "hotel abc")
- Race conditions when multiple requests arrive simultaneously
- Complex duplicate detection for room availability across date ranges
- User trying to rate the same booking multiple times

Solution

1. Case-Insensitive Duplicate Checking:

```
// Using TypeORM Raw query for case-insensitive comparison
const existingHotel = await this.hotelsRepository.findOne({
    where: {
        name: Raw((alias) => `LOWER(${alias}) = LOWER(:name)`), {
            name: dto.name,
        },
    },
});

if (existingHotel) {
    throw new ConflictException(
        `Hotel with name "${dto.name}" already exists`
    );
}
```

2. Scoped Duplicate Prevention (Room within Hotel):

```
const existingRoom = await this.roomsRepository.findOne({
  where: {
    name: Raw((alias) => `LOWER(${alias}) = LOWER(:name)`), {
      name: createRoomDto.name,
    },
    hotelId: createRoomDto.hotelId, // Scope to hotel
  },
});
```

3. Room Availability Validation Before Booking:

```
// Validate availability for each room type before processing
for (const [roomId, requestedQuantity] of roomQuantities) {
  const availableCount = await this.roomsService.getAvailableRoomCount(
    roomId,
    checkIn,
    checkOut,
  );

  if (availableCount < requestedQuantity) {
    throw new BadRequestException(
      `Room "${room.name}" only has ${availableCount} available for the selected dates`
    );
  }
}
```

4. Rating Duplicate Prevention:

```
const existingRating = await this.ratingRepository.findOne({
  where: { bookingId: createRatingDto.bookingId, userId },
});

if (existingRating) {
  throw new BadRequestException('You have already rated this booking');
}
```

9.3 Google OAuth 2.0 Integration

Challenge

Implementing Google OAuth 2.0 authentication to allow users to sign in with their Google accounts while seamlessly integrating with the existing JWT-based authentication system.

Issues Faced

- Configuring Google Cloud Console credentials correctly
- Handling callback URL differences between development and production
- Managing users who sign up via Google vs email/password
- Cross-origin popup handling with security headers
- Missing profile information (some Google accounts don't expose full names)

Solution

1. Passport Google Strategy Configuration:

```
@Injectable()
export class GoogleStrategy extends PassportStrategy(Strategy, 'google') {
  constructor(private authService: AuthService) {
    super({
      clientID: process.env.GOOGLE_CLIENT_ID || 'placeholder-client-id',
      clientSecret: process.env.GOOGLE_CLIENT_SECRET || 'placeholder-secret',
      callbackURL: process.env.GOOGLE_CALLBACK_URL,
      scope: ['email', 'profile'],
      passReqToCallback: true,
    });
  }

  async validate(
    req: any,
    accessToken: string,
    refreshToken: string,
    profile: Profile,
    done: VerifyCallback,
  ): Promise<any> {
    const email = profile.emails?.[0]?.value;
    const firstName = profile.name?.givenName || '';
    const lastName = profile.name?.familyName || '';

    if (!email) {
      return done(new Error('No email found in Google profile'), false);
    }

    const user = await this.authService.validateOAuthLogin(
      email,
      'google',
      firstName,
      lastName,
    );
    done(null, user);
  }
}
```

2. Unified User Creation (OAuth + Local):

```

async validateOAuthLogin(
  email: string,
  provider: 'google',
  firstName: string,
  lastName: string,
): Promise<User> {
  let user = await this.userService.findByEmail(email);

  if (user) {
    // Existing user - check if active
    if (!user.isActive) {
      throw new ForbiddenException('Account deactivated');
    }
    return user;
  }

  // New user - create with OAuth provider
  const newUserDto = new UserRegisterDto();
  newUserDto.email = email;
  newUserDto.provider = provider;
  newUserDto.password = crypto.randomBytes(16).toString('hex'); // Random password
  newUserDto.firstName = firstName || '';
  newUserDto.lastName = lastName || '';

  user = await this.userService.create(newUserDto);
  return user;
}

```

3. Security Headers for OAuth Popup:

```

// Allow popups for Google OAuth
app.use(
  helmet({
    crossOriginOpenerPolicy: { policy: 'same-origin-allow-popups' },
  }),
);

```

4. Provider Field in User Entity:

```

@Column({ default: 'local' })
provider: 'local' | 'google';

```

9.4 Hotel Process Knowledge (Business Domain)

Challenge

Understanding and implementing the complex hotel booking business logic, including room inventory management, booking workflows, payment timelines, and status transitions that accurately reflect real-world hotel operations.

Issues Faced

- Understanding hotel room inventory (same room type with multiple units)
- Implementing correct date overlap detection for availability
- Defining appropriate booking status transitions
- Setting reasonable payment timeframes
- Handling cancellations and refunds appropriately
- Tax calculation standards

Solution

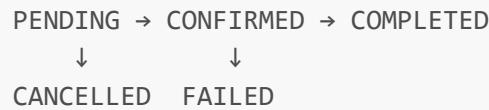
1. Room Inventory Model:

```
// Room entity with 'available' field for inventory count
@Entity()
export class Room {
    @Column()
    name: string; // Room type name (e.g., "Deluxe Suite")

    @Column({ default: 1 })
    available: number; // Number of rooms of this type (inventory)

    @Column({ type: 'decimal', precision: 10, scale: 2 })
    price: number; // Price per night
}
```

2. Booking Lifecycle Design:



Each status change has clear business rules:

- **PENDING:** Booking request submitted, awaiting admin approval
- **CONFIRMED:** Admin approved, user has 1 hour to pay
- **COMPLETED:** Payment successful, stay confirmed
- **CANCELLED:** User or admin cancelled
- **FAILED:** Payment timeout (automated)

3. Availability Algorithm:

```
// A room is "booked" if:
// 1. It's in a PENDING booking (blocking room for review)
// 2. It's in a CONFIRMED booking (awaiting payment)
// 3. It's in a COMPLETED booking AND checkout hasn't passed

async getBookedRoomCount(roomId: string, checkIn: Date, checkOut: Date) {
  const today = new Date();

  return await this.bookingItemRepository
    .createQueryBuilder('bookingItem')
    .innerJoin('bookingItem.booking', 'booking')
    .where('bookingItem.roomId = :roomId', { roomId })
    .andWhere(
      '(booking.status IN (:...activeStatuses) OR ' +
      '(booking.status = :completed AND bookingItem.checkOut > :today))',
      {
        activeStatuses: ['pending', 'confirmed'],
        completed: 'completed',
        today: today,
      }
    )
    .andWhere('bookingItem.checkIn < :checkOut', { checkOut })
    .andWhere('bookingItem.checkOut > :checkIn', { checkIn })
    .getCount();
}
```

4. Tax Implementation:

```
// Standard 10% tax rate
const tax = subtotal * 0.10;
const total = subtotal + tax;
```

5. Payment Expiration Logic:

```
// 1-hour payment window after admin approval
const paymentExpiresAt = new Date(now.getTime() + 60 * 60 * 1000);
booking.paymentExpiresAt = paymentExpiresAt;
```

10. Future Plans

10.1 Deeper Data Validation & Redundancy

Current State: Basic validation with class-validator decorators and manual duplicate checks.

Future Improvements:

Enhancement	Description
Database Constraints	Add unique indexes and constraints at database level
Transaction Support	Wrap multi-step operations in database transactions
Optimistic Locking	Implement version-based locking to prevent race conditions
Soft Delete	Implement soft delete for data recovery capabilities
Audit Logging	Track all changes with user, timestamp, and previous values

```
// Future: Optimistic Locking
@VersionColumn()
version: number;

// Future: Soft Delete
@DeleteDateColumn()
deletedAt: Date;
```

10.2 Microservice Architecture

Current State: Monolithic NestJS application with modular structure.

Migration Plan:

Planned Microservice Architecture:

Layer	Component	Responsibilities
Gateway	API Gateway (Kong/NGINX)	Load balancing, routing
Core Services	Auth Service	User Auth, JWT/OAuth, Profiles
Core Services	Booking Service	Bookings, Availability, Scheduling
Core Services	Payment Service	Stripe, KHQR, Refunds
Message Queue	RabbitMQ	Event Bus for async communication
Support Services	Hotel Service	Hotels, Rooms, Amenities
Support Services	Rating Service	Reviews, Analytics
Support Services	Notification Service	Email, SMS, Push

Communication Flow:

- API Gateway → Core Services (sync)
- Core Services → Message Queue → Support Services (async)

Benefits:

- Independent scaling of services

- Technology flexibility per service
- Improved fault isolation
- Easier team organization

10.3 More Management Features

Planned Features:

Feature	Description
Reporting Dashboard	Advanced analytics with PDF/Excel exports
Bulk Operations	Bulk edit prices, availability, status
Seasonal Pricing	Date-based dynamic pricing
Promotion Management	Create and manage promotional codes
Inventory Forecasting	AI-powered demand prediction
Email Templates	Customizable notification templates
Activity Logs	Comprehensive admin action logging

10.4 KHQR Payment Integration

Description: Cambodian QR payment standard for local payment support.

Implementation Plan:

KHQR Payment Flow Steps:

1. User selects KHQR Payment on Frontend
2. Frontend requests QR Code from Backend
3. Backend requests KHQR generation from Bank
4. Bank returns QR Data to Backend
5. Backend sends QR Code Response to Frontend
6. Frontend displays QR Code to User
7. User scans QR and pays via Banking App
8. Bank sends Webhook (Payment Confirmed) to Backend
9. Backend sends Payment Success Event to Frontend
10. Frontend shows Booking Confirmed to User

10.5 More Roles (Staff, Hotel Owner, Owner Staff)

Planned Role Hierarchy:

Planned Role Hierarchy:

Role	Reports To	Key Permissions
SUPER ADMIN	-	All access, System config, All hotels
ADMIN	Super Admin	All hotels, User mgmt, Analytics, Amenities

Role	Reports To	Key Permissions
HOTEL OWNER	Super Admin	Own hotels, Room mgmt, Staff mgmt, Reports
OWNER STAFF	Hotel Owner	View hotel, Manage rooms, View bookings, Check-in/out
STAFF	Owner Staff	View bookings, Check-in/out, Limited edit
USER (Guest)	-	Book rooms, View/cancel, Rate hotels

Permission Matrix:

Permission	Super Admin	Admin	Hotel Owner	Owner Staff	Staff	User
Manage System Settings	✓	X	X	X	X	X
Manage All Hotels	✓	✓	X	X	X	X
Manage Own Hotels	✓	✓	✓	X	X	X
Manage Users	✓	✓	X	X	X	X
Manage Staff	✓	X	✓	X	X	X
View Analytics	✓	✓	✓	✓	X	X
Manage Rooms	✓	✓	✓	✓	X	X
View Bookings	✓	✓	✓	✓	✓	Own
Process Check-in/out	✓	✓	✓	✓	✓	X
Make Booking	✓	✓	✓	✓	✓	✓
Rate Hotels	✓	✓	✓	✓	✓	✓

10.6 Additional Future Enhancements

Enhancement	Target Timeline	Description
Mobile App	Phase 2	React Native or Flutter mobile applications
Multi-language	Phase 2	i18n support (English, Khmer, Chinese)
Multi-currency	Phase 2	USD, KHR, and other currencies
Loyalty Program	Phase 3	Points system with rewards
Chat System	Phase 3	Real-time messaging between guests and hotels
Calendar Sync	Phase 3	Integration with Google/Outlook Calendar
Channel Manager	Phase 4	Integration with Booking.com, Agoda, Airbnb

11. Conclusion

The Hotel Booking Website project successfully delivers a comprehensive, modern hotel booking solution that addresses the core challenges faced by the hospitality industry. Through careful system design and implementation, the team has created a platform that serves both guests seeking accommodation and administrators managing hotel operations.

Key Achievements

1. **Full-Stack Implementation:** Complete end-to-end solution with Vue.js frontend applications and NestJS backend API, demonstrating proficiency in modern web development technologies.
2. **Robust Business Logic:** Implementation of complex hotel booking workflows including real-time availability checking, booking status lifecycle management, and automated payment expiration handling.
3. **Security-First Approach:** Multiple layers of security including JWT authentication, bcrypt password hashing, Google OAuth 2.0, role-based access control, and comprehensive input validation.
4. **Third-Party Integration:** Successful integration with Stripe for payment processing, Google for social authentication, and Cloudinary for cloud image storage.
5. **Scalable Architecture:** Modular design with clear separation of concerns, preparing the system for future microservice migration and feature expansion.
6. **Production-Ready Deployment:** Full deployment pipeline with Docker containerization for local development and cloud hosting on Render for backend and frontends, with PostgreSQL database on Neon.

Technical Accomplishments

- **10+ Database Entities** with complex relationships
- **9 Backend Modules** with complete CRUD operations
- **2 Frontend Applications** (User and Admin)
- **3 Authentication Methods** (Local, JWT, Google OAuth)
- **Automated Scheduling** for payment expiration handling
- **Real-Time Availability** calculation algorithm
- **Cloud Image Management** with transformation support

Learning Outcomes

Through this project, the team gained valuable experience in:

- Enterprise-level application architecture
- TypeScript and modern JavaScript frameworks
- Database design and ORM implementation
- RESTful API design and documentation
- Authentication and authorization patterns
- Payment gateway integration
- Cloud deployment and DevOps practices
- Agile development and teamwork

Future Direction

The project lays a solid foundation for future enhancements including microservice architecture migration, KHQR payment integration for the Cambodian market, expanded role management for hotel owners and staff, and mobile application development.

The Hotel Booking Website demonstrates the team's ability to deliver a production-quality full-stack application that meets real-world business requirements while maintaining code quality, security standards, and user experience excellence.

Appendix

A. Default Admin Credentials

The system automatically seeds a default administrator account on first startup:

Field	Value
Email	admin@cambook.kh
Password	Hello123!
Role	Admin

Security Note: Change the default admin password immediately after first login in production environment.

B. Status Code Reference

Booking Status

Status	Code	Description
Pending	pending	Awaiting admin approval
Confirmed	confirmed	Approved, awaiting payment
Completed	completed	Payment received, booking active
Cancelled	cancelled	Cancelled by user or admin
Failed	failed	Payment timeout or failure

Payment Status

Status	Code	Description
Pending	pending	Payment initiated
Completed	completed	Payment successful
Failed	failed	Payment failed

Status	Code	Description
Refunded	refunded	Payment refunded

C. Full API Endpoints Summary

Authentication (/auth)

Method	Endpoint	Auth	Description
POST	/auth/register	Public	User registration with email/password
POST	/auth/login	Public	User login (LocalStrategy)
GET	/auth/google	Public	Initiate Google OAuth 2.0
GET	/auth/google/redirect	Public	Google OAuth callback
GET	/auth/profile	User	Get current user profile
PATCH	/auth/profile	User	Update profile (with image upload)
PATCH	/auth/change-password	User	Change password

Admin Management (/admin)

Method	Endpoint	Auth	Description
GET	/admin/users	Admin	Get all users with filters
POST	/admin/users	Admin	Create new user
GET	/admin/users/stats	Admin	Get user statistics
GET	/admin/users/:id	Admin	Get user by ID
PATCH	/admin/users/:id	Admin	Update user
DELETE	/admin/users/:id	Admin	Delete user
PATCH	/admin/users/:id/roles	Admin	Update user roles
PATCH	/admin/users/:id/status	Admin	Update user status (active/inactive)
GET	/admin/roles	Admin	Get all roles

Hotels (/hotels)

Method	Endpoint	Auth	Description
GET	/hotels	Public	Get all active hotels
GET	/hotels/:id	Public	Get hotel by ID
POST	/hotels	Admin	Create hotel (with images)

Method	Endpoint	Auth	Description
PATCH	/hotels/:id	Admin	Update hotel
DELETE	/hotels/:id	Admin	Delete hotel
PATCH	/hotels/:id/status	Admin	Toggle hotel active status
GET	/hotels/admin/all	Admin	Get all hotels (including inactive)
GET	/hotels/admin/:id	Admin	Get hotel by ID (admin view)
GET	/hotels/search/availability	Public	Search hotels with availability
GET	/hotels/filter/lowest-price	Public	Get hotels sorted by lowest price
GET	/hotels/filter/highest-price	Public	Get hotels sorted by highest price
GET	/hotels/filter/highest-rating	Public	Get hotels sorted by rating
GET	/hotels/filter/highest-discount	Public	Get hotels with highest discounts
GET	/hotels/filter/by-amenities	Public	Filter hotels by amenities
GET	/hotels/filter/by-bed-type	Public	Filter hotels by bed types
GET	/hotels/filter/combined	Public	Combined filters with sort

Rooms (/rooms)

Method	Endpoint	Auth	Description
GET	/rooms/available	Public	Get all available rooms
GET	/rooms/:id	Public	Get room by ID
GET	/rooms/hotel/:hotelId	Public	Get rooms by hotel ID
GET	/rooms/hotel/:hotelId/availability	Public	Get rooms with availability check
GET	/rooms/:id/availability	Public	Check room availability for dates
POST	/rooms	Admin	Create room (with images)
PATCH	/rooms/:id	Admin	Update room
DELETE	/rooms/:id	Admin	Delete room

Bookings (/bookings)

Method	Endpoint	Auth	Description
POST	/bookings	User	Create new booking
GET	/bookings	User	Get user's bookings
GET	/bookings/:id	User	Get booking by ID

Method	Endpoint	Auth	Description
PATCH	/bookings/:id/cancel	User	Cancel booking
POST	/bookings/calculate-price	User	Calculate booking price
GET	/bookings/admin/all	Admin	Get all bookings
PATCH	/bookings/admin/:id/approve	Admin	Approve pending booking
PATCH	/bookings/admin/:id/reject	Admin	Reject pending booking

Payments ([/payments](#))

Method	Endpoint	Auth	Description
GET	/payments/:id/status	User	Get payment status
GET	/payments/booking/:bookingId	User	Get payment by booking ID
GET	/payments/my-payments	User	Get user's payments
PATCH	/payments/:id/cancel	User	Cancel payment
POST	/payments/:id/refund	User	Request refund
GET	/payments/admin/all	Admin	Get all payments
GET	/payments/admin/:id/details	Admin	Get payment details

Stripe Payments ([/payments/stripe](#))

Method	Endpoint	Auth	Description
POST	/payments/stripe/checkout	User	Create Stripe checkout session
GET	/payments/stripe/verify	User	Verify payment session
POST	/payments/stripe/webhook	Public	Stripe webhook handler

Ratings ([/ratings](#))

Method	Endpoint	Auth	Description
POST	/ratings	User	Create rating
GET	/ratings/:id	User	Get rating by ID
PATCH	/ratings/:id	User	Update rating
DELETE	/ratings/:id	User	Delete own rating
GET	/ratings/hotel/:hotelId	Public	Get ratings by hotel
GET	/ratings/hotel/:hotelId/user	User	Get user's rating for hotel

Method	Endpoint	Auth	Description
GET	/ratings/booking/:bookingId	User	Get rating by booking
GET	/ratings/my-ratings	User	Get user's ratings
DELETE	/ratings/admin/:id	Admin	Admin delete rating

Amenities ([/amenities](#))

Method	Endpoint	Auth	Description
GET	/amenities	Public	Get all amenities
GET	/amenities/:id	Public	Get amenity by ID
GET	/amenities/category/:category	Public	Get amenities by category
POST	/amenities	Admin	Create amenity
PATCH	/amenities/:id	Admin	Update amenity
DELETE	/amenities/:id	Admin	Delete amenity

Bed Types ([/bed-types](#))

Method	Endpoint	Auth	Description
GET	/bed-types	Public	Get all bed types
GET	/bed-types/:id	Public	Get bed type by ID
POST	/bed-types	Admin	Create bed type
PATCH	/bed-types/:id	Admin	Update bed type
DELETE	/bed-types/:id	Admin	Delete bed type

Health Check

Method	Endpoint	Auth	Description
GET	/	Public	API health check

D. Important Links

Source Code Repository

GitHub Repository: <https://github.com/Sokleap-SAM/Hotel-Booking-System>

Deployed Applications

Application	URL

Application	URL
User Frontend	https://cambook.onrender.com
Admin Frontend	https://cambook-admin.onrender.com
Backend API	https://hotel-booking-system-qjxx.onrender.com
API Documentation	https://hotel-booking-system-qjxx.onrender.com/api

E. Environment Variables Reference

```

# Database Configuration
DB_HOST=localhost
DB_PORT=5432
DB_USER=postgres
DB_PASSWORD=password
DB_NAME=hotel_booking

# JWT Configuration
JWT_SECRET=your-secret-key

# Stripe Configuration
STRIPE_SECRET_KEY=sk_test_xxx
STRIPE_WEBHOOK_SECRET=whsec_xxx
STRIPE_SUCCESS_URL=http://localhost:5173/payment/success
STRIPE_CANCEL_URL=http://localhost:5173/payment/cancel

# Google OAuth Configuration
GOOGLE_CLIENT_ID=xxx.apps.googleusercontent.com
GOOGLE_CLIENT_SECRET=GOCSPX-xxx
GOOGLE_CALLBACK_URL=http://localhost:3000/auth/google/redirect

# Cloudinary Configuration
CLOUDINARY_CLOUD_NAME=your-cloud-name
CLOUDINARY_API_KEY=xxx
CLOUDINARY_API_SECRET=xxx

# CORS Configuration
CORS_ORIGIN=http://localhost:5173,http://localhost:5174

# Environment
NODE_ENV=development
PORT=3000

```

F. Diagram Summaries (PDF-Friendly)

These tables provide text-based summaries of the visual diagrams for PDF compatibility.

System Architecture Summary

Layer	Components	Technologies
Presentation	User Frontend, Admin Frontend	Vue.js 3, Pinia, Vue Router, Axios
Application	Backend API	NestJS, Passport, JWT, class-validator
Data	Database, Payment, Storage	PostgreSQL (Neon), Stripe, Cloudinary
External	Authentication	Google OAuth 2.0

Backend Modules

Module	Responsibilities
Auth	Login, Register, OAuth, Profile, Password
Admin	User management, Roles, Statistics
Hotels	CRUD, Search, Filters, Status
Rooms	CRUD, Availability, Images, Beds
Booking	Create, Status, Cancel, Cron jobs
Payment	Stripe Checkout, Webhooks, Verify
Ratings	Create, Update, Delete, Hotel averages
Amenities	CRUD, Categories
Bed Types	CRUD

Entity Relationships

Entity	Relationships
User	Has many Bookings, Payments, Ratings; Has many-to-many Roles
Hotel	Has many Rooms, Ratings; Has many-to-many Amenities
Room	Belongs to Hotel; Has many-to-many Amenities, Bed Types
Booking	Belongs to User; Has many BookingItems, Payment
BookingItem	Belongs to Booking, Room
Payment	Belongs to Booking, User
Rating	Belongs to Hotel, User, Booking

Booking Status Flow

From Status	Action	To Status
(New)	User creates booking	PENDING

From Status	Action	To Status
PENDING	Admin approves	CONFIRMED
PENDING	Admin rejects	CANCELLED
PENDING	User cancels	CANCELLED
CONFIRMED	Payment succeeds	COMPLETED
CONFIRMED	Payment timeout (1hr)	FAILED
CONFIRMED	User cancels	CANCELLED

Deployment Summary

Service	Platform	Type
Backend API	Render	Web Service
User Frontend	Render	Static Site
Admin Frontend	Render	Static Site
Database	Neon	PostgreSQL 16
Images	Cloudinary	CDN
Payments	Stripe	API

End of Final Project Report