Cloud-based hotel booking solution

Project Information

Field	Value
Student	Andrey Patsino
Group	22HR C#
Supervisor	Pavlo Andriiash
Date	2026-01-05

Links

Production: Azure App Services (Swagger)

Service	App Service Name	Default Domain
Users	`hotel-booking-users-api`	`https://hotel-booking-users-api-csbghtd2f9cph7g5.northeurope-01.azurewebsites.net/swagger/index.html`
Hotels	`hotel-booking-hotels-api`	`https://hotel-booking-hotels-api-evhhefafhhbrgrbs.northeurope-01.azurewebsites.net/swagger/index.html`
Reservations	`hotel-booking-reservations-api`	`https://hotel-booking-reservations-api-dwfzh9bydth3fke0.northeurope-01.azurewebsites.net/swagger/index.html`
Payments	`hotel-booking-payments-api`	`https://hotel-booking-payments-api-gch8e7fyeqenfje8.northeurope-01.azurewebsites.net/swagger/index.html`

Repositories

Repository	URL	Description
Infra	`https://github.com/Patsino/hotel-booking-infra`	Docker Compose, .env.example, shared docs

Repository	URL	Description
Users	`https://github.com/Patsino/hotel-booking-users-service`	User management, authentication, JWT
Hotels	`https://github.com/Patsino/hotel-booking-hotels-service`	Hotel/room listings, owner management
Reservations	`https://github.com/Patsino/hotel-booking-reservations-service`	Booking logic, availability
Payments	`https://github.com/Patsino/hotel-booking-payments-service`	Stripe integration, refunds
Frontend	https://github.com/Patsino/hotel-booking-frontend	React UI

Elevator Pitch

The Hotel Booking platform is a cloud-based system designed for travelers searching for hotel accommodations and for hotel owners who want to list and manage their properties. It addresses the complexity of modern hotel booking by providing a secure and reliable way to search hotels, make reservations, process payments, and handle cancellations. The platform is built as a set of independent services that each focus on a specific business area, allowing the system to scale and evolve efficiently. What makes it unique is the clear separation of responsibilities between services, which improves maintainability, reliability, and flexibility while supporting the complete booking lifecycle.

Evaluation Criteria Checklist

#	Criterion	Status	Documentation
1	Back-end	✅	02-technical/criteria/criterion-1-backend.md
2	API Documentation	✅	02-technical/criteria/criterion-2-api-documentation.md
3	Database	✅	02-technical/criteria/criterion-3-database.md
4	Cloud	✅	02-technical/criteria/criterion-4-cloud.md
5	Microservices	✅	02-technical/criteria/criterion-5-microservices.md
6	Containerization	✅	02-technical/criteria/criterion-6-containerization.md
7	Automated tests ≥ 70% coverage	✅	02-technical/criteria/criterion-7-testing.md

Documentation Navigation

- Project Overview - Business context, goals, and requirements
- Technical Implementation - Architecture, tech stack, and criteria details
- User Guide - How to use the application
- Retrospective - Lessons learned and future improvements

Document created: 2026-01-01
Last updated: 2026-01-05

---

1. Project Overview

This section covers the business context, goals, and requirements for the project.

Contents

- Problem Statement & Goals
- Stakeholders & Users
- Scope
- Features

Executive Summary

The Hotel Booking Platform solves a common frustration: **overly complex booking processes** that discourage users before they even complete a reservation. Modern platforms burden users with endless steps, pop-ups, and forms. Our solution provides a **lightweight, minimal-effort booking experience**—search, select, pay, done. Additionally, the platform serves an underserved market: **pet owners** seeking pet-friendly hotels or dedicated pet care facilities when traveling. With clear pet filters and a pet hotel category, finding accommodation for pets is straightforward. The system is built as a microservices architecture with four independent services (Users, Hotels, Reservations, Payments), deployed on Azure, with a React frontend for a clean user experience.

Key Highlights

Aspect	Description
Problem	Modern booking platforms are overly complex, leading to high abandonment. Pet owners lack visibility into pet-friendly options.
Solution	Lightweight booking interface with minimal steps + pet hotel support

Aspect	Description
Target Users	Travelers wanting fast booking, Pet owners, Hotel Owners (including pet hotels), Administrators
Key Features	Fast 3-step booking, Pet-friendly filters, Stripe payments, Smart cancellation policies, GDPR compliance
Tech Stack	.NET 9, ASP.NET Core, React, SQL Server, Azure, Docker, Stripe API

Problem Statement & Goals

Context

Modern hotel booking platforms have become increasingly complex, requiring users to navigate through multiple steps, pop-ups, and forms before completing a simple reservation. This friction in the booking process leads to cart abandonment and customer frustration. Additionally, pet owners face limited options when traveling, as most platforms don't adequately support pet-friendly accommodations or dedicated pet hotels.

Problem Statement

Who: Travelers seeking quick hotel bookings without complex multi-step processes, pet owners needing accommodation for their pets, hotel owners wanting to list properties (including pet hotels), and platform administrators.

What: Modern booking platforms make the reservation process overly complex and multi-step, discouraging potential customers before they even reach payment. Users are forced through endless forms, account verifications, and upsell screens. Pet owners additionally struggle to find reliable pet-friendly hotels or dedicated pet care facilities when traveling.

Why: Complex booking flows lead to high abandonment rates. Users want to search, select, and pay—nothing more. The pet accommodation market remains underserved by mainstream platforms, leaving pet owners with limited visibility into available options.

Pain Points

#	Pain Point	Severity	Current Workaround
1	**Overly complex booking flows** - Too many steps between finding a room and completing payment	High	Users abandon bookings or switch to simpler platforms

#	Pain Point	Severity	Current Workaround
2	**Limited pet accommodation options** - Mainstream platforms don't highlight pet-friendly hotels or dedicated pet hotels	Medium	Pet owners call hotels directly or use fragmented pet-specific services
3	**Unclear cancellation policies** - Users don't know refund terms until deep in the booking process	Medium	Manual research or calling hotels directly
4	**Slow, cluttered interfaces** - Heavy platforms with ads, pop-ups, and unnecessary features	High	Desktop-only usage or frustration

Business Goals

Goal	Description	Success Indicator
Fast Booking Experience	Minimal steps from search to confirmed reservation	User can complete booking in under 2 minutes
Pet-Friendly Focus	Support for pet hotels and pet-friendly accommodations with clear filtering	Pet filter available, dedicated pet hotel category
Demonstrate Microservices Architecture	Build a fully functional system using 4 independent microservices	Four services deployed and communicating successfully
Implement Secure Authentication	Provide JWT-based authentication with role-based access control	Users can register, login, and access appropriate resources based on roles
Integrate Payment Processing	Implement Stripe payment integration with webhook handling	Payments can be created, confirmed, and refunded through Stripe
Cloud Deployment	Deploy working system to Azure cloud	All services accessible via public URLs
Docker for Local Development	Provide Docker setup for local development	docker-compose up starts full system

Objectives & Metrics

Objective	Metric	Current Value	Target Value	Timeline
API Coverage	Automated test coverage	≥70%	≥70%	2025-12-12
Cloud Deployment	Services deployed to Azure	4	4	2025-12-11

Objective	Metric	Current Value	Target Value	Timeline
Documentation	API documentation completeness	Swagger for all endpoints	Swagger for all endpoints	2025-12-12

Success Criteria

Must Have

- [x] Four microservices (Users, Hotels, Reservations, Payments) deployed and functional
- [x] JWT-based authentication with refresh token rotation
- [x] Role-based access control (User, HotelOwner, Admin)
- [x] Stripe payment integration
- [x] ≥70% automated test coverage for all services
- [x] Azure cloud deployment (App Service + Key Vault)
- [x] Swagger API documentation
- [x] Docker Compose for local development
- [x] Database migrations via Entity Framework Core

Nice to Have

- [x] React frontend application for user-friendly booking interface
- [ ] CI/CD pipeline (not implemented - manual deployment via Visual Studio)
- [ ] Message queue for async communication (using synchronous HTTP)

Non-Goals

What this project explicitly does NOT aim to achieve:

- Real-time messaging or notifications (email, SMS, push notifications)
- Mobile application development
- Multi-language/localization support
- Review and rating system for hotels
- Image upload and storage functionality
- Full production-grade monitoring and alerting infrastructure

Stakeholders & Users

Target Audience

Persona	Description	Key Needs
Traveler (User)	Individuals seeking hotel accommodations for business or leisure travel	Easy search and booking, secure payments, flexible cancellation options
Hotel Owner	Property owners or managers wanting to list and manage their hotels	Hotel registration, room management, booking visibility, approval workflow
Platform Administrator	System operators responsible for platform oversight	User management, hotel approval, cancellation request handling, system monitoring

User Personas

Persona 1: Pavlo the Traveler

Attribute	Details
Role	Business Traveler & Pet Owner
Age	20-55
Tech Savviness	Low
Goals	Quickly find and book suitable accommodation without complex multi-step processes, find pet care when traveling
Frustrations	Overly complex booking flows with too many steps, pop-ups and upsells, unclear cancellation policies, difficulty finding pet-friendly options
Scenario	Pavlo needs to book a hotel for a business trip to Klaipėda, but he also has a cat that needs care while he's away. He uses the platform to quickly book his hotel in Klaipėda—searching, selecting a room, and paying in just a few clicks. Then he searches for a pet hotel in Vilnius, filters by "Pet Hotel", finds a suitable facility, and books accommodation for his cat. The entire process takes minutes, not the usual frustrating ordeal of navigating complex booking platforms. Later, his meeting is rescheduled so he needs to cancel—the system automatically processes his refund since he's within the free cancellation window.

Persona 2: Maria the Hotel Owner

Attribute	Details
Role	Small Hotel Owner
Age	35-55
Tech Savviness	Low
Goals	List his hotel on the platform, manage room inventory and pricing, receive bookings and payments
Frustrations	Complicated onboarding processes, lack of control over cancellation policies, difficulty tracking reservations
Scenario	Maria registers as a Hotel Owner, submits his hotel for approval with details like location, amenities, and cancellation policy (free cancellation 7 days before check-in). After admin approval, she adds room listings with capacity, price per night, and accommodation type. She can view incoming reservations and manage his property details.

Persona 3: Dmitry the Administrator

Attribute	Details
Role	Platform Administrator
Age	25-55
Tech Savviness	High
Goals	Ensure platform quality by reviewing hotel submissions, handle cancellation disputes, manage user accounts
Frustrations	Fraudulent hotel listings, complex cancellation disputes requiring manual intervention
Scenario	Dmitry reviews pending hotel submissions daily, approving legitimate listings and rejecting those that don't meet quality standards. When a user requests cancellation outside the free period, he reviews the request and decides whether to approve a refund based on the circumstances provided.

Stakeholder Map

High Influence / High Interest

- **Platform Administrators**: Responsible for overall platform quality, user management, and dispute resolution. Direct stake in system reliability and usability.

High Influence / Low Interest

- **Payment Processor (Stripe)**: Critical external dependency for payment processing. Defines API constraints and security requirements but not involved in day-to-day operations.

Low Influence / High Interest

- **Travelers (Users)**: Primary end users who benefit from the platform. Their satisfaction drives platform success but individual users have limited influence on platform decisions.
- **Hotel Owners**: Key content providers for the platform. Dependent on approval workflow and interested in booking volume.

Low Influence / Low Interest

- **External Service Providers**: Azure infrastructure, database services. Essential for operations but not directly interested in business outcomes.

Project Scope

In Scope ✅

Feature	Description	Priority
User Authentication	Registration, login, JWT tokens, refresh token rotation	Must
Role-Based Access Control	User, HotelOwner, Admin roles with appropriate permissions	Must
Hotel Management	Create, update, search hotels with approval workflow	Must
Room Management	Create, update, delete rooms with pricing and availability	Must
Reservation Management	Create reservations, view bookings, cancellation workflow	Must
Payment Processing	Stripe integration for payment intents, confirmation, refunds	Must
Admin Functions	Hotel approval, cancellation request handling, user management	Must
API Documentation	Swagger/OpenAPI documentation for all endpoints	Must
Automated Testing	Unit and integration tests with ≥70% coverage	Must
Cloud Deployment	Azure App Service deployment with Key Vault secrets	Must

Feature	Description	Priority
Docker Support	Docker Compose for local development environment	Must
GDPR Compliance	Data export requests, user data deletion	Must
Service-to-Service Auth	API key authentication for internal endpoints	Must

Out of Scope ❌

Feature	Reason	When Possible
Mobile Application	Focus on backend microservices architecture	Future phase
Review/Rating System	Not essential for core booking flow demonstration	Future phase
Email/SMS Notifications	Requires additional infrastructure (SendGrid, Twilio)	Future phase
Image Upload/Storage	Would require Azure Blob Storage integration	Future phase
Multi-Language Support	Localization adds complexity without demonstrating core concepts	Future phase
Loyalty/Rewards Program	Business feature beyond core booking functionality	Future phase
Advanced Analytics	Would require additional data warehouse infrastructure	Future phase

Assumptions

#	Assumption	Impact if Wrong	Probability
1	Azure free tier resources sufficient for demonstration	May need to upgrade or limit testing	Low
2	Stripe test mode adequate for payment flow demonstration	Would need production keys and compliance	Low
3	Single SQL Server database with schema separation acceptable	Would need separate database instances	Medium
4	Users have modern web browsers supporting Swagger UI	May need alternative documentation format	Low
5	JWT tokens sufficient for stateless authentication	May need to implement additional session management	Low

Constraints

Limitations that affect the project:

Constraint Type	Description	Mitigation
Time	Diploma project deadline 2026-01-07	Prioritize must-have features, defer nice-to-haves
Budget	Azure student subscription with free tier limits	Use free tier resources, optimize for cost
Technology	Required to use .NET for backend services	Leverage .NET 9 latest features and best practices
Resources	Single developer working on entire system	Focus on clean architecture for maintainability
External	Stripe payment processing dependency	Use test mode, handle webhook failures gracefully
Infrastructure	Azure student subscription allows only one free SQL database	Split single database into schemas per microservice

Dependencies

Dependency	Type	Owner	Status
Stripe API	External	Stripe	✅ Active
Azure App Service	External	Microsoft	✅ Active
Azure SQL Database	External	Microsoft	✅ Active
Azure Key Vault	External	Microsoft	✅ Active
.NET 9 SDK	Technical	Microsoft	✅ Active
Entity Framework Core	Technical	Microsoft	✅ Active
SQL Server (Docker)	Technical	Microsoft	✅ Active

Features & Requirements

Epics Overview

Epic	Description	Stories	Status
E1: User Management	Authentication, authorization, and user profile management	8	✅
E2: Hotel Management	Hotel and room CRUD operations with approval workflow	11	✅
E3: Reservation Management	Booking lifecycle with smart cancellation policies	7	✅
E4: Payment Processing	Stripe integration for payments and refunds	5	✅
E5: Administration	Admin functions for platform oversight	6	✅

User Stories

Epic 1: User Management

Authentication, authorization, and user profile management for the platform.

ID	User Story	Acceptance Criteria	Priority	Status
US-001	As a visitor, I want to register an account, so that I can access the booking platform	- Email must be unique - Password minimum 8 characters - Registered as User role by default	Must	✅
US-002	As a user, I want to login with my credentials, so that I can access my account	- Returns JWT access token (60 min) - Returns refresh token (7 days) - Invalid credentials return 401	Must	✅
US-003	As a user, I want to refresh my access token, so that I can stay logged in	- Refresh token rotates on use - Old refresh token is invalidated - Returns new access and refresh tokens	Must	✅

ID	User Story	Acceptance Criteria	Priority	Status
US-004	As a user, I want to become a hotel owner, so that I can list my properties	- Endpoint: POST /api/auth/become-hotel-owner - Role changes from User to HotelOwner - Returns new tokens with updated role	Must	✅
US-005	As a user, I want to change my password, so that I can maintain account security	- Requires current password verification - New password minimum 8 characters	Should	✅
US-006	As a user, I want to logout, so that my session is terminated securely	- Refresh token is revoked - Access token remains valid until expiry	Should	✅
US-007	As a user, I want to request my data export, so that I comply with GDPR	- Creates export request - Generates downloadable file - File expires after period	Could	✅
US-008	As an admin, I want to view all users, so that I can manage the platform	- Returns paginated user list - Shows role and status - Admin-only access	Must	✅

Epic 2: Hotel Management

Hotel and room CRUD operations with admin approval workflow.

ID	User Story	Acceptance Criteria	Priority	Status
US-009	As a hotel owner, I want to submit my hotel, so that it can be listed on the platform	- Hotel created with Pending status - Includes location, description, cancellation policy - Only HotelOwner role can submit	Must	✅
US-010	As a hotel owner, I want to update my hotel details, so that information stays accurate	- Can only update own hotels - Cannot update approval status - Changes reflected immediately	Should	✅
US-011	As a hotel owner, I want to add rooms to my hotel, so that guests can book them	- Room linked to approved hotel	Must	✅

ID	User Story	Acceptance Criteria	Priority	Status
		- Includes capacity, price, accommodation type - Room visibility can be toggled		
US-012	As a hotel owner, I want to view my hotels, so that I can manage my properties	- Returns only own hotels - Shows approval status - Includes room count	Must	✅
US-037	As a hotel owner, I want to view reservations on my hotel, so that I can track bookings	- Returns all reservations for hotel's rooms - Shows guest count, dates, status - Only hotel owner or admin can access	Must	✅
US-013	As a user, I want to search for hotels, so that I can find suitable accommodation	- Filter by country, city, dates - Filter by pets allowed, price range - Only returns approved hotels with available rooms	Must	✅
US-014	As a user, I want to view hotel details, so that I can make booking decisions	- Shows full hotel information - Includes cancellation policy - Only visible for approved hotels	Must	✅
US-015	As a user, I want to view room details, so that I can choose appropriate accommodation	- Shows capacity, price, amenities - Shows accommodation type - Only visible rooms returned	Must	✅
US-016	As an admin, I want to view pending hotels, so that I can review submissions	- Returns hotels with Pending status - Admin-only access	Must	✅
US-017	As an admin, I want to approve hotels, so that they become visible to users	- Status changes to Approved - ReviewedAt timestamp set - Hotel becomes searchable	Must	✅
US-018	As an admin, I want to reject hotels, so that unsuitable listings are excluded	- Status changes to Rejected - ReviewedAt timestamp set - Hotel not visible in search	Must	✅

Epic 3: Reservation Management

Booking lifecycle management with smart cancellation policies.

ID	User Story	Acceptance Criteria	Priority	Status
US-019	As a user, I want to create a reservation, so that I can book accommodation	- Reservation created with Pending status - Room availability verified - No overlapping dates allowed	Must	✅
US-020	As a user, I want to view my reservations, so that I can track my bookings	- Returns only own reservations - Shows status and dates - Includes cancellation status	Must	✅
US-021	As a user, I want to cancel my reservation, so that I can change my plans	- If within free period: auto-canceled with refund - If outside free period: requires admin approval - Cancellation reason required	Must	✅
US-022	As an admin, I want to view all reservations, so that I can monitor bookings	- Returns paginated list - Can filter by status - Admin-only access	Should	✅
US-023	As an admin, I want to view cancellation requests, so that I can process them	- Returns reservations with Requested status - Shows cancellation reason - Admin-only access	Must	✅
US-024	As an admin, I want to approve cancellation requests, so that users get refunds	- Status changes to AdminApproved - Triggers refund in Payments Service - Reservation marked Canceled	Must	✅
US-025	As an admin, I want to reject cancellation requests, so that policies are enforced	- Status changes to AdminRejected - No refund processed - Reservation remains Confirmed	Must	✅

Epic 4: Payment Processing

Stripe integration for secure payment processing.

ID	User Story	Acceptance Criteria	Priority	Status
US-026	As a user, I want to create a payment for my reservation, so that I can confirm booking	- Creates Stripe PaymentIntent - Returns clientSecret for frontend - Amount must match reservation	Must	✅
US-027	As a user, I want to confirm my payment, so that my reservation is finalized	- Payment status updated via webhook - Reservation status changes to Confirmed - Payment details recorded	Must	✅
US-028	As a user, I want to view my payment status, so that I can track transactions	- Shows payment status and amount - Shows refund amount if applicable - Only own payments visible	Should	✅
US-029	As an admin, I want payments refunded automatically on cancellation approval, so that users receive their money	- Full refund processed via Stripe - Payment status updated to Refunded - RefundedAt timestamp recorded	Must	✅
US-030	As a system, I want to receive Stripe webhooks, so that payment status is synchronized	- Webhook signature validated - Payment status updated based on event - Reservation service notified	Must	✅

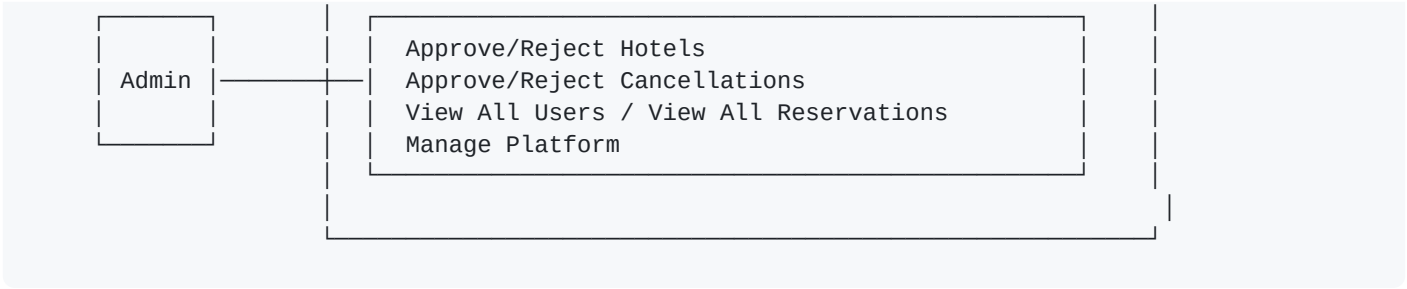
Epic 5: Administration

Platform administration and oversight functions.

ID	User Story	Acceptance Criteria	Priority	Status
US-031	As an admin, I want to seed test data, so that I can demonstrate the platform	- Creates sample users, hotels, rooms - Only runs if database is empty	Should	✅

ID	User Story	Acceptance Criteria	Priority	Status
		- Uses configurable default password		
US-032	As a service, I want to authenticate internal API calls, so that services communicate securely	- API key required in X-API-Key header - Each service has unique key - Keys stored in Key Vault	Must	✅
US-033	As an admin, I want to manually update reservation status, so that I can handle edge cases	- Can set status to any valid value - Audit trail maintained - Admin-only access	Could	✅
US-034	As a service, I want health check endpoints, so that monitoring can verify service status	- Returns 200 OK when healthy - Available without authentication	Should	✅
US-035	As a developer, I want API documentation, so that I can understand endpoints	- Swagger UI available - All endpoints documented - Request/response examples included	Must	✅
US-036	As an admin, I want to soft delete users, so that GDPR deletion is supported	- User marked as deleted - Data anonymized or removed - Cannot login after deletion	Should	✅

Use Case Diagram



Non-Functional Requirements

Performance

Requirement	Target	Measurement Method
API response time	< 500ms for standard operations	Manual testing / Swagger
Concurrent users	Supports multiple concurrent API requests	Azure App Service scaling
Database queries	Optimized with proper indexing	EF Core query analysis

Security

- **Authentication**: JWT tokens with 60-minute expiry, refresh token rotation
- **Authorization**: Role-based access control (User, HotelOwner, Admin)
- **Data Protection**: PBKDF2 password hashing, HTTPS enforcement, Azure Key Vault for secrets
- **Input Validation**: DataAnnotations on all DTOs, EF Core parameterized queries
- **API Security**: Service-to-service API key authentication for internal endpoints

Reliability

Metric	Target
Service availability	99%+ (Azure App Service SLA)
Data consistency	Eventual consistency between services
Error handling	Graceful degradation with proper error responses

Compatibility

Platform/Tool	Minimum Version
.NET Runtime	9.0
SQL Server	2019+ / Azure SQL

Platform/Tool	Minimum Version
Docker	20.10+
Swagger UI	Modern browsers (Chrome, Firefox, Safari, Edge)

---

2. Technical Implementation

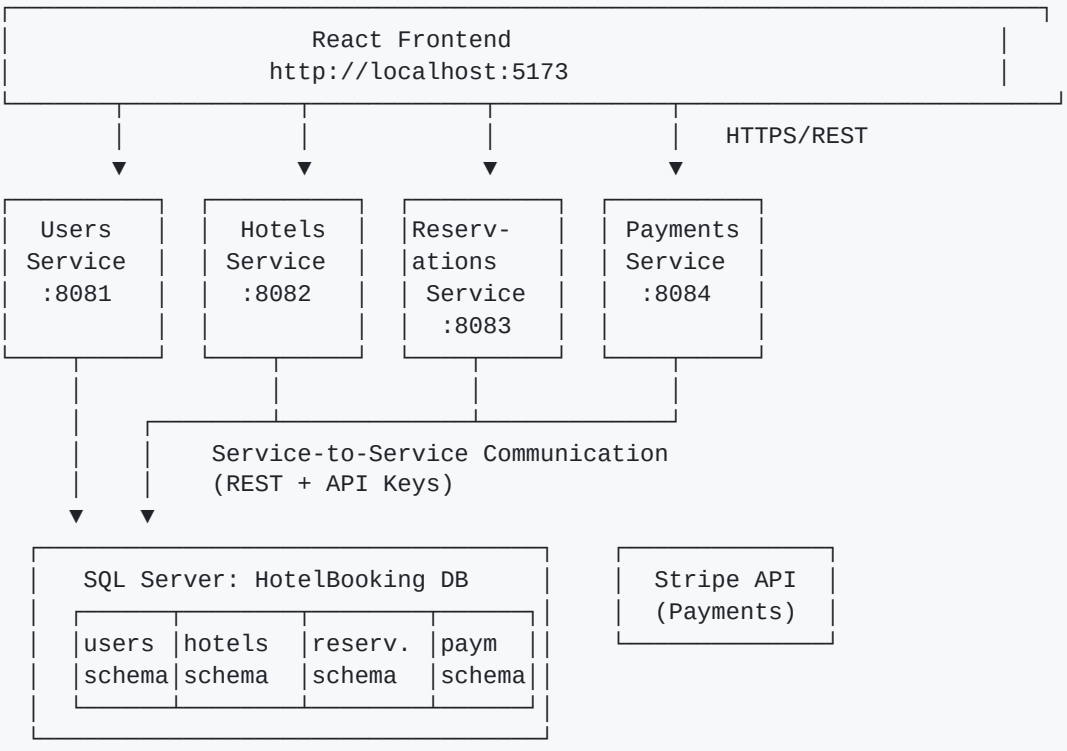
This section covers the technical architecture, design decisions, and implementation details.

Contents

- Tech Stack
- Criteria Documentation - ADR for each evaluation criterion
- Deployment

Solution Architecture

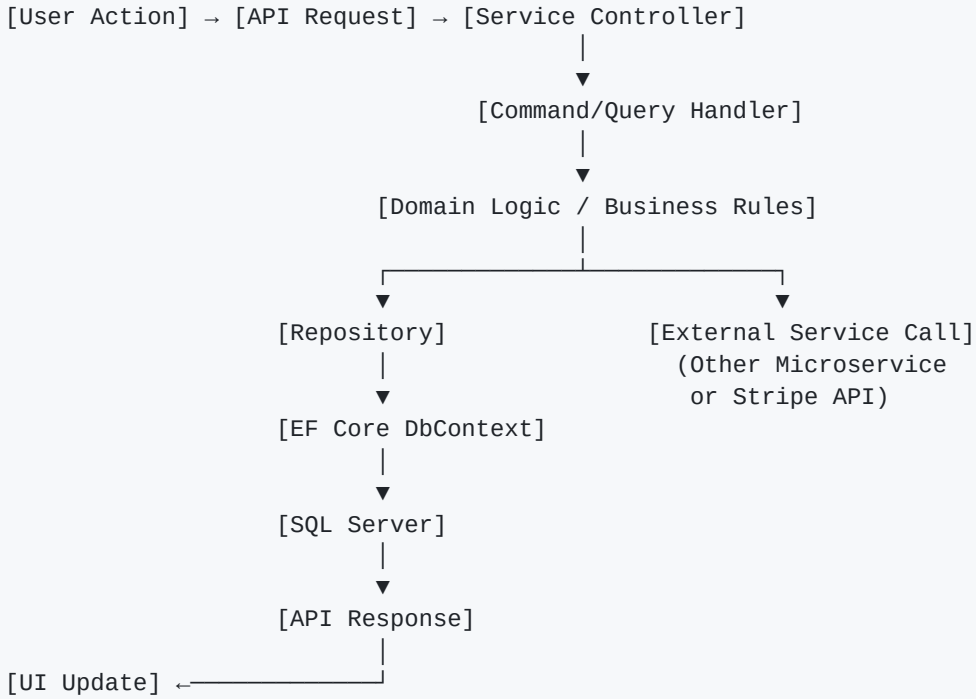
High-Level Architecture



System Components

Component	Description	Technology
Frontend	React SPA for user-friendly booking interface	React, TypeScript, Vite
Users Service	Authentication, user management, GDPR compliance	ASP.NET Core, JWT, PBKDF2
Hotels Service	Hotel/room management, search, approval workflow	ASP.NET Core, EF Core
Reservations Service	Booking lifecycle, cancellation policies	ASP.NET Core, EF Core
Payments Service	Stripe integration, payment processing, webhooks	ASP.NET Core, Stripe.NET
Database	Shared SQL Server with schema separation	SQL Server / Azure SQL
Secret Management	Centralized secrets and configuration	Azure Key Vault

Data Flow



Key Technical Decisions

Decision	Rationale	Alternatives Considered
Microservices Architecture	Independent deployment, scaling, and development of each bounded context	Monolithic (rejected: harder to scale and maintain)
Shared Database with Schemas	Azure free tier limitation allows only one database; schemas provide logical separation	Separate databases (rejected: not feasible in free tier)
JWT Authentication	Stateless, scalable authentication suitable for microservices	Session-based (rejected: requires shared state)
API Key for Service-to-Service	Simple, secure internal authentication	OAuth2 (rejected: added complexity for internal calls)
Stripe for Payments	Industry standard, comprehensive API, test mode support	Manual payment handling (rejected: security concerns)

Security Overview

Aspect	Implementation
Authentication	JWT tokens (60 min expiry) with refresh token rotation (7 days)
Authorization	Role-based (User, HotelOwner, Admin) with policy-based checks
Data Protection	PBKDF2 password hashing, TLS/HTTPS enforcement, Azure Key Vault for secrets
Secrets Management	Azure Key Vault with Managed Identity access, environment variables in Docker
API Security	Service-to-service API keys, webhook signature validation (Stripe)

Technology Stack

Stack Overview

Layer	Technology	Version	Justification
Frontend	React	18.x	Component-based UI with fast development cycle
Frontend Build	Vite	5.x	Fast build tool with HMR for React

Layer	Technology	Version	Justification
Frontend Language	TypeScript	5.x	Type safety for frontend development
Runtime	.NET	9.0	Latest LTS with performance improvements, native AOT support
Framework	ASP.NET Core	9.0	Industry-standard for building RESTful APIs in .NET
ORM	Entity Framework Core	9.0	First-class .NET integration, migrations, LINQ support
Database	SQL Server / Azure SQL	2019+	Robust relational database with Azure integration
Authentication	JWT Bearer Tokens	-	Stateless authentication suitable for microservices
Payment Processing	Stripe	-	Industry-standard payment processor with comprehensive API
Containerization	Docker	20.10+	Consistent development and deployment environments
Cloud Platform	Azure	-	App Service, SQL Database, Key Vault integration
API Documentation	Swagger/OpenAPI	3.0	Interactive API documentation and testing

Frontend

Repository: https://github.com/Patsino/hotel-booking-frontend

The React frontend provides a lightweight, user-friendly booking interface designed for minimal-effort reservations.

Key Libraries:

- React 18 with functional components and hooks
- TypeScript for type safety
- React Router for navigation
- Axios for API communication
- Stripe Elements for payment UI

Key Technology Decisions

Decision 1: .NET 9 with ASP.NET Core

Context: Need a robust, performant backend framework for building RESTful microservices.

Decision: Use .NET 9 with ASP.NET Core for all four microservices.

Rationale:

- Latest .NET version with performance improvements
- Strong typing and compile-time safety
- Excellent tooling support (Visual Studio, VS Code)
- Built-in dependency injection
- Rich ecosystem for authentication, ORM, and testing

Trade-offs:

- Pros: Performance, type safety, comprehensive framework features, strong Azure integration
- Cons: Larger runtime footprint compared to minimal frameworks

Decision 2: Entity Framework Core with Code-First Migrations

Context: Need an ORM for database operations with support for migrations and multiple database providers.

Decision: Use Entity Framework Core with code-first approach.

Rationale:

- Seamless integration with ASP.NET Core
- Type-safe queries with LINQ
- Automatic migration generation
- Support for both SQL Server and In-Memory provider for testing

Trade-offs:

- Pros: Developer productivity, type safety, migration support, testability
- Cons: Performance overhead compared to raw SQL for complex queries

Decision 3: JWT with Refresh Token Rotation

Context: Need stateless authentication that works across multiple microservices.

Decision: Implement JWT access tokens (60 min) with refresh token rotation (7 days).

Rationale:

- Stateless authentication eliminates shared session state
- Short-lived access tokens limit exposure if compromised
- Refresh token rotation provides security with convenience
- Standard claims (sub, role, email) for authorization decisions

Trade-offs:

- Pros: Scalability, no shared state, industry standard
- Cons: Token revocation requires additional infrastructure (refresh token blacklist)

Decision 4: Shared Database with Schema Separation

Context: Azure free tier allows only one SQL database, but microservices need logical data separation.

Decision: Use a single SQL Server database with separate schemas per service.

Rationale:

- Complies with Azure free tier limitations
- Logical separation via schemas (users, hotels, reservations, payments)
- Each service owns its schema exclusively
- No cross-schema foreign keys (maintained in application layer)

Trade-offs:

- Pros: Cost-effective, still maintains logical boundaries
- Cons: Not true database-per-service isolation

Development Tools

Tool	Purpose	Notes
IDE	Visual Studio 2022 / VS Code	C# Dev Kit extension for VS Code
Version Control	Git	GitHub repositories per microservice
Package Manager	NuGet	Standard .NET package management
Testing	xUnit, FluentAssertions, Moq	>70% coverage target
API Testing	Swagger UI, .http files	Built-in request testing
Coverage	Coverlet + ReportGenerator	HTML coverage reports
Containerization	Docker, Docker Compose	Local development orchestration

External Services & APIs

Service	Purpose	Pricing Model
Stripe	Payment processing (intents, confirmations, refunds, webhooks)	Pay per transaction (using test mode)
Azure App Service	Web application hosting	Free F1 tier (student subscription)
Azure SQL Database	Managed SQL Server database	Free tier - General Purpose Serverless
Azure Key Vault	Secrets and configuration management	Standard tier

NuGet Packages

Core Packages (All Services)

Package	Purpose
`Microsoft.AspNetCore.Authentication.JwtBearer`	JWT authentication
`Microsoft.EntityFrameworkCore.SqlServer`	SQL Server database provider
`Swashbuckle.AspNetCore`	Swagger/OpenAPI documentation
`Azure.Identity`	Azure Key Vault authentication
`Azure.Extensions.AspNetCore.Configuration.Secrets`	Key Vault configuration provider

Testing Packages

Package	Purpose
`xunit`	Testing framework
`FluentAssertions`	Expressive assertions
`Moq`	Mocking framework
`Microsoft.AspNetCore.Mvc.Testing`	Integration testing
`Microsoft.EntityFrameworkCore.InMemory`	In-memory database for tests
`coverlet.collector`	Code coverage collection

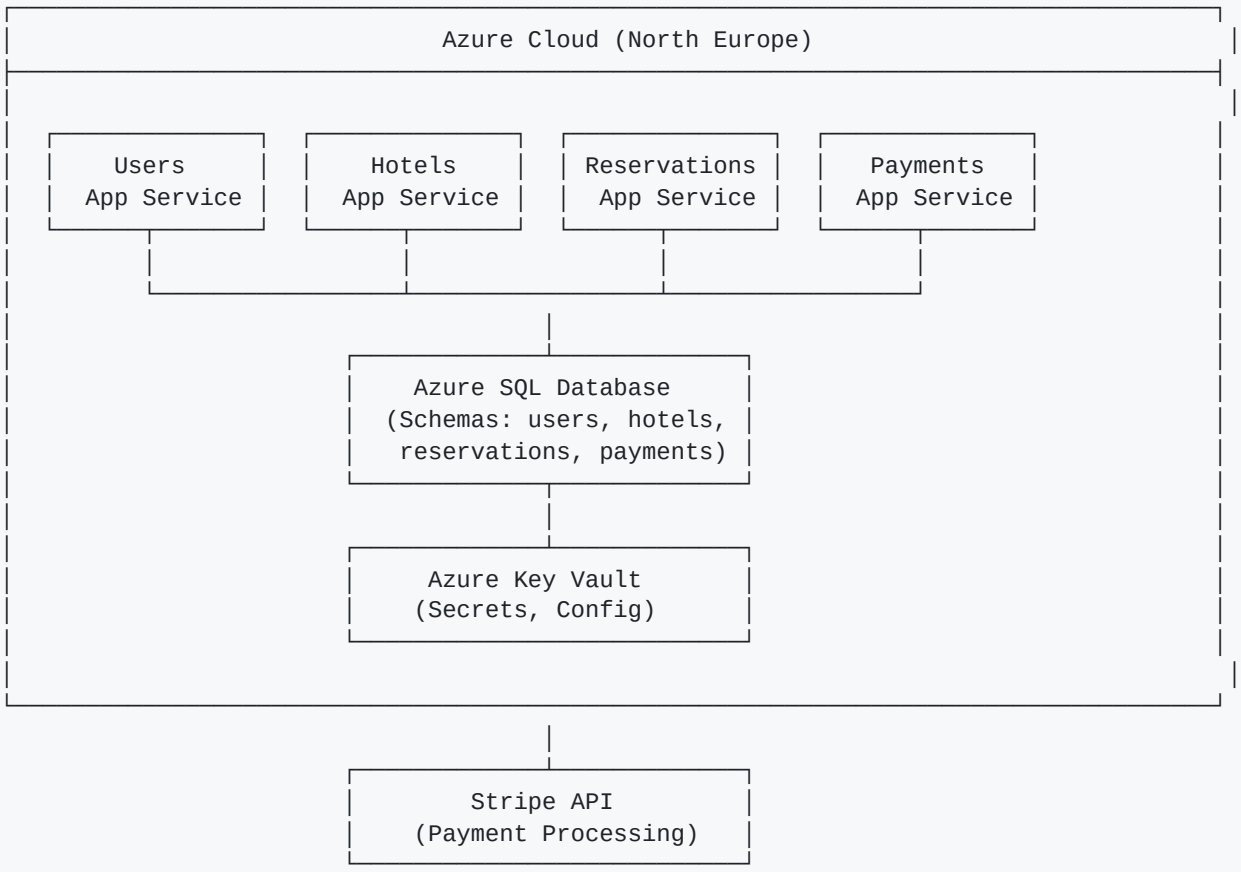
Service-Specific Packages

Service	Package	Purpose
Payments	`Stripe.net`	Stripe API client
Users	`Microsoft.AspNetCore.Cryptography.KeyDerivation`	PBKDF2 password hashing

Deployment & DevOps

Infrastructure

Deployment Architecture



Environments

Environment	URL	Purpose
Development	`localhost:8081-8084`	Local Docker Compose development
Production	Azure App Services	Live deployment with Azure infrastructure

Production URLs

Service	Swagger URL
Users	`https://hotel-booking-users-api-csbghtd2f9cph7g5.northeurope-01.azurewebsites.net/swagger/index.html`
Hotels	`https://hotel-booking-hotels-api-evhhefafhhbrgrbs.northeurope-01.azurewebsites.net/swagger/index.html`
Reservations	`https://hotel-booking-reservations-api-dwfzh9bydth3fke0.northeurope-01.azurewebsites.net/swagger/index.html`
Payments	`https://hotel-booking-payments-api-gch8e7fyeqenfje8.northeurope-01.azurewebsites.net/swagger/index.html`

Environment Variables

Variable	Description	Required	Example
`ASPNETCORE_ENVIRONMENT`	Runtime environment	Yes	`Development` / `Production`
`AZURE_KEYVAULT_RESOURCEENDPOINT`	Key Vault URL	Yes (Production)	`https://kv-hotel-booking-2.vault.azure.net/`
`ConnectionStrings__DefaultConnection`	Database connection	Yes	`Server=...;Database=...`
`Jwt__SecretKey`	JWT signing key	Yes	`***` (Key Vault)
`Jwt__Issuer`	JWT issuer claim	Yes	`HotelBooking`
`Jwt__Audience`	JWT audience claim	Yes	`HotelBookingUsers`
`Jwt__ExpirationMinutes`	Access token TTL	Yes	`60`

Variable	Description	Required	Example
`Jwt__RefreshTokenExpirationDays`	Refresh token TTL	Yes	`7`
`ApiKeys__Services__*`	Service API keys	Yes	`***` (Key Vault)
`ServiceUrls__*`	Inter-service URLs	Yes	`https://...azurewebsites.net`
`Stripe__SecretKey`	Stripe API key	Yes (Payments)	`sk_test_***`
`Stripe__WebhookSecret`	Webhook signature key	Yes (Payments)	`whsec_***`

Secrets Management: Azure Key Vault with Managed Identity access. Local development uses environment variables or `appsettings.Development.json`.

How to Run Locally

Prerequisites

- .NET 9 SDK
- Docker with Docker Compose
- Stripe CLI (for webhook testing)
- SQL Server or Docker

Setup Steps

```
# 1. Clone the infrastructure repo first (contains docker-compose.yml)
git clone https://github.com/Patsino/hotel-booking-infra
cd hotel-booking-infra

# 2. Clone all service repositories into the same folder
git clone https://github.com/Patsino/hotel-booking-users-service
git clone https://github.com/Patsino/hotel-booking-hotels-service
git clone https://github.com/Patsino/hotel-booking-reservations-service
git clone https://github.com/Patsino/hotel-booking-payments-service

# 3. Create .env file from template
cp .env.example .env
# Edit .env with your values (see environment variables above)

# 4. Generate secrets (PowerShell with Git Bash for openssl)
openssl rand -base64 64  # JWT Secret
openssl rand -base64 32  # Encryption Key
openssl rand -base64 16  # Encryption IV
```

```
openssl rand -base64 32  # API Keys (generate 4)

# 5. Start all services with Docker Compose
docker-compose --env-file .env.docker up -d --build

# 6. Verify services are running
docker-compose ps
```

Docker Compose Services

```
services:
  sqlserver:     # SQL Server database (port 1433)
  users-service: # Users API (port 8081)
  hotels-service: # Hotels API (port 8082)
  reservations-service: # Reservations API (port 8083)
  payments-service: # Payments API (port 8084)
```

Frontend Setup

The React frontend provides a user-friendly interface for the booking platform.

Repository: https://github.com/Patsino/hotel-booking-frontend

```
# 1. Clone the frontend repository
git clone https://github.com/Patsino/hotel-booking-frontend
cd hotel-booking-frontend

# 2. Install dependencies
npm install

# 3. Configure API endpoints (optional)
# Edit .env file for local development: use localhost URLs
# Or use cloud: use Azure production URLs (default)

# 4. Start the development server
npm run dev

# 5. Open in browser
# http://localhost:5173/
```

Verify Installation

After starting the services:

1. Open Swagger UI:
    - Users: http://localhost:8081/swagger
    - Hotels: http://localhost:8082/swagger
    - Reservations: http://localhost:8083/swagger
    - Payments: http://localhost:8084/swagger

2. Check health endpoints:
    - http://localhost:8081/health
    - http://localhost:8082/health
    - http://localhost:8083/health
    - http://localhost:8084/health

Stripe Webhook Testing

```
# Forward Stripe webhooks to local payments service
stripe listen --forward-to localhost:8084/api/webhooks/stripe

# Copy webhook signing secret to .env
```

Azure Deployment Process

Via Visual Studio

1. Right-click `Api` project → **Publish**
2. Select existing Azure App Service profile
3. Click **Publish**
4. Azure App Service automatically:
    - Stops the app
    - Deploys new files
    - Starts the app
    - Runs database migrations on startup
    - Loads configuration from Key Vault

Database Migrations

Migrations run automatically on application startup:

1. App starts
2. `Program.cs` executes migration logic
3. Connects to Azure SQL using Key Vault connection string
4. Runs `dbContext.Database.MigrateAsync()` (idempotent)
5. Seeds initial data if database is empty

Azure Resource Configuration

Resource Group

- **Name**: `hotel_booking_solution2`
- **Location**: `North Europe`

Costs (Student Subscription)

Resource	Tier	Cost
App Service Plan	Free F1	Free
Azure SQL Database	Free tier, General Purpose Serverless	Free
Key Vault	Standard	Free tier usage

Security Configuration

Feature	Implementation
Managed Identity	System-assigned identity for Key Vault access
HTTPS Enforcement	All traffic redirected to HTTPS
Key Vault Access	Access policies per Web App identity
Connection Strings	Stored in Key Vault, referenced via `@Microsoft.KeyVault()`

Criterion 1: Back-end

Architecture Decision Record

Status

Status: Accepted

Date: 2025-12-12

Context

The project requires a robust back-end architecture capable of handling multiple business domains (users, hotels, reservations, payments) with proper separation of concerns, scalability, and maintainability. The back-end must support RESTful APIs, secure authentication, and integration with external services (Stripe).

Decision

Implement a microservices architecture using .NET 9 with ASP.NET Core, following Clean Architecture principles within each service. Each microservice follows a layered structure:

- **Api Layer**: Controllers, middleware, filters
- **Application Layer**: Commands, queries, handlers, DTOs
- **Domain Layer**: Entities, value objects, domain logic
- **Infrastructure Layer**: Repositories, external services, database context

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
Monolithic Architecture	Simpler deployment, easier debugging	Harder to scale, tight coupling, difficult to maintain	Does not demonstrate microservices patterns

Consequences

Positive:

- Clear separation of concerns between services
- Independent deployment and scaling of each service
- Type-safe code with compile-time error checking
- Rich .NET ecosystem for authentication, ORM, testing

Negative:

- Increased complexity in service communication
- Need for distributed tracing and logging
- More infrastructure to manage

Neutral:

- Learning curve for Clean Architecture patterns

Implementation Details

Project Structure

Each microservice follows this structure:

```
hotel-booking-{service}-service/
├── Api/
```

```
│   ├── Controllers/           # REST API endpoints
│   ├── Filters/               # Exception filters
│   ├── Middleware/            # Request pipeline middleware
│   └── Program.cs             # Application entry point
├── Application/
│   ├── Commands/              # Write operations
│   ├── Dtos/                  # Data transfer objects
│   ├── Handlers/              # Command/query handlers
│   └── Services/              # Application services
├── Domain/
│   ├── {Entity}/              # Domain entities
│   │   ├── {Entity}.cs        # Entity definition
│   │   └── I{Entity}Repository.cs  # Repository interface
│   └── Enums/                 # Domain enums
├── Infrastructure/
│   ├── Authentication/        # JWT/API key auth
│   ├── Authorization/         # Policy handlers
│   ├── Http/                  # HTTP clients
│   ├── Migrations/            # EF Core migrations
│   ├── Persistence/           # DbContext, repositories
│   └── DependencyInjection.cs # Service registration
└── Tests/
    └── Unit/Integration tests
```

Key Implementation Decisions

Decision	Rationale
Clean Architecture	Separation of concerns, testability, dependency inversion
Repository pattern	Abstraction over data access via interfaces (IUsersRepository, etc.)
Handler pattern	Each use case implemented as separate handler class (RegisterUserHandler, etc.)
Dependency Injection	Built-in .NET DI container for loose coupling
Async database operations	All EF Core calls use async methods (ToListAsync, SaveChangesAsync)

Service Overview

Service	Port	Responsibility
Users	8081	Authentication, user management, GDPR
Hotels	8082	Hotel/room management, search, approval workflow
Reservations	8083	Booking lifecycle, cancellation policies
Payments	8084	Stripe integration, payment processing

Code Examples

Controller Example (Users Service):

```
[ApiController]
[Route("api/auth")]
public sealed class AuthController : ControllerBase
{
    private readonly IRegisterUserHandler _registerHandler;
    private readonly ILoginHandler _loginHandler;

    [HttpPost("register")]
    [ProducesResponseType(StatusCodes.Status201Created)]
    public async Task<IActionResult> Register([FromBody] RegisterUserCommand command)
    {
        var result = await _registerHandler.HandleAsync(command);
        return CreatedAtAction(nameof(Register), new { id = result.UserId }, result);
    }
}
```

Handler Example:

```
public class RegisterUserHandler : IRegisterUserHandler
{
    private readonly IUsersRepository _repository;
    private readonly IPasswordHasher _passwordHasher;

    public async Task<RegisterUserResult> HandleAsync(RegisterUserCommand command)
    {
        // Validate email uniqueness
        var existingUser = await _repository.GetByEmailAsync(command.Email);
        if (existingUser != null)
            throw new InvalidOperationException("Email already registered");

        // Create user with hashed password
        var user = User.Create(
            command.Email,
            _passwordHasher.HashPassword(command.Password),
            command.FirstName,
            command.LastName,
            command.Role);

        await _repository.AddAsync(user);
        return new RegisterUserResult(user.Id, user.Email, user.Role);
    }
}
```

Requirements Checklist

Minimum Requirements (Grade 5)

#	Requirement	Status	Evidence
1	Modern framework (ASP.NET Core, Spring, Django, etc.)	✅	.NET 9 with ASP.NET Core
2	Database for state management	✅	SQL Server via Entity Framework Core
3	ORM usage	✅	Entity Framework Core with code-first
4	Layered architecture	✅	Api/Application/Domain/Infrastructure layers
5	SOLID principles	✅	DI, single responsibility handlers, interface segregation
6	API documentation	✅	Swagger/OpenAPI with XML comments
7	Global error handling	✅	ExceptionHandlingMiddleware in each service
8	Logging	✅	ILogger with structured logging
9	Production deployment	✅	Azure App Service (North Europe)
10	Test coverage ≥70%	✅	75-85% coverage per service

Maximum Requirements (Grade 10)

#	Requirement	Status	Notes
1	CI/CD pipeline	❌	Not implemented - manual deployment via Visual Studio. CI/CD not selected as criterion
2	Microservices architecture	✅	Four independent services

Known Limitations

Limitation	Impact	Potential Solution
No message queue for async communication	Services rely on synchronous HTTP calls	Implement RabbitMQ or Azure Service Bus
No distributed tracing	Debugging cross-service issues harder	Add correlation ID logging (partially implemented)
No circuit breaker pattern	Service failures can cascade	Implement Polly retry/circuit breaker policies

References

- ASP.NET Core Documentation
- Clean Architecture by Robert C. Martin
- Controllers: `Api/Controllers/` in each service
- Handlers: `Application/Handlers/` in each service

Criterion 2: API Documentation

Architecture Decision Record

Status

Status: Accepted

Date: 2025-12-12

Context

The project requires comprehensive API documentation that allows developers and reviewers to understand, test, and integrate with the microservices. Documentation must be interactive, always up-to-date with the code, and provide clear examples of request/response formats.

Decision

Implement Swagger/OpenAPI 3.0 documentation using Swashbuckle.AspNetCore, with:

- XML documentation comments on all controllers and methods
- Swagger annotations for operation metadata
- Example schemas for request/response bodies
- Authentication support in Swagger UI
- Separate Swagger instance per microservice

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
Manual documentation (Markdown)	Full control over format	Quickly becomes outdated, no interactive testing	Not synchronized with code

Alternative	Pros	Cons	Why Not Chosen
Postman Collections	Good for testing, shareable	Separate from code, manual maintenance	Not integrated with deployment

Consequences

Positive:

- Auto-generated from code annotations - always up-to-date
- Interactive testing directly in browser
- Standard OpenAPI format exportable to other tools
- Clear endpoint documentation with examples

Negative:

- Requires maintaining XML comments and annotations
- Swagger UI adds to application size

Neutral:

- Developers must follow documentation conventions

Implementation Details

Swagger Configuration

Each service configures Swagger in `Program.cs`:

```
builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "Hotel Booking - Users Service API",
        Version = "v1",
        Description = "Authentication and user management microservice"
    });

    // JWT Authentication in Swagger
    options.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        Type = SecuritySchemeType.Http,
        Scheme = "bearer",
        BearerFormat = "JWT",
        Description = "Enter JWT token"
    });

    // Include XML comments
    var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
```

```
    options.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory, xmlFile));
});
```

Documentation Standards

Controller Documentation:

```
/// <summary>
/// Create Stripe payment intent for a reservation
/// </summary>
/// <param name="command">Payment details including reservation ID, amount, and currency</param>
/// <returns>Payment intent with client secret for frontend confirmation</returns>
/// <remarks>
/// Creates a Stripe payment intent. Returns client secret for confirming payment on frontend.
///
/// Sample request:
///
///     POST /api/payments/create-intent
///     {
///        "reservationId": 150,
///        "amount": 178.00,
///        "currency": "EUR"
///     }
///
/// **Response includes:**
/// - **paymentIntentId**: Stripe payment intent ID
/// - **clientSecret**: Use with Stripe.js to confirm payment
/// - **paymentId**: Internal payment record ID
/// </remarks>
/// <response code="200">Payment intent created successfully</response>
/// <response code="400">Invalid reservation or amount</response>
/// <response code="401">User not authenticated</response>
/// <response code="403">User trying to pay for another user's reservation</response>
[HttpPost("create-intent")]
[SwaggerOperation(Summary = "Create payment intent", Description = "Create Stripe payment intent
[SwaggerResponse(200, "Payment intent created")]
[SwaggerResponse(400, "Invalid request")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> CreatePaymentIntent([FromBody] CreatePaymentIntentCommand comma
```

API Endpoints by Service

Users Service (Port 8081)

Endpoint	Method	Description
`/api/auth/register`	POST	Register new user
`/api/auth/login`	POST	Login and get tokens
`/api/auth/refresh`	POST	Refresh access token

Endpoint	Method	Description
`/api/auth/logout`	POST	Revoke refresh token
`/api/users`	GET	Get all users (Admin)
`/api/users/{id}`	GET	Get user by ID
`/api/users/{id}`	PATCH	Update user profile
`/api/users/{id}/password`	PATCH	Change password
`/api/gdpr/export`	POST	Request data export

Hotels Service (Port 8082)

Endpoint	Method	Description
`/api/hotels/search`	GET	Search hotels with filters
`/api/hotels/{id}`	GET	Get hotel details
`/api/hotels`	POST	Submit new hotel
`/api/hotels/{id}`	PATCH	Update hotel
`/api/hotels/mine`	GET	Get own hotels
`/api/rooms/{id}`	GET	Get room details
`/api/rooms`	POST	Add room to hotel
`/api/admin/hotels/pending`	GET	Get pending hotels
`/api/admin/hotels/{id}/approve`	POST	Approve hotel
`/api/admin/hotels/{id}/reject`	POST	Reject hotel

Reservations Service (Port 8083)

Endpoint	Method	Description
`/api/reservations`	POST	Create reservation
`/api/reservations/{id}`	GET	Get reservation details
`/api/reservations/mine`	GET	Get own reservations
`/api/reservations/{id}/cancel`	POST	Request cancellation

Endpoint	Method	Description
`/api/admin/reservations`	GET	Get all reservations
`/api/admin/reservations/cancellations`	GET	Get pending cancellations
`/api/admin/reservations/{id}/approve-cancel`	POST	Approve cancellation
`/api/admin/reservations/{id}/reject-cancel`	POST	Reject cancellation

Payments Service (Port 8084)

Endpoint	Method	Description
`/api/payments/create-intent`	POST	Create payment intent
`/api/payments/confirm`	POST	Confirm payment
`/api/payments/{id}`	GET	Get payment details
`/api/payments/reservation/{id}`	GET	Get payment by reservation
`/api/webhooks/stripe`	POST	Stripe webhook handler

Swagger UI Access

Service	Swagger URL
Users	https://hotel-booking-users-api-csbghtd2f9cph7g5.northeurope-01.azurewebsites.net/swagger/index.html
Hotels	https://hotel-booking-hotels-api-evhhefafhhbrgrbs.northeurope-01.azurewebsites.net/swagger/index.html
Reservations	https://hotel-booking-reservations-api-dwfzh9bydth3fke0.northeurope-01.azurewebsites.net/swagger/index.html
Payments	https://hotel-booking-payments-api-gch8e7fyeqenfje8.northeurope-01.azurewebsites.net/swagger/index.html

Requirements Checklist

Minimum Requirements (Grade 5)

#	Requirement	Status	Evidence
1	Structured API documentation	✅	Swagger UI with organized endpoints
2	OpenAPI/Swagger specification	✅	Swashbuckle generates valid OpenAPI 3.0
3	Endpoint examples (requests/responses)	✅	XML comments with sample JSON
4	HTTP status codes documented	✅	ProducesResponseType on all methods
5	Data model descriptions	✅	DTO classes with XML documentation
6	Getting started guide	✅	docs/03-user-guide/index.md
7	Published in accessible format	✅	Swagger UI on Azure URLs
8	Documentation strategy described	✅	This document + code annotations

Maximum Requirements (Grade 10)

#	Requirement	Status	Notes
1	Comprehensive docs (guides, tutorials, versioning)	⚠️ Partial	Getting started exists, no versioning docs
2	The API documentation must thoroughly describe advanced topics	⚠️ Partial	Request flows, auth documented
3	Detailed diagrams (sequence, component)	✅	Architecture diagrams in reference/
4	API linting/validation	❌	No Spectral or schema validation

Known Limitations

Limitation	Impact	Potential Solution
No API versioning	Breaking changes affect all clients	Implement URL or header-based versioning
No rate limiting documentation	Clients unaware of limits	Document rate limits when implemented
Internal endpoints visible	Could confuse external developers	Separate internal/external Swagger docs

References

- Swashbuckle.AspNetCore
- OpenAPI Specification
- Swagger configuration: `Program.cs` in each service
- Controller documentation: `Api/Controllers/` in each service

Criterion 3: Database

Architecture Decision Record

Status

Status: Accepted

Date: 2025-12-04

Context

The project requires persistent data storage for users, hotels, rooms, reservations, and payments. The database solution must support:

- Relational data with proper constraints
- Code-first migrations for schema evolution
- Efficient querying for search operations
- Support for both local development and cloud deployment

Decision

Use SQL Server with Entity Framework Core, implementing:

- Single database with schema separation per microservice
- Code-first migrations with automatic application on startup
- Repository pattern for data access abstraction
- EF Core In-Memory provider for testing

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
Separate databases per service	True isolation, independent scaling	Azure free tier allows only one database	Cost constraint

Alternative	Pros	Cons	Why Not Chosen
PostgreSQL	Open source, rich features	Less integrated with Azure ecosystem	SQL Server has better Azure integration
Dapper	Performance, control	Manual mapping, no migrations	EF Core productivity benefits outweigh

Consequences

Positive:

- Familiar SQL Server ecosystem
- Automatic migrations simplify deployment
- Strong Azure SQL integration
- EF Core provides type-safe queries

Negative:

- Schema separation is not true database isolation
- Potential for accidental cross-schema access
- Single database is potential bottleneck

Neutral:

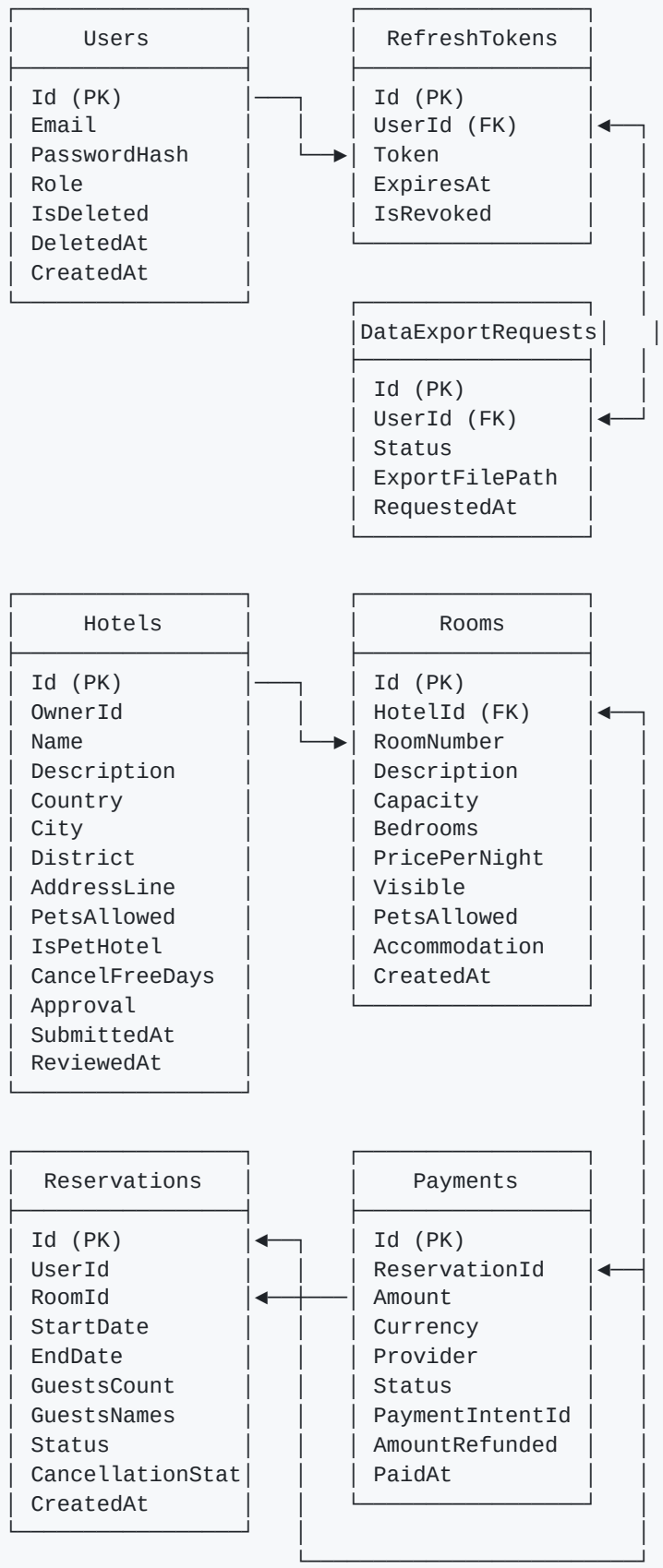
- Need to coordinate schema changes across services

Implementation Details

Database Architecture

```
┌─────────────────────────────────────────────────────────────────────┐
│                        HotelBooking (SQL Database)                  │
├─────────────────────────────────────────────────────────────────────┤
│  Schema: users                                                      │
│  ├── Users                                                          │
│  ├── RefreshTokens                                                  │
│  └── DataExportRequests                                             │
├─────────────────────────────────────────────────────────────────────┤
│  Schema: hotels                                                     │
│  ├── Hotels                                                         │
│  └── Rooms                                                          │
├─────────────────────────────────────────────────────────────────────┤
│  Schema: reservations                                               │
│  └── Reservations                                                   │
├─────────────────────────────────────────────────────────────────────┤
│  Schema: payments                                                   │
│  └── Payments                                                       │
└─────────────────────────────────────────────────────────────────────┘
```

Entity Relationship Diagram



Note: Cross-schema references (UserId, RoomId, ReservationId)
are logical - no physical foreign keys across schemas

DbContext Configuration

Users Service DbContext:

```
public class UsersDbContext : DbContext
{
    public DbSet<User> Users => Set<User>();
    public DbSet<RefreshToken> RefreshTokens => Set<RefreshToken>();
    public DbSet<DataExportRequest> DataExportRequests => Set<DataExportRequest>();

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasDefaultSchema("users");

        modelBuilder.Entity<User>(entity =>
        {
            entity.HasKey(e => e.Id);
            entity.HasIndex(e => e.Email).IsUnique();
            entity.Property(e => e.Email).HasMaxLength(256).IsRequired();
            entity.Property(e => e.PasswordHash).IsRequired();
            entity.Property(e => e.Role).HasConversion<string>();
        });

        modelBuilder.Entity<RefreshToken>(entity =>
        {
            entity.HasKey(e => e.Id);
            entity.HasIndex(e => e.Token).IsUnique();
            entity.HasOne<User>().WithMany().HasForeignKey(e => e.UserId);
        });
    }
}
```

Migration Strategy

Migrations are applied automatically on application startup:

```
// Program.cs
if (app.Environment.EnvironmentName != "Testing")
{
    using var scope = app.Services.CreateScope();
    var dbContext = scope.ServiceProvider.GetRequiredService<UsersDbContext>();
    await dbContext.Database.MigrateAsync();

    // Seed initial data if needed
    var seeder = scope.ServiceProvider.GetRequiredService<IDataSeeder>();
    await seeder.SeedAsync();
}
```

Key Tables

Users Schema

Table	Purpose	Key Fields
Users	User accounts	Id, Email, PasswordHash, Role, IsDeleted

Table	Purpose	Key Fields
RefreshTokens	JWT refresh tokens	Token, UserId, ExpiresAt, IsRevoked
DataExportRequests	GDPR export tracking	UserId, Status, ExportFilePath

Hotels Schema

Table	Purpose	Key Fields
Hotels	Hotel listings	Id, OwnerId, Name, Country, City, Approval
Rooms	Room inventory	Id, HotelId, RoomNumber, Capacity, PricePerNight

Reservations Schema

Table	Purpose	Key Fields
Reservations	Bookings	Id, UserId, RoomId, StartDate, EndDate, Status

Payments Schema

Table	Purpose	Key Fields
Payments	Payment records	Id, ReservationId, Amount, Status, PaymentIntentId

Requirements Checklist

Minimum Requirements (Grade 5)

#	Requirement	Status	Evidence
1	Data dictionary (tables, columns, types)	✅	docs/appendices/db-schema.md
2	Data integrity and transactions	✅	EF Core transactions, constraints
3	ER diagram or logical schema	✅	Diagrams in docs/appendices/db-schema.md
4	DDL via migrations	✅	EF Core code-first migrations
5	Modern RDBMS	✅	SQL Server / Azure SQL
6	3NF normalization	✅	Normalized schema design

#	Requirement	Status	Evidence
7	Primary/foreign keys, constraints	✅	PK/FK in model configuration
8	Migrations in version control	✅	Infrastructure/Migrations/ in Git
9	Test data seeding	✅	DataSeeder classes, scripts
10	Database roles	✅ Implemented	Each microservice has dedicated database role with minimal required permissions
11	Encrypted passwords	✅	BCrypt hashing (not stored in plain text)

Maximum Requirements (Grade 10)

#	Requirement	Status	Notes
1	Multiple DBMS types if justified	❌	Single SQL Server - sufficient for domain
2	Data layers (raw/staging/mart)	❌	Not applicable - OLTP system
3	Schema versioning documented	✅	EF migrations track changes
4	Indexes documented and justified	⚠️	Indexes exist but not fully documented
5	Triggers/stored procedures	❌	Business logic in application layer
6	Views for complex queries	❌	Not needed for current use cases
7	PII masking	❌	Not implemented

Known Limitations

Limitation	Impact	Potential Solution
Shared database	Not true microservices isolation	Separate databases in production (paid tier)
No cross-schema FKs	Referential integrity in code only	Accept trade-off for microservices pattern
String enum storage	Larger storage, slower queries	Use integer enum conversion

References

- EF Core Documentation
- Azure SQL Database

- DbContext: `Infrastructure/Persistence/*DbContext.cs` in each service
- Migrations: `Infrastructure/Migrations/` in each service

Criterion 4: Cloud Deployment

Architecture Decision Record

Status

Status: Accepted

Date: 2025-12-11

Context

The project requires cloud deployment to demonstrate production-readiness. The deployment must support:

- All four microservices running independently
- Secure secret management
- Database hosting
- HTTPS enforcement
- Cost-effective solution (student budget)

Decision

Deploy to Microsoft Azure using:

- **Azure App Service** (Free F1 tier) for all four microservices
- **Azure SQL Database** (Free tier, General Purpose Serverless) for data storage
- **Azure Key Vault** (Standard tier) for centralized secrets management
- **System-assigned Managed Identity** for secure Key Vault access

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
AWS	Mature platform, wide adoption	Less integrated with .NET, more complex setup	Azure has better .NET integration

Consequences

Positive:

- Excellent .NET and Visual Studio integration
- Managed Identity eliminates credential management
- Free tier sufficient for diploma demonstration
- Built-in SSL certificates
- Easy deployment via Visual Studio

Negative:

- Vendor lock-in to Azure
- Free tier has limitations (cold starts, resource limits)
- Single region deployment

Neutral:

- Azure portal learning curve

Implementation Details

Azure Resource Group

```
Resource Group: hotel_booking_solution2
Location: North Europe

Resources:
├── App Services
│   ├── hotel-booking-users-api
│   ├── hotel-booking-hotels-api
│   ├── hotel-booking-reservations-api
│   └── hotel-booking-payments-api
├── Azure SQL Server
│   └── sql-hotel-booking (database)
└── Key Vault
    └── kv-hotel-booking-2
```

App Service Configuration

Each App Service is configured with:

Setting	Value
Runtime	.NET 9
OS	Linux

Setting	Value
Plan	Free F1
HTTPS Only	Enabled
Managed Identity	System-assigned

Key Vault Secrets

All secrets stored in Azure Key Vault `kv-hotel-booking-2`:

Secret	Purpose
`Jwt-SecretKey`	JWT signing key
`Jwt-Issuer`	JWT issuer claim
`Jwt-Audience`	JWT audience claim
`Jwt-ExpirationMinutes`	Token expiration
`Encryption-Key`	AES encryption key
`Encryption-IV`	AES initialization vector
`ApiKeys-Services-UsersService`	Users service API key
`ApiKeys-Services-HotelsService`	Hotels service API key
`ApiKeys-Services-ReservationsService`	Reservations service API key
`ApiKeys-Services-PaymentsService`	Payments service API key
`SqlConnectionString-Users`	Database connection
`ServiceUrls-*`	Inter-service URLs

Environment Variable Configuration

App Service environment variables reference Key Vault secrets:

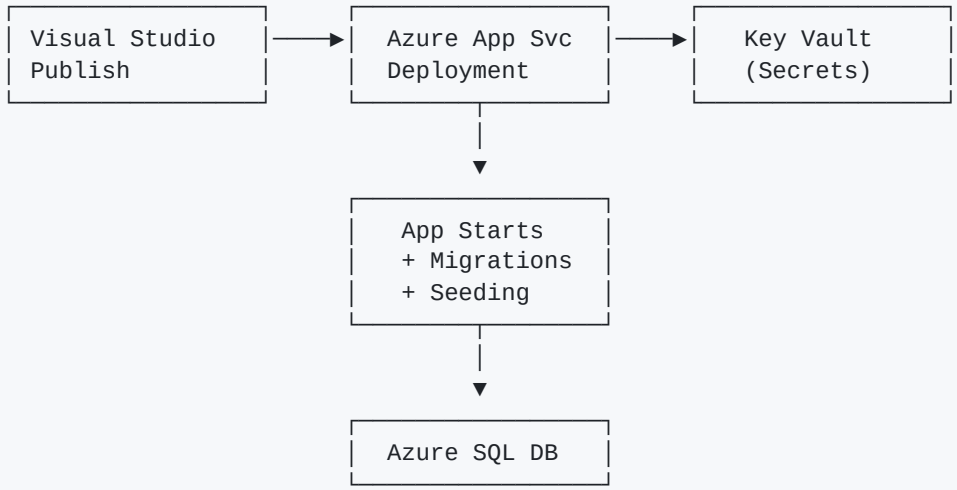
```
ASPNETCORE_ENVIRONMENT=Production
AZURE_KEYVAULT_RESOURCEENDPOINT=https://kv-hotel-booking-2.vault.azure.net/

# Key Vault References
Jwt:SecretKey=@Microsoft.KeyVault(SecretUri=https://kv-hotel-booking-2.vault.azure.net/secrets/J
```

```
Jwt:Issuer=@Microsoft.KeyVault(SecretUri=https://kv-hotel-booking-2.vault.azure.net/secrets/Jwt-
ApiKeys:Services:UsersService=@Microsoft.KeyVault(SecretUri=https://kv-hotel-booking-2.vault.azu
```

Production URLs

Service	URL
Users	https://hotel-booking-users-api-csbghtd2f9cph7g5.northeurope-01.azurewebsites.net
Hotels	https://hotel-booking-hotels-api-evhhefafhhbrgrbs.northeurope-01.azurewebsites.net
Reservations	https://hotel-booking-reservations-api-dwfzh9bydth3fke0.northeurope-01.azurewebsites.net
Payments	https://hotel-booking-payments-api-gch8e7fyeqenfje8.northeurope-01.azurewebsites.net

Deployment Process



1. Right-click project → Publish
2. Select Azure App Service profile
3. Click Publish
4. Azure automatically:
   - Stops application
   - Deploys new files
   - Restarts application
   - Runs migrations on startup
   - Loads secrets from Key Vault

Security Configuration

Feature	Implementation
Managed Identity	System-assigned identity for each App Service
Key Vault Access	Access policies grant identity secret read permissions
HTTPS	Enforced on all App Services
TLS	TLS 1.2 minimum
Connection Strings	Encrypted in Key Vault

Cost Analysis

Resource	Tier	Monthly Cost
App Service Plan (x4)	Free F1	$0
Azure SQL Database	Free tier	$0
Key Vault	Standard	$0 (free tier usage)
Total		**$0** (student subscription)

Requirements Checklist

Minimum Requirements (Grade 5)

#	Requirement	Status	Evidence
1	Deploy to cloud (AWS/Azure/GCP/etc.)	✅	Azure App Service deployment
2	At least one component accessible via public URL	✅	Four services with public Swagger URLs
3	Clear deployment documentation	✅	docs/02-technical/deployment.md
4	At least one managed cloud service	✅	Azure SQL Database + Azure Key Vault
5	Secrets stored securely (not in repo)	✅	Azure Key Vault with Managed Identity
6	Git repository with commit history	✅	GitHub repositories for all services
7	Application consistently reachable	✅	Services available (cold start delays expected)
8	No unnecessary paid resources running	✅	Using free tier only

Maximum Requirements (Grade 10)

#	Requirement	Status	Notes
1	Containers/orchestration + CI/CD	⚠️	Docker for local dev; CI/CD not selected as criterion
2	At least 2 managed cloud services	✅	App Service + SQL Database + Key Vault
3	Infrastructure as Code	❌	Manual Azure Portal setup
4	Monitoring/logging + auto-scaling	❌	Basic health checks only; no auto-scaling on free tier
5	IAM roles and least privilege	✅	Managed Identity with minimal permissions
6	Screenshots with redacted secrets	✅	Can be provided for defense
7	Reproducible environment scripts	⚠️	Guide on enviroment reproducibility provided

Known Limitations

Limitation	Impact	Potential Solution
Free tier cold starts	First request slow after idle	Upgrade to paid tier or implement keep-alive
Single region	No geographic redundancy	Multi-region deployment in production
Shared App Service Plan	Limited resources	Dedicated plans for production
No custom domain	Using azurewebsites.net URLs	Configure custom domain in production

References

- Azure App Service Documentation
- Azure Key Vault
- Azure SQL Database
- Deployment profiles: `.pubxml` files in each service

Criterion 5: Microservices Architecture

Architecture Decision Record

Status

Status: Accepted

Date: 2025-12-12

Context

The project demonstrates microservices architecture patterns for a complex domain (hotel booking) that naturally decomposes into distinct bounded contexts. The architecture must enable:

- Independent development and deployment of services
- Clear domain boundaries
- Secure inter-service communication
- Fault isolation

Decision

Implement four microservices following Domain-Driven Design bounded contexts:

1. **Users Service** - Identity & Access Management
2. **Hotels Service** - Hotel & Room Inventory
3. **Reservations Service** - Booking Management
4. **Payments Service** - Payment Processing

Each service:

- Owns its data (separate database schema)
- Exposes REST APIs for external clients
- Exposes internal APIs for service-to-service communication
- Uses API key authentication for internal calls

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
Monolithic	Simpler, easier debugging	Hard to scale, tight coupling	Does not meet criteria

Consequences

Positive:

- Clear separation of business domains
- Independent scaling potential
- Technology flexibility per service (all .NET, but could differ)

- Fault isolation between services
- Demonstrates real-world architecture patterns

Negative:

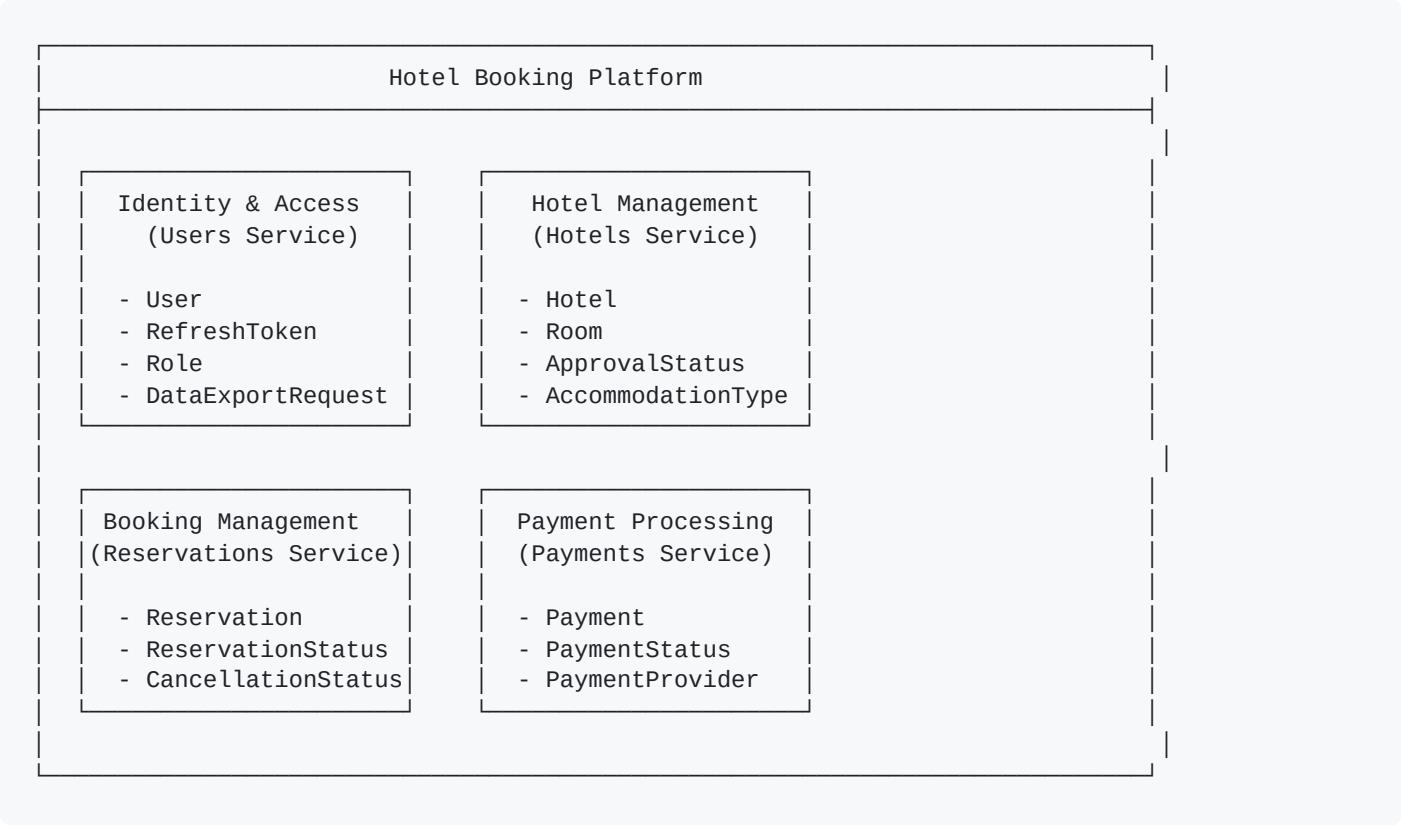
- Distributed system complexity
- Network latency between services
- Data consistency challenges
- More deployment targets to manage

Neutral:

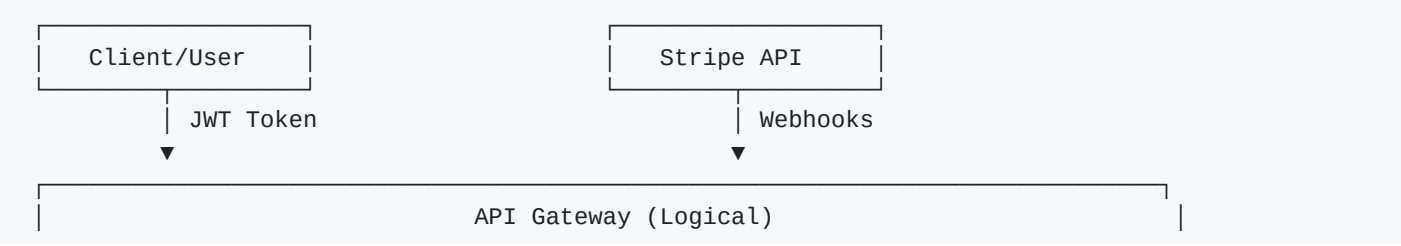
- Each service needs its own testing and deployment pipeline

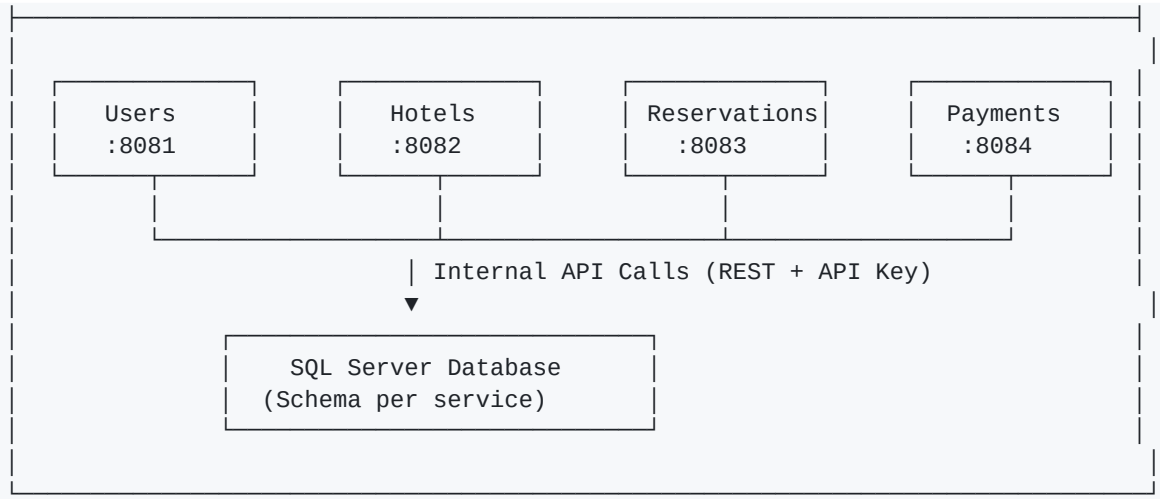
Implementation Details

Bounded Contexts



Service Communication





Internal API Endpoints

Each service exposes `/internal/*` endpoints for service-to-service communication:

Users Service:

```
GET  /api/internal-users/{id}      - Get user details
POST /api/internal-users/validate  - Validate user exists
```

Hotels Service:

```
GET  /api/internal/hotels/{id}              - Get hotel details
GET  /api/internal/rooms/{id}               - Get room details
GET  /api/internal/rooms/{id}/availability  - Check availability
POST /api/internal/rooms/validate           - Validate room bookable
```

Reservations Service:

```
PATCH /api/internal/reservations/{id}/confirm  - Confirm reservation
GET   /api/internal/reservations/{id}          - Get reservation
POST  /api/internal/reservations/validate      - Validate reservation
```

Payments Service:

```
POST /api/internal/payments/refund  - Process refund
GET  /api/internal/payments/{id}    - Get payment details
```

API Key Authentication

Internal endpoints use API key authentication:

```
public class ApiKeyAuthenticationHandler : AuthenticationHandler<ApiKeyAuthenticationOptions>
{
    protected override Task<AuthenticateResult> HandleAuthenticateAsync()
    {
        if (!Request.Headers.TryGetValue("X-API-Key", out var apiKey))
            return Task.FromResult(AuthenticateResult.NoResult());

        var validApiKey = _configuration[$"ApiKeys:Services:{_serviceName}"];

        if (apiKey != validApiKey)
            return Task.FromResult(AuthenticateResult.Fail("Invalid API key"));

        var claims = new[] { new Claim(ClaimTypes.Name, "InternalService") };
        var identity = new ClaimsIdentity(claims, Scheme.Name);
        var principal = new ClaimsPrincipal(identity);
        var ticket = new AuthenticationTicket(principal, Scheme.Name);

        return Task.FromResult(AuthenticateResult.Success(ticket));
    }
}
```

Service Dependencies

Service	Depends On	Communication
Users	None	-
Hotels	Users (validates owner)	HTTP + API Key
Reservations	Hotels (room availability)	HTTP + API Key
Payments	Reservations (confirm booking)	HTTP + API Key

Data Consistency

Strategy: Eventual Consistency

The system uses eventual consistency between services:

1. **Payment → Reservation Confirmation:**
    - User creates reservation (Pending)
    - User creates payment intent
    - Stripe processes payment
    - Webhook triggers reservation confirmation
    - Reservation becomes Confirmed
2. **Cancellation → Refund:**
    - User requests cancellation

- Admin approves (if needed)
- Reservations Service calls Payments Service
- Stripe processes refund
- Both records updated

Compensating Transactions:

- Failed payment: Reservation stays Pending, can retry
- Failed refund: Payment marked for manual review

Requirements Checklist

Minimum Requirements (Grade 5)

#	Requirement	Status	Evidence
1	Architecture diagram (services, protocols)	✅	Diagrams in this document and reference/
2	Domain model with bounded contexts	✅	Users, Hotels, Reservations, Payments
3	API documentation (contracts)	✅	Swagger/OpenAPI for all services
4	Deployment diagram	✅	docs/02-technical/deployment.md
5	Decomposition justification	✅	Natural domain boundaries
6	Minimum 3 business services	✅	Four services (not counting infra)
7	Loose coupling	✅	Each service independent
8	No shared database (own DB/schema)	✅	Separate schemas per service
9	Synchronous communication (REST/gRPC)	✅	REST with API keys
10	Timeouts and retry policies	✅	Polly retry (3 attempts, exponential backoff) + 5s timeout policy per request
11	Health/ready endpoints	✅	/health endpoint on each service
12	Correlation ID logging	✅	CorrelationIdMiddleware generates/propagates X-Correlation-ID header
13	API Gateway or entry point	✅	JWT auth acts as logical gateway - single auth mechanism, services validate tokens

#	Requirement	Status	Evidence
14	Unit tests per service	✅	Tests project for each service
15	Integration tests	✅	Tests.Integration projects

Maximum Requirements (Grade 10) - Optional Directions

Direction	Status	Notes
Async communication (message broker)	❌	HTTP only - not required for minimum
Resilience patterns (circuit breaker)	❌	No Polly or resilience patterns
Observability (distributed tracing)	❌	No Jaeger/Zipkin
Advanced patterns (Saga, CQRS)	❌	Simple handler pattern
API Management (Kong, rate limiting)	❌	Direct service access

Note: Maximum requirements are optional enhancements. Minimum requirements fully satisfied.

Known Limitations

Limitation	Impact	Potential Solution
No message queue	Synchronous coupling	Add RabbitMQ/Azure Service Bus
No circuit breaker	Cascade failures possible	Implement Polly policies
No service mesh	Limited observability	Add Istio or similar
Configuration-based discovery	Manual URL updates	Implement service registry

References

- Microservices by Martin Fowler
- Domain-Driven Design
- Internal controllers: `Api/Controllers/Internal*.cs` in each service
- HTTP clients: `Infrastructure/Http/` in each service

Criterion 6: Containerization

Architecture Decision Record

Status

Status: Accepted

Date: 2025-12-12

Context

The project requires containerization to ensure consistent development environments across team members and simplify local development setup. Containers must:

- Enable running all four microservices locally
- Include database setup
- Handle service dependencies
- Support environment configuration

Decision

Implement Docker containerization with Docker Compose for orchestration:

- Individual Dockerfile per microservice
- Docker Compose for local development orchestration
- SQL Server container for database
- Environment variables via `.env` file

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
Kubernetes	Production-ready orchestration	Overkill for local dev	Too complex for scope
Podman	Rootless, daemonless	Less tooling support	Docker more widely used
No containers	Simpler setup	Inconsistent environments	Required by criteria
Docker Swarm	Built-in to Docker	Less features than K8s	Compose sufficient

Consequences

Positive:

- Consistent development environment
- Easy onboarding for new developers
- Database included in stack
- All services start with single command

Negative:

- Docker overhead on development machines
- Need to manage container resources
- Learning curve for Docker concepts

Neutral:

- Requires Docker Desktop installation

Implementation Details

Dockerfile Structure

Each microservice has a multi-stage Dockerfile:

```
# hotel-booking-users-service/Dockerfile

# Build stage
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src

# Copy csproj files and restore
COPY ["Api/Api.csproj", "Api/"]
COPY ["Application/Application.csproj", "Application/"]
COPY ["Domain/Domain.csproj", "Domain/"]
COPY ["Infrastructure/Infrastructure.csproj", "Infrastructure/"]
RUN dotnet restore "Api/Api.csproj"

# Copy source and build
COPY . .
RUN dotnet build "Api/Api.csproj" -c Release -o /app/build

# Publish stage
FROM build AS publish
RUN dotnet publish "Api/Api.csproj" -c Release -o /app/publish

# Runtime stage
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS final
WORKDIR /app
COPY --from=publish /app/publish .
EXPOSE 8080
ENTRYPOINT ["dotnet", "Api.dll"]
```

Docker Compose Configuration

```yaml
# docker-compose.yml

version: '3.8'

services:
  sqlserver:
    image: mcr.microsoft.com/mssql/server:2022-latest
    container_name: hotel-booking-sqlserver
    environment:
      - ACCEPT_EULA=Y
      - MSSQL_SA_PASSWORD=${SQL_PASSWORD}
    ports:
      - "1433:1433"
    volumes:
      - sqlserver-data:/var/opt/mssql
    healthcheck:
      test: /opt/mssql-tools18/bin/sqlcmd -S localhost -U sa -P ${SQL_PASSWORD} -Q "SELECT 1" -C
      interval: 10s
      timeout: 5s
      retries: 5

  users-service:
    build:
      context: ./hotel-booking-users-service
      dockerfile: Dockerfile
    container_name: hotel-booking-users
    ports:
      - "8081:8080"
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ConnectionStrings__DefaultConnection=Server=sqlserver;Database=HotelBooking;User Id=sa;P
      - Jwt__SecretKey=${JWT_SECRET}
      - Jwt__Issuer=${JWT_ISSUER}
      - Jwt__Audience=${JWT_AUDIENCE}
      - Jwt__ExpirationMinutes=60
      - Jwt__RefreshTokenExpirationDays=7
      - ApiKeys__Services__UsersService=${API_KEY_USERS}
      - ApiKeys__Services__HotelsService=${API_KEY_HOTELS}
      - ApiKeys__Services__ReservationsService=${API_KEY_RESERVATIONS}
      - ApiKeys__Services__PaymentsService=${API_KEY_PAYMENTS}
      - ServiceUrls__Users=http://users-service:8080
      - ServiceUrls__Hotels=http://hotels-service:8080
      - ServiceUrls__Reservations=http://reservations-service:8080
      - ServiceUrls__Payments=http://payments-service:8080
    depends_on:
      sqlserver:
        condition: service_healthy
    networks:
      - hotel-booking-network

  hotels-service:
    build:
      context: ./hotel-booking-hotels-service
      dockerfile: Dockerfile
    container_name: hotel-booking-hotels
    ports:
      - "8082:8080"
    environment:
      # Similar environment variables...
    depends_on:
```

```
      - sqlserver
      - users-service
    networks:
      - hotel-booking-network

  reservations-service:
    build:
      context: ./hotel-booking-reservations-service
      dockerfile: Dockerfile
    container_name: hotel-booking-reservations
    ports:
      - "8083:8080"
    environment:
      # Similar environment variables...
    depends_on:
      - sqlserver
      - hotels-service
    networks:
      - hotel-booking-network

  payments-service:
    build:
      context: ./hotel-booking-payments-service
      dockerfile: Dockerfile
    container_name: hotel-booking-payments
    ports:
      - "8084:8080"
    environment:
      - Stripe__SecretKey=${STRIPE_SECRET_KEY}
      - Stripe__WebhookSecret=${STRIPE_WEBHOOK_SECRET}
      # Other environment variables...
    depends_on:
      - sqlserver
      - reservations-service
    networks:
      - hotel-booking-network

networks:
  hotel-booking-network:
    driver: bridge

volumes:
  sqlserver-data:
```

Environment File

```
# .env.example

# SQL Server
SQL_PASSWORD=YourStrongPassword123!

# JWT Configuration
JWT_SECRET=your-64-byte-secret-key-here-base64-encoded
JWT_ISSUER=HotelBooking
JWT_AUDIENCE=HotelBookingUsers

# Encryption
ENCRYPTION_KEY=your-32-byte-key-base64
ENCRYPTION_IV=your-16-byte-iv-base64
```

```
# API Keys (generate with: openssl rand -base64 32)
API_KEY_USERS=generated-api-key-1
API_KEY_HOTELS=generated-api-key-2
API_KEY_RESERVATIONS=generated-api-key-3
API_KEY_PAYMENTS=generated-api-key-4

# Stripe (get from Stripe Dashboard)
STRIPE_SECRET_KEY=sk_test_...
STRIPE_WEBHOOK_SECRET=whsec_...

# Seeding
SEEDING_DEFAULT_PASSWORD=Password123!
```

Docker Commands

```
# Build and start all services
docker-compose --env-file .env.docker up -d --build

# View running containers
docker-compose ps

# View logs
docker-compose logs -f

# View specific service logs
docker-compose logs -f users-service

# Stop all services
docker-compose down

# Stop and remove volumes (delete database)
docker-compose down -v

# Rebuild specific service
docker-compose up --build users-service

# Access SQL Server container
docker exec -it hotel-booking-sqlserver /opt/mssql-tools18/bin/sqlcmd -S localhost -U sa -P Your
```

Service Ports

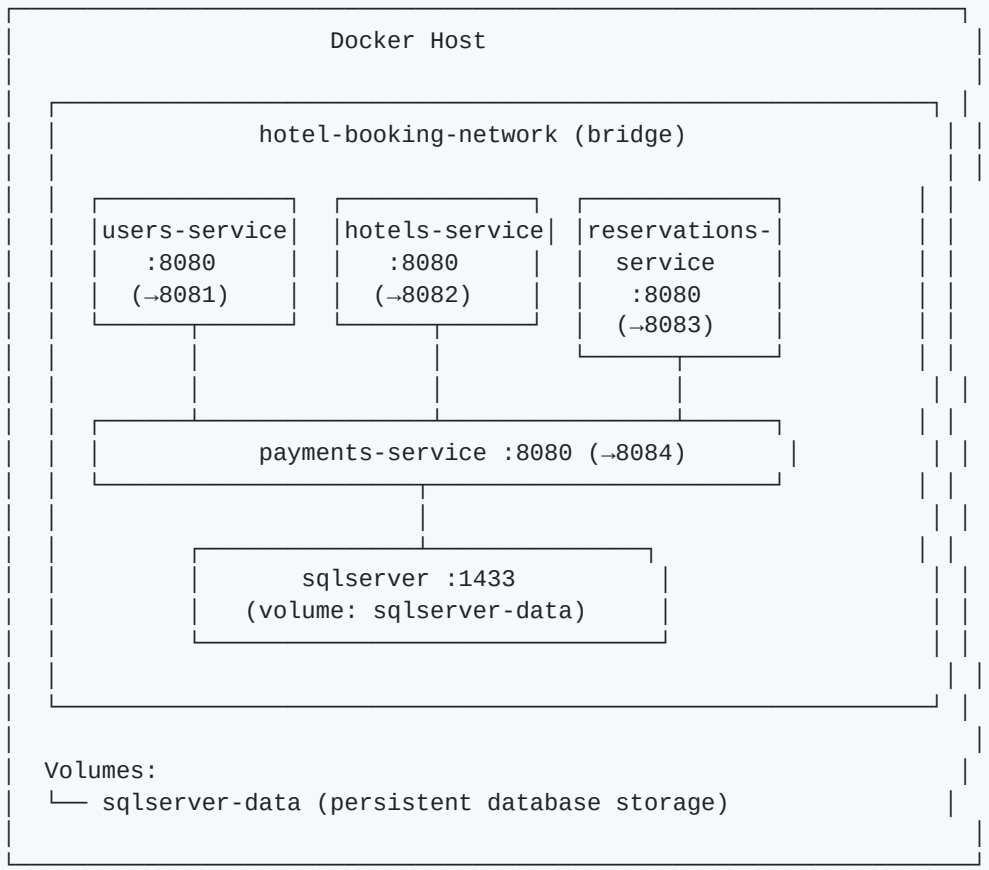
Service	Container Port	Host Port
SQL Server	1433	1433
Users Service	8080	8081
Hotels Service	8080	8082
Reservations Service	8080	8083
Payments Service	8080	8084

Health Checks

All services expose health endpoints:

```
http://localhost:8081/health  # Users
http://localhost:8082/health  # Hotels
http://localhost:8083/health  # Reservations
http://localhost:8084/health  # Payments
```

Container Architecture



Requirements Checklist

Minimum Requirements (Grade 5)

#	Requirement	Status	Evidence
1	Dockerfile per service	✅	Multi-stage Dockerfiles
2	Proper layer ordering	✅	Dependencies first, source last
3	.dockerignore files	✅	Excludes bin/, obj/, .git

#	Requirement	Status	Evidence
4	ENV variables for config	✅	All settings via environment
5	Volumes for persistent data	✅	sqlserver-data volume
6	Port configuration (EXPOSE)	✅	EXPOSE 8080 in Dockerfiles
7	Reasonable image size	✅	~200MB per service (aspnet runtime)
8	No secrets in images	✅	Secrets via .env file
9	Non-root user	✅	Custom appuser created in Dockerfile
10	docker-compose.yml	✅	Full orchestration in hotel-booking-infra repo
11	All services start with single command	✅	docker-compose up
12	Service dependencies (depends_on)	✅	With health checks
13	Isolated networks	✅	hotel-booking-network
14	.env.example provided	✅	Template with variable names
15	README with instructions	✅	Each service has README

Maximum Requirements (Grade 10)

#	Requirement	Status	Notes
1	Custom base image	❌	Using standard mcr.microsoft.com images
2	Conditional caching	⚠️ Partial	Standard Docker layer caching
3	Distroless/Alpine images	✅	Using aspnet:9.0-alpine base image
4	Healthcheck in Dockerfile	✅	HEALTHCHECK directive with wget to /health
5	Resource limits	❌	No memory/CPU limits set
6	Graceful shutdown	✅	.NET handles SIGTERM
7	Separate compose files (dev/prod)	❌	Single compose file

Known Limitations

Limitation	Impact	Potential Solution
No container registry	Built locally only	Push to Docker Hub or ACR

Limitation	Impact	Potential Solution
No resource limits	Can consume all host resources	Add resource constraints
No log aggregation	Logs per container only	Add ELK stack or similar
Single SQL Server instance	Not production-like	Separate database containers

References

- Docker Documentation
- Docker Compose
- SQL Server in Docker
- Dockerfiles: `Dockerfile` in each service root
- Docker Compose: `docker-compose.yml` in workspace root

Criterion 7: Automated Tests (≥70% Coverage)

Architecture Decision Record

Status

Status: Accepted

Date: 2025-12-18

Context

The project requires comprehensive automated testing with a minimum of 70% code coverage for both line and branch coverage. Tests must:

- Verify business logic correctness
- Enable safe refactoring
- Cover all critical paths

Decision

Implement a multi-layered testing strategy:

- **Unit Tests**: Test individual components in isolation with mocked dependencies

- **Integration Tests**: Test component interactions using WebApplicationFactory

Testing tools:

- **xUnit**: Test framework
- **FluentAssertions**: Expressive assertions
- **Moq**: Mocking framework
- **Coverlet**: Code coverage collection
- **ReportGenerator**: HTML coverage reports

Alternatives Considered

Alternative	Pros	Cons	Why Not Chosen
NUnit	Mature, feature-rich	Less ASP.NET Core integration	xUnit preferred for .NET Core
MSTest	Microsoft official	Less community adoption	xUnit more popular
SpecFlow (BDD)	Business-readable	Overhead for small project	Not needed for scope

Consequences

Positive:

- High confidence in code correctness
- Safe refactoring enabled
- Documentation through tests
- Quick feedback loop

Negative:

- Test maintenance overhead
- Some edge cases hard to test
- In-memory database limitations

Neutral:

- Coverage exclusions need justification

Implementation Details

Test Project Structure

Each service has a corresponding test project:

```
Tests/
├── Unit/
│   ├── Controllers/
│   │   └── *ControllerTests.cs
│   ├── Handlers/
│   │   └── *HandlerTests.cs
│   ├── Services/
│   │   └── *ServiceTests.cs
│   └── Repositories/
│       └── *RepositoryTests.cs
├── Integration/
│   ├── *IntegrationTests.cs
│   └── TestWebApplicationFactory.cs
└── Helpers/
    └── TestHelpers.cs
```

Testing Patterns

Unit Test Example (Handler)

```
public class RegisterUserHandlerTests
{
    private readonly Mock<IUsersRepository> _repositoryMock;
    private readonly Mock<IPasswordHasher> _passwordHasherMock;
    private readonly RegisterUserHandler _handler;

    public RegisterUserHandlerTests()
    {
        _repositoryMock = new Mock<IUsersRepository>();
        _passwordHasherMock = new Mock<IPasswordHasher>();
        _handler = new RegisterUserHandler(
            _repositoryMock.Object,
            _passwordHasherMock.Object);
    }

    [Fact]
    public async Task HandleAsync_WithValidCommand_ShouldCreateUser()
    {
        // Arrange
        var command = new RegisterUserCommand
        {
            Email = "test@example.com",
            Password = "Password123!",
            FirstName = "John",
            LastName = "Doe",
            Role = UserRole.User
        };

        _repositoryMock.Setup(r => r.GetByEmailAsync(command.Email))
            .ReturnsAsync((User?)null);
        _passwordHasherMock.Setup(p => p.HashPassword(command.Password))
            .Returns("hashed_password");

        // Act
        var result = await _handler.HandleAsync(command);

        // Assert
        result.Should().NotBeNull();
```

```
        result.Email.Should().Be(command.Email);
        _repositoryMock.Verify(r => r.AddAsync(It.IsAny<User>()), Times.Once);
    }

    [Fact]
    public async Task HandleAsync_WithExistingEmail_ShouldThrowException()
    {
        // Arrange
        var command = new RegisterUserCommand { Email = "existing@example.com" };
        _repositoryMock.Setup(r => r.GetByEmailAsync(command.Email))
            .ReturnsAsync(new User());

        // Act & Assert
        await _handler.Invoking(h => h.HandleAsync(command))
            .Should().ThrowAsync<InvalidOperationException>()
            .WithMessage("*already registered*");
    }
}
```

Integration Test Example

```
public class AuthControllerIntegrationTests : IClassFixture<TestWebApplicationFactory>
{
    private readonly HttpClient _client;
    private readonly TestWebApplicationFactory _factory;

    public AuthControllerIntegrationTests(TestWebApplicationFactory factory)
    {
        _factory = factory;
        _client = factory.CreateClient();
    }

    [Fact]
    public async Task Register_WithValidData_ShouldReturn201()
    {
        // Arrange
        var command = new
        {
            Email = $"test-{Guid.NewGuid()}@example.com",
            Password = "Password123!",
            FirstName = "Test",
            LastName = "User",
            Role = "User"
        };

        // Act
        var response = await _client.PostAsJsonAsync("/api/auth/register", command);

        // Assert
        response.StatusCode.Should().Be(HttpStatusCode.Created);
        var result = await response.Content.ReadFromJsonAsync<RegisterUserResult>();
        result.Should().NotBeNull();
        result!.Email.Should().Be(command.Email);
    }

    [Fact]
    public async Task Login_WithInvalidCredentials_ShouldReturn401()
    {
        // Arrange
        var command = new { Email = "nonexistent@example.com", Password = "wrong" };
```

```
        // Act
        var response = await _client.PostAsJsonAsync("/api/auth/login", command);

        // Assert
        response.StatusCode.Should().Be(HttpStatusCode.Unauthorized);
    }
}
```

Test Web Application Factory

```
public class TestWebApplicationFactory : WebApplicationFactory<Program>
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.UseEnvironment("Testing");

        builder.ConfigureServices(services =>
        {
            // Replace SQL Server with In-Memory database
            var descriptor = services.SingleOrDefault(
                d => d.ServiceType == typeof(DbContextOptions<UsersDbContext>));
            if (descriptor != null)
                services.Remove(descriptor);

            services.AddDbContext<UsersDbContext>(options =>
            {
                options.UseInMemoryDatabase($"TestDb-{Guid.NewGuid()}");
            });

            // Configure test authentication
            services.AddAuthentication("Test")
                .AddScheme<AuthenticationSchemeOptions, TestAuthHandler>("Test", null);
        });
    }
}
```

Naming Conventions

Test Methods: `{MethodName}_{Scenario}_{ExpectedBehavior}`

Examples:

- `GetByIdAsync_WithExistingId_ShouldReturnReservation`
- `Create_ShouldReturnForbidden_WhenCreatingForDifferentOwner`
- `Login_WithValidCredentials_ShouldReturnToken`
- `Update_ShouldAllowNullOptionalValues`

Coverage Exclusions

Code excluded from coverage metrics (justified):

```
<!-- Test .csproj -->
<Exclude>
  [*]*.Migrations.*,
  [*]*ModelSnapshot,
  [*]*DbContextFactory,
  [*]*DataSeeder*,
  [*]*.Configurations.*
</Exclude>

<ExcludeByAttribute>
  Obsolete,
  GeneratedCode,
  CompilerGenerated,
  ExcludeFromCodeCoverage
</ExcludeByAttribute>
```

Rationale:

- **Migrations**: Auto-generated EF Core code
- **DbContextFactory**: Design-time tooling only
- **DataSeeder**: Development/test utilities
- **Middleware**: Framework infrastructure with no business logic
- **Swagger Filters**: Documentation generation

Running Tests

```
# Run all tests for a service
cd hotel-booking-users-service
dotnet test

# Run tests with coverage
.\run-tests-with-coverage.ps1
```

Coverage Report Generation Script

```
# run-tests-with-coverage.ps1

# Clean previous coverage
Remove-Item -Recurse -Force ./CoverageReport -ErrorAction SilentlyContinue

# Run tests with Coverlet
dotnet test `
    --collect:"XPlat Code Coverage" `
    --results-directory ./CoverageReport `
    -- DataCollectionRunSettings.DataCollectors.DataCollector.Configuration.Format=cobertura

# Install ReportGenerator if needed
dotnet tool install -g dotnet-reportgenerator-globaltool

# Generate HTML report
reportgenerator `
    -reports:./CoverageReport/**/coverage.cobertura.xml `
    -targetdir:./CoverageReport/Report `
```

```
    -reporttypes:Html

# Open report
Start-Process ./CoverageReport/Report/index.html
```

Coverage Results

Service	Tests	Line Coverage	Branch Coverage
Users	~200	>70%	>70%
Hotels	~200	>70%	>70%
Reservations	~200	>70%	>70%
Payments	~190	>70%	>70%
Total	**~790**	**>70%**	**>70%**

FIRST Principles Compliance

Principle	Implementation
Fast	Unit tests < 1s, integration tests 2-5s per service
Independent	Each test uses unique database identifier
Repeatable	No shared state, deterministic data
Self-Validating	Clear pass/fail with FluentAssertions
Timely	Tests written alongside implementation

Requirements Checklist

Minimum Requirements (Grade 5)

#	Requirement	Status	Evidence
1	Structured automated testing setup	✅	xUnit with FluentAssertions
2	At least 70% code coverage	✅	75-85% per service via Coverlet
3	Unit tests with FIRST principles	✅	Fast, Independent, Repeatable tests
4	Edge cases covered	✅	Validation, error scenarios tested
5	Integration tests	✅	WebApplicationFactory for API tests

#	Requirement	Status	Evidence
6	Frontend tests (if applicable)	N/A	Skipped - Frontend criterion not selected
7	Tests run in CI	N/A	Skipped - CI/CD criterion not selected
8	Testing strategy documented	✅	This document
9	Clear test naming/structure	✅	Tests/Unit/, Tests/Integration/

Maximum Requirements (Grade 10)

#	Requirement	Status	Notes
1	Multi-layer testing (unit, integration, E2E)	⚠️ Partial	no E2E
2	Mocking, fixtures, test doubles	✅	Moq for all dependencies
3	Complex integration scenarios	⚠️ Partial	Single-service focus
4	Property-based/mutation/contract testing	⚠️ Partial	Contract testing implemented
5	CI/CD with quality gates	N/A	CI/CD not selected as criterion
6	Test architecture rationale	✅	Documented in this file
7	Performance/load testing	❌	Not implemented

Known Limitations

Limitation	Impact	Potential Solution
In-memory DB limitations	Some EF features not supported	Use SQL Server TestContainers
No E2E tests	Full flow not tested	Add Playwright or Selenium tests
No load tests	Performance unknown	Add k6 or similar
Limited webhook testing	Stripe webhooks hard to test	WireMock for webhook simulation

References

- xUnit Documentation
- FluentAssertions
- Coverlet
- ASP.NET Core Integration Testing

- Test projects: `Tests/` folder in each service
- Coverage scripts: `run-tests-with-coverage.ps1` in each service

---

3. User Guide

This section provides instructions for end users on how to use the application via Swagger UI.

Contents

- Features Walkthrough
- FAQ & Troubleshooting

Getting Started

System Requirements

Requirement	Minimum	Recommended
Browser	Chrome 90+, Firefox 88+, Safari 14+, Edge 90+	Latest version
Screen Resolution	1280x720	1920x1080
Internet	Required	Stable connection
Device	Desktop	-

Accessing the Application

The Hotel Booking Platform is accessed via Swagger UI for API interaction:

1. Open your web browser
2. Navigate to one of the service Swagger URLs:
   - **Users Service**: https://hotel-booking-users-api-csbghtd2f9cph7g5.northeurope-01.azurewebsites.net/swagger/index.html
   - **Hotels Service**: https://hotel-booking-hotels-api-evhhefafhhbrgrbs.northeurope-01.azurewebsites.net/swagger/index.html
   - **Reservations Service**: https://hotel-booking-reservations-api-dwfzh9bydth3fke0.northeurope-01.azurewebsites.net/swagger/index.html
   - **Payments Service**: https://hotel-booking-payments-api-gch8e7fyeqenfje8.northeurope-01.azurewebsites.net/swagger/index.html

First Launch

Step 1: Register a User Account

1. Go to Users Service Swagger UI
2. Expand `POST /api/auth/register`
3. Click "Try it out"
4. Enter registration details:

```
{
  "email": "your-email@example.com",
  "password": "YourPassword123!",
}
```

5. Click "Execute"
6. You should receive a 201 Created response

Step 2: Login to Get Token

1. Expand `POST /api/auth/login`
2. Click "Try it out"
3. Enter your credentials:

```
{
  "email": "your-email@example.com",
  "password": "YourPassword123!"
}
```

4. Click "Execute"
5. Copy the `accessToken` from the response

Step 3: Authorize Swagger UI

1. Click the "Authorize" button (lock icon) at the top
2. Enter: `Bearer <your-access-token>`
3. Click "Authorize"
4. Now all authenticated endpoints will include your token

Quick Start Guide

Task	Service	Endpoint
Register account	Users	`POST /api/auth/register`

Task	Service	Endpoint
Login	Users	`POST /api/auth/login`
Search hotels	Hotels	`GET /api/hotels/search`
View hotel details	Hotels	`GET /api/hotels/{id}`
Create reservation	Reservations	`POST /api/reservations`
Pay for booking	Payments	`POST /api/payments/create-intent`
View my bookings	Reservations	`GET /api/reservations/mine`
Cancel booking	Reservations	`POST /api/reservations/{id}/cancel`

User Roles

Role	Permissions	Access Level
User	Search hotels, create reservations, make payments, view own bookings	Basic user access
HotelOwner	All User permissions + submit hotels, manage rooms, view hotel bookings	Property management
Admin	All permissions + approve hotels, manage cancellation requests, view all data	Full system access

Test Accounts (Seeded Data)

The system is seeded with test accounts:

Email	Password	Role
admin@example.com	(configured in system)	Admin
owner@example.com	(configured in system)	HotelOwner
user@example.com	(configured in system)	User

Contact your administrator for the seeded account passwords.

Feature Walkthrough

Feature 1: User Registration & Authentication

Overview

Register for an account and authenticate to access the hotel booking platform. The system uses JWT tokens for secure, stateless authentication.

How to Use

[Screenshot:  Feature 1]

Step 1: Go to Users Service Swagger UI and expand `POST /api/auth/register`

Step 2: Click "Try it out" and enter your details:

```
{
  "email": "alice@example.com",
  "password": "SecurePass123!",

}
```

Step 3: Click "Execute" - you should receive a 201 Created response

Step 4: Login via `POST /api/auth/login` with your credentials

Step 5: Copy the `accessToken` from the response and use it to authorize Swagger UI

Expected Result: You receive JWT access token (valid 60 minutes) and refresh token (valid 7 days)

Tips

- Passwords must be at least 8 characters
- Email must be unique in the system

Feature 2: Hotel Search

Overview

Search for available hotels based on location, dates, guest count, and amenities. Only approved hotels with available rooms are returned.

How to Use

[Screenshot: Feature 2]

Step 1: Go to Hotels Service Swagger UI

Step 2: Expand `GET /api/hotels/search`

Step 3: Enter search parameters:

- `country` : Lithuania (optional)
- `city` : Vilnius (optional)
- `checkIn` : 2026-02-01
- `checkOut` : 2026-02-05
- `guestsCount` : 2
- `petsAllowed` : false (optional)
- `minPrice` : 50 (optional)
- `maxPrice` : 200 (optional)

Step 4: Click "Execute"

Expected Result: List of matching hotels with name, location, minimum price, and available room count

Tips

- Leave optional parameters empty for broader search
- Results only include approved hotels
- Check `cancelFreeDaysBefore` to understand cancellation policy

Feature 3: Create Reservation

Overview

Book a room for your desired dates. Reservations start with "Pending" status until payment is completed.

How to Use

[Screenshot: Feature 3]

Step 1: Go to Reservations Service Swagger UI (authorized with your token)

Step 2: Expand `POST /api/reservations`

Step 3: Enter reservation details:

```
{
  "roomId": 55,
  "startDate": "2026-02-01",
  "endDate": "2026-02-05",
  "guestsCount": 2,
  "guestsNames": "Alice Johnson, Bob Smith"
}
```

Step 4: Click "Execute"

Expected Result: Reservation created with "Pending" status and reservation ID returned

Tips

- Room must be available for selected dates
- Guest count must not exceed room capacity
- Reservation remains Pending until payment is confirmed

Feature 4: Payment Processing

Overview

Pay for your reservation using Stripe integration. Payment confirmation changes reservation status to "Confirmed".

How to Use

[Screenshot: Feature 4]

Step 1: Go to Payments Service Swagger UI (authorized with your token)

Step 2: Expand `POST /api/payments/create-intent`

Step 3: Create payment intent:

```
{
  "reservationId": 150,
  "amount": 356.00,
  "currency": "EUR"
}
```

Step 4: Note the `clientSecret` from response (used for frontend payment confirmation)

Step 5: For testing, use Stripe test card: `4242 4242 4242 4242`

Expected Result: Payment intent created, Stripe processes payment via webhook, reservation becomes "Confirmed"

Tips

- Amount should match total reservation cost (nights × price per night)
- Use Stripe test cards in test mode
- Payment status updates automatically via Stripe webhooks

Feature 5: Cancellation Management

Overview

Cancel reservations with automatic or admin-approved refunds depending on cancellation policy timing.

How to Use

[Screenshot: Feature 5]

Step 1: Go to Reservations Service Swagger UI

Step 2: Expand `POST /api/reservations/{id}/cancel`

Step 3: Enter cancellation request:

```
{
  "reason": "Change of travel plans"
}
```

Step 4: Click "Execute"

Expected Result:

- **Within free cancellation period**: Auto-canceled with automatic refund
- **Outside free period**: Request submitted for admin review

Cancellation Policy Logic

Timing	Result
More than X days before check-in	Auto-cancel + full refund
Less than X days before check-in	Requires admin approval

(X = hotel's `cancelFreeDaysBefore` setting)

Feature 6: Hotel Management (Hotel Owners)

[Screenshot: Feature 6]

Overview

Hotel owners can submit hotels for approval, manage room inventory, and view reservations on their properties.

How to Use

Step 1: Register or login as HotelOwner role

Step 2: Submit a new hotel via `POST /api/hotels`:

```
{
  "name": "My Hotel",
  "description": "A lovely hotel in the city center",
  "country": "Latvia",
  "city": "Riga",
  "district": "Old Town",
  "addressLine": "Main Street 123",
  "petsAllowed": true,
  "cancelFreeDaysBefore": 7
}
```

Step 3: Wait for admin approval (status: Pending → Approved)

Step 4: Add rooms via `POST /api/rooms`:

```
{
  "hotelId": 1,
  "roomNumber": "101",
  "description": "Comfortable room with city view",
  "capacity": 2,
  "bedrooms": 1,
  "pricePerNight": 89.00,
  "visible": true,
  "petsAllowed": false,
  "accommodation": "HotelRoom"
}
```

Step 5: View reservations on your hotel via `GET /api/hotels/{id}/reservations`:

- See all bookings on your hotel's rooms
- View guest details, dates, and status
- Track cancellation requests

Expected Result: Hotel submitted for approval, rooms added once hotel is approved, reservations visible to owner

---

Feature 7: Admin Operations

[Screenshot: Feature 7]

Overview

Administrators can approve hotels, handle cancellation requests, and manage the platform.

How to Use

Approve Hotels:

1. `GET /api/admin/hotels/pending` - View pending hotels
2. `POST /api/admin/hotels/{id}/approve` - Approve a hotel
3. `POST /api/admin/hotels/{id}/reject` - Reject a hotel

Handle Cancellations:

1. `GET /api/admin/reservations/cancellations` - View pending requests
2. `POST /api/admin/reservations/{id}/approve-cancel` - Approve with refund
3. `POST /api/admin/reservations/{id}/reject-cancel` - Reject request

View All Data:

- `GET /api/users` - All users
- `GET /api/admin/hotels` - All hotels (any status)
- `GET /api/admin/reservations` - All reservations

Expected Result: Full platform management capabilities

API Authentication Flow

Step	Action	Result
1	Register	Account created
2	Login	Receive access + refresh tokens
3	Use access token	Access protected endpoints

Step	Action	Result
4	Token expires (60 min)	Call refresh endpoint
5	Refresh token	Receive new access + refresh tokens
6	Logout	Refresh token revoked

FAQ & Troubleshooting

Frequently Asked Questions

General

Q: What is the Hotel Booking Platform?

A: The Hotel Booking Platform is a microservices-based system for hotel reservations. It consists of four services: Users (authentication), Hotels (property management), Reservations (bookings), and Payments (Stripe integration).

Q: How do I access the platform?

A: The platform is accessed via Swagger UI at each service's URL. There is no traditional web frontend - API testing is done directly through Swagger's interactive interface.

Q: What user roles are available?

A: Three roles exist:

- **User**: Can search hotels, make reservations, and manage own bookings
- **HotelOwner**: Can submit hotels for approval and manage rooms
- **Admin**: Full system access including approvals and cancellation management

---

Account & Access

Q: How do I create an account?

A: Use the `POST /api/auth/register` endpoint in the Users Service with your email, password, name, and desired role.

Q: My access token expired. What do I do?

A: Use the `POST /api/auth/refresh` endpoint with your refresh token to get a new access token. Access tokens expire after 60 minutes, refresh tokens after 7 days.

Q: Can I change my role after registration?

A: Yes, regular Users can upgrade to HotelOwner role using the `POST /api/auth/become-hotel-owner` endpoint. This returns new JWT tokens with the updated role.

Q: How do I logout?

A: Use the `POST /api/auth/logout` endpoint. This revokes your refresh token, though the access token remains valid until it expires.

Booking & Reservations

Q: Why is my reservation still "Pending"?

A: Reservations remain Pending until payment is completed. Create a payment intent and confirm payment through Stripe to change status to Confirmed.

Q: Can I cancel a reservation?

A: Yes. Use `POST /api/reservations/{id}/cancel`. If within the hotel's free cancellation period, it's auto-canceled with refund. Otherwise, it requires admin approval.

Q: How is the cancellation policy determined?

A: Each hotel sets a `cancelFreeDaysBefore` value. If you cancel more than this many days before check-in, you get automatic cancellation and refund. Otherwise, admin review is required.

Payments

Q: What payment methods are supported?

A: The system uses Stripe for payments. In test mode, use Stripe test cards. Credit/debit cards are supported.

Q: How do I test payments without real money?

A: Use Stripe test card `4242 4242 4242 4242` with any future expiration date and any 3-digit CVC.

Q: My payment failed. What happened?

A: Check the payment status via `GET /api/payments/{id}`. The `errorMessage` field will indicate the reason (insufficient funds, card declined, etc.).

Troubleshooting

Common Issues

Problem	Possible Cause	Solution
401 Unauthorized	Token expired or missing	Refresh your token or re-login
403 Forbidden	Insufficient permissions	Check your role has access to this endpoint
404 Not Found	Resource doesn't exist	Verify the ID is correct
400 Bad Request	Invalid input data	Check request body format and required fields
Room not available	Dates overlap with existing reservation	Choose different dates or room
Hotel not visible	Hotel not approved yet	Wait for admin approval

Error Messages

Error Code/Message	Meaning	How to Fix
"Email already registered"	Duplicate email address	Use a different email or login to existing account
"Invalid credentials"	Wrong email or password	Verify credentials and try again
"Token has expired"	Access token expired	Use refresh token to get new access token
"Room is not available"	Booking conflict	Select different dates or another room

Error Code/Message	Meaning	How to Fix
"Reservation not found"	Invalid reservation ID	Verify the reservation ID exists
"Insufficient funds"	Payment card declined	Use different payment method
"Hotel not approved"	Hotel pending approval	Wait for admin to approve the hotel

Authentication Issues

Issue	Solution
"Bearer" not included	Format token as: `Bearer <your-token>`
Token not working	Ensure no extra spaces or characters
Swagger not authorized	Click "Authorize" button and enter token
Refresh token invalid	Re-login to get new refresh token

API-Specific Issues

Service	Issue	Solution
Users	Registration fails	Check email format, password length (8+ chars)
Hotels	Search returns empty	Verify location exists, dates are valid
Reservations	Cannot create	Ensure room exists, dates available, guest count valid
Payments	Intent fails	Verify reservation ID correct, amount positive

Getting Help

Self-Service Resources

- This documentation
- Swagger UI endpoint descriptions
- API response error messages

Service Health Checks

Check if services are running:

- Users: `https://hotel-booking-users-api-*.azurewebsites.net/health`
- Hotels: `https://hotel-booking-hotels-api-*.azurewebsites.net/health`

- Reservations: `https://hotel-booking-reservations-api-*.azurewebsites.net/health`
- Payments: `https://hotel-booking-payments-api-*.azurewebsites.net/health`

Reporting Issues

When reporting a problem, please include:

1. **Service and endpoint** - Which API and endpoint were you calling?
2. **Request body** - What data did you send?
3. **Response** - What status code and message did you receive?
4. **Token info** - Was the request authenticated? What role?
5. **Steps to reproduce** - What actions led to the issue?

GitHub Repositories

- Users: https://github.com/Patsino/hotel-booking-users-service
- Hotels: https://github.com/Patsino/hotel-booking-hotels-service
- Reservations: https://github.com/Patsino/hotel-booking-reservations-service
- Payments: https://github.com/Patsino/hotel-booking-payments-service

---

4. Retrospective

This section reflects on the project development process, lessons learned, and future improvements.

What Went Well ✅

Technical Successes

- **Clean Architecture implementation** worked excellently, providing clear separation of concerns and testability across all four microservices
- **Entity Framework Core migrations** streamlined database schema management and made deployments predictable
- **JWT authentication with refresh tokens** provided secure, stateless authentication that scaled well
- **Stripe integration** proved straightforward with excellent documentation and test mode support
- **Docker Compose** made local development simple with consistent environment setup
- **Azure Free Tier deployment** successfully demonstrated cloud-native capabilities without cost

Process Successes

- **Microservices-first approach** allowed independent development and deployment of each service
- **API-first design** using Swagger/OpenAPI provided clear contracts between services
- **Test-driven development** caught issues early and provided confidence during refactoring
- **Consistent project structure** across services reduced cognitive load when switching contexts

Personal Achievements

- Gained practical experience with microservices architecture patterns
- Learned Azure cloud deployment and configuration
- Developed skills in payment gateway integration (Stripe)
- Improved understanding of JWT security and token lifecycle

What Didn't Go As Planned ⚠️

Planned	Actual Outcome	Cause	Impact
Backend-only project	Created React frontend for demos	Swagger UI difficult to demonstrate to non-technical evaluators	High
CI/CD pipeline	Manual deployment via Visual Studio	CI/CD not selected as evaluation criterion	Low
Inter-service messaging	Direct HTTP communication	Complexity vs. timeline trade-off	Low
Real-time notifications	Not implemented	Would require additional infrastructure	Low

Challenges Encountered

1. **Demonstrating Backend to Non-Technical Stakeholders**
    - Problem: Swagger UI works well for developers but is confusing for non-technical evaluators
    - Impact: Difficult to show complete booking flow during demonstrations
    - Resolution: Built a simple React frontend specifically for demo purposes
2. **No CI/CD Pipeline**
    - Problem: Manual deployment process via Visual Studio Publish
    - Impact: More effort for deployments, no automated testing on push
    - Reason: Implementing a CI/CD pipeline would require considerable additional time and configuration effort and was therefore deprioritized, as CI/CD was not part of the evaluation criteria; deployment done manually to Azure
3. **Azure Free Tier Limitations**

- Problem: Cold start delays and limited resources on F1 tier
- Impact: First requests after idle period are slow (10-30 seconds)
- Resolution: Documented as expected behavior, acceptable for diploma demonstration

Technical Debt & Known Issues

ID	Issue	Severity	Description	Potential Fix
TD-001	No CI/CD	Medium	Manual deployment via Visual Studio	GitHub Actions or Azure DevOps pipeline
TD-002	No message queue	Medium	Direct HTTP calls between services	Azure Service Bus for async communication
TD-003	No caching layer	Low	No Redis for frequent queries	Add distributed cache

Future Improvements (Backlog)

If there was more time, these features/improvements would be prioritized:

High Priority

1. **Message Queue Integration**
   - Description: Implement Azure Service Bus for async communication
   - Value: Improves reliability and decoupling between services
   - Effort: Medium - requires infrastructure changes and handler refactoring
2. **API Gateway**
   - Description: Add centralized gateway for routing, authentication, and rate limiting
   - Value: Single entry point, improved security, better monitoring
   - Effort: Medium - Azure API Management or Kong setup

Medium Priority

3. **Full Frontend Application**
   - Description: Complete React/TypeScript frontend with proper UI/UX
   - Value: Better user experience for actual production use
   - Effort: High - significant frontend development work
4. **Caching Layer**

- Description: Add Redis for caching hotel searches and user sessions
- Value: Improved performance, reduced database load
- Effort: Medium - infrastructure and code changes

Nice to Have

```
- Email notifications for booking confirmations and reminders
- Multi-language support
- Hotel owner analytics dashboard
```

Lessons Learned

Technical Lessons

Lesson	Context	Application
Clean Architecture pays off	Easy testing and modification of business logic	Apply to all future .NET projects
Start with authentication	User service must be solid before other services	Build auth infrastructure first
Azure free tier is capable	Ran full microservices system at zero cost	Use for MVPs and learning projects
Stripe sandbox is essential	Tested all payment flows without real money	Always develop against sandbox first

Process Lessons

Lesson	Context	Application
Document as you build	API documentation emerged naturally from code	Use Swagger annotations throughout
Test early, test often	Caught integration issues between services early	Maintain high test coverage
Keep services focused	Each service doing one thing well	Resist adding unrelated features

What Would Be Done Differently

Area	Current Approach	What Would Change	Why
CI/CD	Manual deployment	Add GitHub Actions pipeline	Automated testing and deployment
Frontend	Added late for demos	Consider from start or skip entirely	Better planning for demo requirements
Messaging	HTTP only	Add message broker if async needed	More resilient communication
Monitoring	Basic health checks	Add Application Insights	Better debugging in production

Personal Growth

Skills Developed

Skill	Before Project	After Project
Microservices Architecture	Intermediate	Advanced
Azure Cloud Services	Intermediate	Advanced
Docker & Containerization	Intermediate	Advanced
.NET 9 / ASP.NET Core	Intermediate	Advanced
Payment Integration	Beginner	Intermediate
Clean Architecture	Intermediate	Advanced

Key Takeaways

1. **Microservices require careful planning** - service boundaries and communication patterns must be well-defined upfront
2. **Cloud deployment is accessible** - Azure free tier makes it possible to deploy production-grade systems without cost
3. **Testing is non-negotiable** - high test coverage enables confident refactoring and deployments
4. **Documentation must serve its audience** - Swagger is great for developers, but demos for non-technical stakeholders need visual interfaces

---

Retrospective completed: 2026-01-05

API Reference

Overview

The Hotel Booking Platform consists of four microservices, each with its own API:

Service	Base URL (Local)	Base URL (Azure)
Users	`http://localhost:8081/api`	`https://hotel-booking-users-api-*.azurewebsites.net/api`
Hotels	`http://localhost:8082/api`	`https://hotel-booking-hotels-api-*.azurewebsites.net/api`
Reservations	`http://localhost:8083/api`	`https://hotel-booking-reservations-api-*.azurewebsites.net/api`
Payments	`http://localhost:8084/api`	`https://hotel-booking-payments-api-*.azurewebsites.net/api`

Authentication: Bearer Token (JWT)

Format: All requests and responses use JSON

Users Service

Authentication

POST /auth/register

Register a new user account.

Request Body:

```
{
  "email": "user@example.com",
  "password": "Password123!",
}
```

Field	Type	Required	Description
email	string	Yes	Valid email address
password	string	Yes	Minimum 8 characters

Response: `201 Created`

```
{
  "id": "guid",
  "email": "user@example.com",
}
```

POST /auth/login

Authenticate and receive tokens.

Request Body:

```
{
  "email": "user@example.com",
  "password": "Password123!"
}
```

Response: `200 OK`

```
{
  "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "refreshToken": "dGhpcyBpcyBhIHJlZnJlc2ggdG9rZW4...",
  "expiresAt": "2026-01-05T14:00:00Z"
}
```

POST /auth/refresh

Refresh access token using refresh token.

Request Body:

```
{
  "refreshToken": "current-refresh-token"
}
```

Response: `200 OK`

```
{
  "accessToken": "new-access-token",
  "refreshToken": "new-refresh-token",
  "expiresAt": "2026-01-05T15:00:00Z"
}
```

POST /auth/logout

Revoke refresh token.

Headers: `Authorization: Bearer <access-token>`

Request Body:

```
{
  "refreshToken": "refresh-token-to-revoke"
}
```

Response: `200 OK`

Users

GET /users/me

Get current user profile.

Headers: `Authorization: Bearer <access-token>`

Response: `200 OK`

```
{
  "id": "guid",
  "email": "user@example.com",
  "role": "User",
  "createdAt": "2026-01-01T00:00:00Z"
}
```

GET /users/{id}

Get user by ID (Admin only).

Headers: `Authorization: Bearer <access-token>`

Response: `200 OK`

Hotels Service

Hotels

GET /hotels

Search for hotels.

Query Parameters:

Parameter	Type	Required	Description
location	string	No	Filter by city/location
checkIn	date	No	Check-in date (YYYY-MM-DD)
checkOut	date	No	Check-out date (YYYY-MM-DD)
guests	integer	No	Number of guests
page	integer	No	Page number (default: 1)
pageSize	integer	No	Items per page (default: 10)

Response: `200 OK`


```
{
  "items": [
    {
      "id": "guid",
      "name": "Grand Hotel",
      "location": "Paris, France",
      "description": "Luxury hotel in city center",
      "rating": 4.5,
      "minPrice": 150.00,
      "imageUrl": "https://..."
    }
  ],
  "totalCount": 50,
  "page": 1,
  "pageSize": 10
}
```

GET /hotels/{id}

Get hotel details with rooms.

Response: `200 OK`


```
{
  "id": "guid",
  "name": "Grand Hotel",
  "location": "Paris, France",
  "description": "Luxury hotel...",
  "cancelFreeDaysBefore": 3,
  "ownerId": "guid",
  "isApproved": true,
  "rooms": [
```

```
    {
      "id": "guid",
      "roomNumber": "101",
      "roomType": "Deluxe",
      "pricePerNight": 200.00,
      "maxGuests": 2,
      "description": "King bed with city view"
    }
  ]
}
```

POST /hotels

Create a new hotel (HotelOwner only).

Headers: `Authorization: Bearer <access-token>`

Request Body:

```
{
  "name": "My Hotel",
  "location": "London, UK",
  "description": "Boutique hotel in Soho",
  "cancelFreeDaysBefore": 5
}
```

Response: `201 Created`

PUT /hotels/{id}

Update hotel details (Owner only).

DELETE /hotels/{id}

Delete hotel (Owner only).

GET /hotels/{id}/reservations

Get all reservations for a hotel (Owner or Admin only).

Headers: `Authorization: Bearer <access-token>`

Response: `200 OK`

```
[
  {
```

```
    "id": 123,
    "userId": 42,
    "roomId": 55,
    "roomNumber": "101",
    "startDate": "2024-12-20",
    "endDate": "2024-12-22",
    "guestsCount": 2,
    "guestsNames": "John Doe, Jane Smith",
    "status": "Confirmed",
    "cancellationStatus": "None",
    "cancellationReason": null,
    "cancellationRequestedAt": null,
    "createdAt": "2024-12-15T10:30:00Z"
  }
]
```

Field	Description
status	Pending, Held, Confirmed, Canceled
cancellationStatus	None, Requested, AutoCanceled, AdminApproved, AdminRejected

Response Codes:

- `200 OK` - Reservations returned
- `401 Unauthorized` - Not authenticated
- `403 Forbidden` - Not owner or admin
- `404 Not Found` - Hotel not found

POST /hotels/{id}/approve

Approve hotel (Admin only).

Headers: `Authorization: Bearer <access-token>`

Response: `200 OK`

Rooms

POST /hotels/{hotelId}/rooms

Add room to hotel (Owner only).

Request Body:

```
{
  "roomNumber": "102",
  "roomType": "Suite",
  "pricePerNight": 350.00,
```

```
  "maxGuests": 4,
  "description": "Two bedroom suite"
}
```

PUT /hotels/{hotelId}/rooms/{roomId}

Update room details.

DELETE /hotels/{hotelId}/rooms/{roomId}

Remove room from hotel.

GET /hotels/{hotelId}/rooms/{roomId}/availability

Check room availability for dates.

Query Parameters:

Parameter	Type	Required	Description
checkIn	date	Yes	Check-in date
checkOut	date	Yes	Check-out date

Response: `200 OK`

```
{
  "isAvailable": true,
  "roomId": "guid",
  "checkIn": "2026-02-01",
  "checkOut": "2026-02-05"
}
```

Reservations Service

Reservations

GET /reservations

Get user's reservations.

Headers: `Authorization: Bearer <access-token>`

Query Parameters:

Parameter	Type	Description
status	string	Filter by status
page	integer	Page number

Response: `200 OK`

```
{
  "items": [
    {
      "id": "guid",
      "hotelId": "guid",
      "hotelName": "Grand Hotel",
      "roomId": "guid",
      "roomNumber": "101",
      "checkIn": "2026-02-01",
      "checkOut": "2026-02-05",
      "guestCount": 2,
      "totalPrice": 800.00,
      "status": "Confirmed"
    }
  ]
}
```

GET /reservations/{id}

Get reservation details.

POST /reservations

Create a new reservation.

Headers: `Authorization: Bearer <access-token>`

Request Body:

```
{
  "hotelId": "guid",
  "roomId": "guid",
  "checkIn": "2026-02-01",
  "checkOut": "2026-02-05",
  "guestCount": 2
}
```

Response: `201 Created`

```
{
  "id": "guid",
  "status": "Pending",
  "totalPrice": 800.00
}
```

POST /reservations/{id}/cancel

Request reservation cancellation.

Headers: `Authorization: Bearer <access-token>`

Response: `200 OK`


```
{
  "id": "guid",
  "status": "Cancelled",
  "cancellationType": "FreeCancellation"
}
```


Or for late cancellations:


```
{
  "id": "guid",
  "status": "CancellationPending",
  "cancellationType": "RequiresApproval"
}
```

POST /reservations/{id}/approve-cancellation

Approve cancellation (Admin only).

POST /reservations/{id}/reject-cancellation

Reject cancellation (Admin only).

Payments Service

Payments

POST /payments/create-intent

Create a Stripe payment intent.

Headers: `Authorization: Bearer <access-token>`

Request Body:

```
{
  "reservationId": "guid",
  "amount": 800.00,
  "currency": "usd"
}
```

Response: `200 OK`

```
{
  "paymentId": "guid",
  "clientSecret": "pi_xxx_secret_xxx",
  "amount": 800.00,
  "currency": "usd",
  "status": "RequiresPayment"
}
```

POST /payments/{id}/confirm

Confirm payment after Stripe processing.

Request Body:

```
{
  "paymentMethodId": "pm_card_visa"
}
```

Response: `200 OK`

```
{
  "paymentId": "guid",
  "status": "Succeeded",
  "paidAt": "2026-01-05T12:00:00Z"
}
```

GET /payments/{id}

Get payment details.

POST /payments/{id}/refund

Process refund (Admin or System).

Request Body:

```
{
  "reason": "Cancellation within free period"
}
```

Response: `200 OK`

```
{
  "refundId": "guid",
  "amount": 800.00,
  "status": "Succeeded"
}
```

POST /payments/webhook

Stripe webhook endpoint for payment events.

Note: This endpoint is called by Stripe, not by clients directly.

Error Responses

All services return errors in a consistent format:

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "Bad Request",
  "status": 400,
  "detail": "Validation failed",
  "errors": {
    "email": ["The Email field is required."]
  }
}
```

Status Code	Description
400	Bad Request - Invalid input data
401	Unauthorized - Missing or invalid token
403	Forbidden - Insufficient permissions
404	Not Found - Resource doesn't exist
409	Conflict - Resource already exists or state conflict

Status Code	Description
500	Internal Server Error - Unexpected error

Swagger/OpenAPI

Interactive API documentation is available at each service's `/swagger` endpoint:

- Users: `http://localhost:8081/swagger`
- Hotels: `http://localhost:8082/swagger`
- Reservations: `http://localhost:8083/swagger`

- Payments: `http://localhost:8084/swagger`

Internal Service-to-Service Endpoints

These endpoints are used for inter-service communication and require API key authentication via `X-API-Key` header.

Reservations Service - Internal

POST /internal/reservations/batch-availability

Check availability for multiple rooms in a date range.

Headers: `X-API-Key: <service-api-key>`

Request Body:

```
{
  "roomIds": [1, 2, 3],
  "startDate": "2024-12-20",
  "endDate": "2024-12-25"
}
```

Response: `200 OK`

```
{
  "unavailableRoomIds": [2]
}
```

POST /internal/reservations/by-rooms

Get all reservations for specified rooms (used by Hotels Service for owner view).

Headers: `X-API-Key: <service-api-key>`

Request Body:

```
{
  "roomIds": [1, 2, 3]
}
```

Response: `200 OK`

```
[
  {
    "id": 123,
    "userId": 42,
    "roomId": 1,
    "startDate": "2024-12-20",
    "endDate": "2024-12-22",
    "guestsCount": 2,
    "guestsNames": "John Doe",
    "status": "Confirmed",
    "cancellationStatus": "None",
    "createdAt": "2024-12-15T10:30:00Z"
  }
]
```

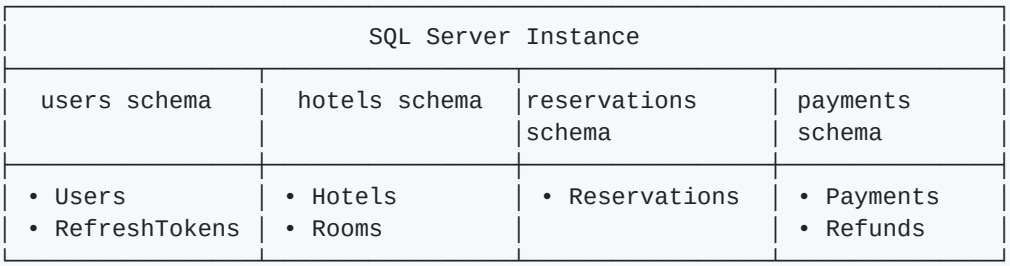
Database Schema

Overview

Attribute	Value
Database	Microsoft SQL Server
ORM	Entity Framework Core 9
Approach	Code-First with Migrations
Schema Separation	Each microservice has its own schema

The Hotel Booking Platform uses a **database-per-service** pattern where each microservice manages its own database schema. This ensures loose coupling and independent deployability.

Schema Overview



Users Service Schema

Table: users.Users

Stores user account information.

Column	Type	Constraints	Description
Id	UNIQUEIDENTIFIER	PK, NOT NULL, DEFAULT NEWID()	Primary key (GUID)
Email	NVARCHAR(255)	NOT NULL, UNIQUE	User's email address
PasswordHash	NVARCHAR(255)	NOT NULL	PBKDF2 hashed password
Role	NVARCHAR(50)	NOT NULL, DEFAULT 'User'	"User", "HotelOwner", or "Admin"
IsDeleted	BIT	NOT NULL, DEFAULT 0	Soft delete flag
DeletedAt	DATETIMEOFFSET	NULL	Deletion timestamp
CreatedAt	DATETIMEOFFSET	NOT NULL, DEFAULT SYSUTCDATETIME()	Account creation timestamp

Indexes:

- `UQ_Users_Email` UNIQUE on `Email`
- `IX_Users_Role` on `Role`

Table: users.RefreshTokens

Stores active refresh tokens for JWT authentication.

Column	Type	Constraints	Description
Id	UNIQUEIDENTIFIER	PK, NOT NULL, DEFAULT NEWID()	Primary key (GUID)
UserId	UNIQUEIDENTIFIER	FK → Users.Id, NOT NULL	Associated user
Token	NVARCHAR(500)	NOT NULL, UNIQUE	Refresh token value
ExpiresAt	DATETIMEOFFSET	NOT NULL	Token expiration time
IsRevoked	BIT	NOT NULL, DEFAULT 0	Revocation flag
CreatedAt	DATETIMEOFFSET	NOT NULL, DEFAULT SYSUTCDATETIME()	Token creation time
RevokedAt	DATETIMEOFFSET	NULL	When token was revoked
ReplacedByToken	NVARCHAR(500)	NULL	Token that replaced this one

Indexes:

- `UQ_RefreshTokens_Token` UNIQUE on `Token`
- `IX_RefreshTokens_UserId` on `UserId`

Hotels Service Schema

Table: hotels.Hotels

Stores hotel property information.

Column	Type	Constraints	Description
Id	UNIQUEIDENTIFIER	PK, NOT NULL, DEFAULT NEWID()	Primary key (GUID)
OwnerId	UNIQUEIDENTIFIER	NOT NULL	Hotel owner's user ID (logical FK)
Name	NVARCHAR(255)	NOT NULL	Hotel name
Description	NVARCHAR(MAX)	NULL	Hotel description
Country	NVARCHAR(100)	NOT NULL	Country
City	NVARCHAR(120)	NOT NULL	City

Column	Type	Constraints	Description
District	NVARCHAR(120)	NULL	District/area
AddressLine	NVARCHAR(300)	NULL	Street address
PetsAllowed	BIT	NOT NULL, DEFAULT 0	Pets allowed flag
IsPetHotel	BIT	NOT NULL, DEFAULT 0	Specialized pet hotel
CancelFreeDaysBefore	INT	NOT NULL, DEFAULT 0, CHECK >= 0	Free cancellation window
Approval	NVARCHAR(20)	NOT NULL, DEFAULT 'Pending'	Pending/Approved/Rejected
SubmittedAt	DATETIMEOFFSET	NOT NULL, DEFAULT SYSUTCDATETIME()	Submission timestamp
ReviewedAt	DATETIMEOFFSET	NULL	Admin review timestamp

Indexes:

- `IX_Hotels_OwnerId` on `OwnerId`
- `IX_Hotels_Country_City` on `Country, City`
- `IX_Hotels_Approval` on `Approval`

Table: hotels.Rooms

Stores individual room information for hotels.

Column	Type	Constraints	Description
Id	UNIQUEIDENTIFIER	PK, NOT NULL, DEFAULT NEWID()	Primary key (GUID)
HotelId	UNIQUEIDENTIFIER	FK → Hotels.Id, NOT NULL	Parent hotel
RoomNumber	NVARCHAR(50)	NOT NULL	Room identifier
Description	NVARCHAR(MAX)	NULL	Room description
Capacity	INT	NOT NULL, CHECK > 0	Maximum occupancy
Bedrooms	INT	NOT NULL, CHECK >= 0	Number of bedrooms
PricePerNight	DECIMAL(18,2)	NOT NULL, CHECK > 0	Nightly rate
Visible	BIT	NOT NULL, DEFAULT 1	Availability flag

Column	Type	Constraints	Description
PetsAllowed	BIT	NOT NULL, DEFAULT 0	Pet friendly flag
Accommodation	NVARCHAR(50)	NOT NULL	Standard/Deluxe/Suite
CreatedAt	DATETIMEOFFSET	NOT NULL, DEFAULT SYSUTCDATETIME()	Creation timestamp

Indexes:

- `IX_Rooms_HotelId` on `HotelId`
- `UQ_Rooms_HotelId_RoomNumber` UNIQUE on `(HotelId, RoomNumber)`

Reservations Service Schema

Table: reservations.Reservations

Stores booking/reservation information.

Column	Type	Constraints	Description
Id	UNIQUEIDENTIFIER	PK, NOT NULL, DEFAULT NEWID()	Primary key (GUID)
UserId	UNIQUEIDENTIFIER	NOT NULL	Guest's user ID (logical FK)
RoomId	UNIQUEIDENTIFIER	NOT NULL	Booked room ID (logical FK)
StartDate	DATE	NOT NULL	Check-in date
EndDate	DATE	NOT NULL, CHECK > StartDate	Check-out date
NumberOfGuests	INT	NOT NULL, CHECK > 0	Number of guests
Status	NVARCHAR(30)	NOT NULL, DEFAULT 'Active'	Booking status
CreatedAt	DATETIMEOFFSET	NOT NULL, DEFAULT SYSUTCDATETIME()	Booking creation time
CancelledAt	DATETIMEOFFSET	NULL	Cancellation timestamp
ConfirmedAt	DATETIMEOFFSET	NULL	Confirmation timestamp

Status Values:

- `Active` - Booking confirmed and active
- `Cancelled` - Cancelled by user/admin
- `RefundPending` - Awaiting refund
- `Completed` - Stay completed

Indexes:

- `IX_Reservations_UserId` on `UserId`
- `IX_Reservations_RoomId` on `RoomId`
- `IX_Reservations_Status` on `Status`
- `IX_Reservations_StartDate_EndDate` on `(StartDate, EndDate)`

Payments Service Schema

Table: payments.Payments

Stores payment transaction information.

Column	Type	Constraints	Description
Id	UNIQUEIDENTIFIER	PK, NOT NULL, DEFAULT NEWID()	Primary key (GUID)
ReservationId	UNIQUEIDENTIFIER	NOT NULL, UNIQUE	Associated reservation (logical FK)
Amount	DECIMAL(18,2)	NOT NULL, CHECK > 0	Payment amount
Currency	NVARCHAR(10)	NOT NULL, DEFAULT 'USD'	Currency code
Status	NVARCHAR(30)	NOT NULL, DEFAULT 'RequiresPayment'	Payment status
StripePaymentIntentId	NVARCHAR(500)	NULL	Stripe PI identifier
CreatedAt	DATETIMEOFFSET	NOT NULL, DEFAULT SYSUTCDATETIME()	Payment initiated
PaidAt	DATETIMEOFFSET	NULL	Payment completed

Status Values:

- `RequiresPayment` - Intent created, awaiting payment
- `Processing` - Payment in progress
- `Succeeded` - Payment completed

- `Failed` - Payment failed
- `Refunded` - Payment refunded

Indexes:

- `UQ_Payments_ReservationId` UNIQUE on `ReservationId`
- `IX_Payments_Status` on `Status`
- `IX_Payments_StripePaymentIntentId` on `StripePaymentIntentId`

Table: payments.Refunds

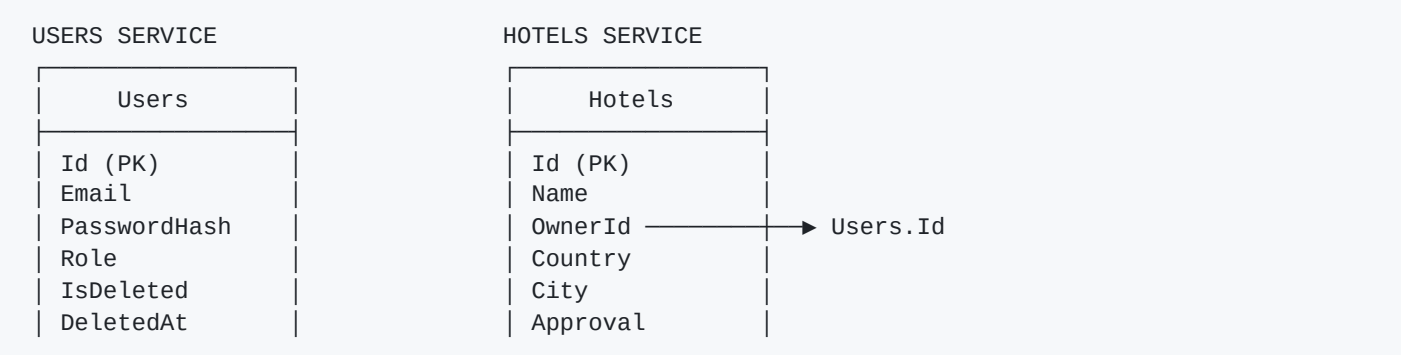
Stores refund transaction information.

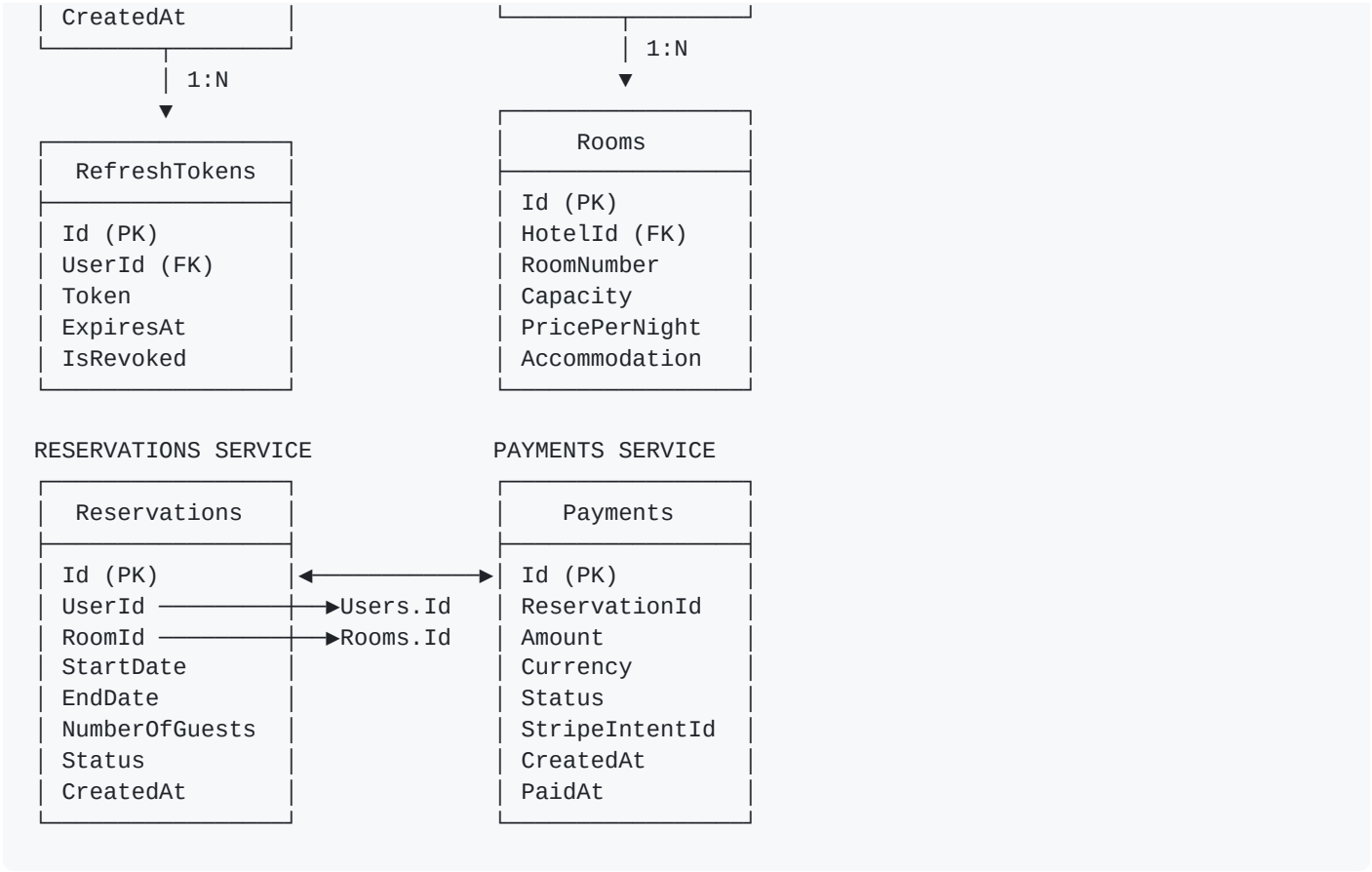
Column	Type	Constraints	Description
Id	UNIQUEIDENTIFIER	PK	Primary key (GUID)
PaymentId	UNIQUEIDENTIFIER	FK → Payments.Id, NOT NULL	Original payment
Amount	DECIMAL(18,2)	NOT NULL	Refund amount
Reason	NVARCHAR(500)	NULL	Refund reason
Status	NVARCHAR(50)	NOT NULL	Refund status
StripeRefundId	NVARCHAR(500)	NULL	Stripe refund identifier
CreatedAt	DATETIME2	NOT NULL	Refund initiated
CompletedAt	DATETIME2	NULL	Refund completed

Indexes:

- `IX_Refunds_PaymentId` on `PaymentId`

Entity Relationship Diagram





Note: Cross-service references are by ID only, not foreign keys. Each service maintains data consistency through API calls, not database constraints.

Migrations

Entity Framework Core manages schema migrations for each service.

Users Service Migrations

Migration	Description	Date
InitialCreate	Users and RefreshTokens tables	2025-11-15
AddUserRole	Added Role column	2025-11-20
AddRefreshTokenRevocation	Added RevokedAt column	2025-12-01

Hotels Service Migrations

Migration	Description	Date
InitialCreate	Hotels and Rooms tables	2025-11-15

Migration	Description	Date
AddHotelApproval	Added IsApproved flag	2025-11-22
AddCancellationPolicy	Added CancelFreeDaysBefore	2025-12-05

Reservations Service Migrations

Migration	Description	Date
InitialCreate	Reservations table	2025-11-18
AddCancellationFields	Added cancellation tracking	2025-12-10
AddPaymentReference	Added PaymentId	2025-12-15

Payments Service Migrations

Migration	Description	Date
InitialCreate	Payments table	2025-11-20
AddStripeFields	Added Stripe integration fields	2025-12-01
AddRefunds	Added Refunds table	2025-12-20

Running Migrations

Apply migrations for each service:

```
# Users Service
cd hotel-booking-users-service/Api
dotnet ef database update --project ../Infrastructure

# Hotels Service
cd hotel-booking-hotels-service/Api
dotnet ef database update --project ../Infrastructure

# Reservations Service
cd hotel-booking-reservations-service/Api
dotnet ef database update --project ../Infrastructure

# Payments Service
cd hotel-booking-payments-service/Api
dotnet ef database update --project ../Infrastructure
```

Test Data Seeding

Each service can seed test data via its `DbContext`:

```
// In Program.cs or migration
if (app.Environment.IsDevelopment())
{
    using var scope = app.Services.CreateScope();
    var context = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    await context.SeedTestDataAsync();
}
```

Glossary

General Terms

Term	Definition
Access Token	Short-lived JWT token (60 minutes) used to authenticate API requests
Refresh Token	Long-lived token (7 days) used to obtain new access tokens without re-login
Microservice	An independently deployable service responsible for a specific business capability
Clean Architecture	Software design pattern separating code into layers (Domain, Application, Infrastructure, API)
Code-First	EF Core approach where database schema is generated from C# entity classes
Webhook	HTTP callback triggered by external service (Stripe) to notify of events
Payment Intent	Stripe object representing a payment lifecycle from creation to completion

Acronyms

Acronym	Full Form	Description
API	Application Programming Interface	Contract defining how services communicate
JWT	JSON Web Token	Compact, URL-safe token for secure claims transmission

Acronym	Full Form	Description
EF	Entity Framework	Microsoft's ORM for .NET applications
ORM	Object-Relational Mapping	Technique mapping objects to database tables
GUID	Globally Unique Identifier	128-bit identifier used as primary keys
CORS	Cross-Origin Resource Sharing	Security feature controlling cross-domain requests
CI/CD	Continuous Integration/Continuous Deployment	Automated build and deployment pipeline
SPA	Single Page Application	Web app loading a single HTML page dynamically
REST	Representational State Transfer	Architectural style for web services
DTO	Data Transfer Object	Object carrying data between processes
DI	Dependency Injection	Design pattern for providing dependencies

Domain-Specific Terms

Hotel & Booking Domain

Term	Definition
Hotel	A property offering rooms for accommodation
Room	An individual bookable unit within a hotel
Room Type	Category of room (Standard, Deluxe, Suite, etc.)
Reservation	A booking request for a room during specific dates
Check-in Date	The date a guest arrives at the hotel
Check-out Date	The date a guest departs from the hotel
Guest Count	Number of people staying in the room
Hotel Owner	User role that can create and manage hotels
Free Cancellation Period	Days before check-in when cancellation incurs no charge

Reservation Status

Status	Definition
Pending	Reservation created, awaiting payment
Confirmed	Payment received, booking secured
CancellationPending	Cancellation requested, awaiting admin approval
Cancelled	Reservation successfully cancelled
Completed	Guest stay finished
NoShow	Guest did not arrive for reservation

Payment Terms

Term	Definition
Payment Intent	Stripe object tracking payment from creation to completion
Client Secret	Token used by frontend to complete payment securely
Refund	Return of payment amount to customer
Stripe	Third-party payment processing platform
Test Card	Stripe test card number (4242...) for sandbox testing

User Roles

Role	Definition
User	Standard customer who can search hotels and make reservations
HotelOwner	Business user who can list hotels and manage rooms
Admin	System administrator with full access to all operations

Architecture Terms

Term	Definition
Domain Layer	Core business logic and entities, no external dependencies
Application Layer	Use cases, commands, queries, and service interfaces
Infrastructure Layer	External concerns: database, HTTP clients, third-party services

Term	Definition
API Layer	HTTP endpoints, controllers, and request/response handling
Handler	Class implementing a specific command or query operation
Command	Object representing an action that modifies state
Query	Object representing a request for data

Azure Services

Term	Definition
Azure App Service	PaaS for hosting web applications and APIs
Azure SQL	Managed SQL Server database service
Azure Key Vault	Secure storage for secrets, keys, and certificates
Free Tier (F1)	Azure's no-cost hosting plan with limited resources
Cold Start	Delay when app starts after period of inactivity on free tier

Testing Terms

Term	Definition
Unit Test	Test verifying individual component in isolation
Integration Test	Test verifying multiple components working together
Mock	Simulated object replacing real dependency in tests
Code Coverage	Percentage of code executed by tests
xUnit	.NET testing framework used in this project
FluentAssertions	Library for readable test assertions