**StayMate – Project Report**

**Designing Modern Software Systems**

**Practice Module Submission**

**Graduate Certificate in Designing Modern Software Systems** National University of Singapore – Institute of Systems Science (NUS-ISS)

**Project Title**

**StayMate: A Modern Hotel Booking Platform**

**Team 5 – Project Members**

* Phillip Chen
* Jonathan Lim
* Lee Jun Heng
* Ng Ju Khang
* Ng Zhi Kai

**Supervisor**

Chandrasekaran Paramasivam  
Institute of Systems Science – NUS

**Date of Submission**

May 2025

**Table of Contents**

[1. Introduction 7](#_heading=h.7786eed0qby8)

[1.1 Background 7](#_heading=h.kt4wg4xamklq)

[1.2 Business Needs 7](#_heading=h.h3iibykp48dn)

[1.3 Stakeholders 8](#_heading=h.8rk07toh6ed0)

[1.4 Project Scope 8](#_heading=h.ofnwdpgtt61i)

[1.4.1 Functionality in Scope 8](#_heading=h.j8tgriiviu7v)

[1.4.2 Functionality out of Scope 9](#_heading=h.ul88bm7mkhl3)

[1.4.3 Quality Attributes 10](#_heading=h.wmiwk68kl9ex)

[2. Project Conduct 11](#_heading=h.fmtokxb4tyiw)

[2.1 Project Plan 11](#_heading=h.xnucft1lj1g2)

[Work Breakdown Structure 11](#_heading=h.7tv9wjiqw1by)

[Use Case Story 13](#_heading=h.uf53ps4jhyrr)

[2.2 Project Status 14](#_heading=h.dmiv3ox3zylx)

[2.2.1 Deployment Overview 14](#_heading=h.4nqqu57wj548)

[2.2.2 Outstanding Issues 15](#_heading=h.cgao6ks7yio6)

[2.3 Project Metrics 16](#_heading=h.9djk4x8jevap)

[Project Milestones Achieved 16](#_heading=h.kq775io6ytbh)

[Effort Expended per Team Member (Estimated Man-Hours) 17](#_heading=h.jnsugparupp7)

[3. System Design 18](#_heading=h.z8is5se4m1la)

[3.1 Software Architecture Overview 18](#_heading=h.3cm7famkrbax)

[3.1.1 External Integrations 18](#_heading=h.1tteuoi4pbq8)

[3.2 Architectural Layers and Components 18](#_heading=h.by9fgjzlx1p)

[3.2.1 Presentation Layer (Frontend) 18](#_heading=h.jhzxfuk7gro1)

[3.2.2 Business Logic Layer (Backend) 19](#_heading=h.7v5lo4z4ihge)

[3.2.3 Data Access Layer 19](#_heading=h.d78nsltagmkw)

[3.3 Component Overview 19](#_heading=h.97b4dfio1ene)

[3.3.1 Frontend (React App) 19](#_heading=h.9gg93c7gavzg)

[3.3.2 Backend (Spring Boot Services) 19](#_heading=h.b09m3iwtrf9m)

[3.3.3 MySQL Database 20](#_heading=h.yil7p4iihrsl)

[3.3.4 Mock Payment Gateway 20](#_heading=h.6nqljwhzll9w)

[3.3.5 SMTP/Email Service 20](#_heading=h.we33luls7r5u)

[3.4 Technology Stack 20](#_heading=h.3xlsex55mxg3)

[3.5 Transition from Analysis to Design 21](#_heading=h.qh92yxn6nkui)

[3.5.1 Transition Strategy 1: Use Case Realization through Sequence Diagrams 22](#_heading=h.b9f9jzhv7q5)

[3.5.2 Transition Strategy 2: Domain Model Mapping 22](#_heading=h.2qep8o1f2rq)

[3.5.3 Transition Strategy 3: Layered Architectural Alignment 23](#_heading=h.4yu6s7v9hg99)

[3.5.4 Transition Strategy 4: Controller-Service-Repository Pattern 24](#_heading=h.awxtomcku1ix)

[3.5.5 Transition Strategy 5: Design Pattern Application 24](#_heading=h.gmxypyuxab8m)

[3.5.6 Transition Strategy 6: API Contract Design 25](#_heading=h.qir4bdzi6suz)

[3.6 Behavioral and Structural Diagrams 26](#_heading=h.7x6wjukhmf3t)

[3.6.1 UML Class Diagram 26](#_heading=h.7fwzppbcn3nl)

[3.6.2 Use Case Diagram 27](#_heading=h.d0zgx0gmbvro)

[3.6.3 Use Case Description 1: Login 28](#_heading=h.oypmvhm7pc1y)

[Flow of Events 28](#_heading=h.3ly9eecjsx22)

[● Normal Flow 28](#_heading=h.er0m2awx0cu3)

[● Exceptional Flow 28](#_heading=h.gw9bvw2av51n)

[3.6.4 Use Case Description 2: Browse Hotels 29](#_heading=h.vyke5h7mg2e7)

[Flow of Events 29](#_heading=h.4ujfp23ggo9f)

[● Normal Flow 29](#_heading=h.7g9q4t4jsslz)

[● Exceptional Flow 29](#_heading=h.sq6u2dssh2pd)

[3.6.5 Use Case Description 3: Search Hotels 30](#_heading=h.f1byntsswmgj)

[Flow of Events 30](#_heading=h.78f803tu1ttg)

[● Normal Flow 30](#_heading=h.u3pfoj8aawgt)

[● Exceptional Flow 30](#_heading=h.rub4cjg2tuts)

[3.6.6 Use Case Description 4: Bookmark Hotel 31](#_heading=h.d6f5umig388g)

[Flow of Events 31](#_heading=h.4smkyu44ni52)

[● Normal Flow 31](#_heading=h.qpc4mfj5y4ez)

[● Exceptional Flow 31](#_heading=h.r3wowhtpmi37)

[3.6.7 Use Case Description 5: Update Profile 32](#_heading=h.qgwal77l9l77)

[Flow of Events 32](#_heading=h.nxdqtkg598ag)

[● Normal Flow 32](#_heading=h.kz8ic78zbs44)

[● Exceptional Flow 32](#_heading=h.6fgf35tduvcv)

[3.6.8 Use Case Description 6: Book Hotel 33](#_heading=h.i472q7a0k433)

[Flow of Events 33](#_heading=h.eoe9vahvfvw)

[● Normal Flow 33](#_heading=h.b35kgjy83f0)

[● Exceptional Flow 33](#_heading=h.nf7ovxpckpcj)

[3.6.9 Use Case Description 7: Payment Service (Included by Book Hotel) 34](#_heading=h.i8mz2ni11yxt)

[Flow of Events 34](#_heading=h.mbrc7ncdas7q)

[● Normal Flow 34](#_heading=h.37sneei9ryv)

[● Exceptional Flow 34](#_heading=h.1rtr403f37gt)

[3.6.10 Use Case Description 8: Manage Accounts 35](#_heading=h.xiiy3trvdc35)

[Flow of Events 35](#_heading=h.4m0tyiy45bcq)

[● Normal Flow 35](#_heading=h.wnad9j6lhggl)

[● Exceptional Flow 35](#_heading=h.kanh72lcutca)

[3.6.11 Use Case Description 9: Manage Hotel 36](#_heading=h.c4va0gy07hnf)

[Flow of Events 36](#_heading=h.ebtw3p3qqsev)

[● Normal Flow 36](#_heading=h.mi2fk7pnd30w)

[● Exceptional Flow 36](#_heading=h.gth5vfmlw0dv)

[3.6.12 Use Case Description 10: Manage Promotion Offers 37](#_heading=h.ivg49jaoyydo)

[Flow of Events 37](#_heading=h.ibfo6uoqdv47)

[● Normal Flow 37](#_heading=h.xug9ctkyplct)

[● Exceptional Flow 37](#_heading=h.6h0teaw37u1h)

[3.6.13 Use Case Description 11: Manage Payments 38](#_heading=h.x2es89pm8fky)

[Flow of Events 38](#_heading=h.qbuewzyjuizx)

[● Normal Flow 38](#_heading=h.5fda7vmxnxky)

[● Exceptional Flow 38](#_heading=h.c4jrsako5le)

[3.7 Sequence Diagrams 39](#_heading=h.78j1mcqnatn6)

[3.7.1 Use Case 1: Login by Ng Ju Khang 39](#_heading=h.ckdwtdyavh94)

[3.7.2 Use Case 2: Browse Hotels by Ng Ju Khang 39](#_heading=h.4jlttqbuydxj)

[3.7.3 Use Case 3: Search for Hotels by Ng Ju Khang 40](#_heading=h.8tnls39cizyv)

[3.7.4 Use Case 4: Bookmark Hotel by Ng Ju Khang 40](#_heading=h.u8a91ixs3h31)

[3.7.5 Use Case 5: Update Profile by Ng Zhi Kai 41](#_heading=h.t7ofmt5f4gnj)

[3.7.6 Use Case 6: Book Hotel by Ng Zhi Kai 41](#_heading=h.4qqy7qhidsb7)

[3.7.7 Use Case 7: Make Payment by Ng Zhi Kai 41](#_heading=h.poo8x24n3v85)

[3.7.8 Use Case 8: Manage Users by Ng Zhi Kai 42](#_heading=h.ww9m2sioy200)

[View All Users 42](#_heading=h.zf317d8no53)

[Register Users 43](#_heading=h.9n08i9aflwdh)

[Edit Users 43](#_heading=h.4i7rsqjkijc)

[Delete Users 43](#_heading=h.jbiq39pr6pze)

[3.7.9 Use Case 9: Manage Hotel by Ng Zhi Kai 44](#_heading=h.u82hc35mnzf7)

[View All Hotels 44](#_heading=h.b8jjr4xoifl9)

[Create Hotels 44](#_heading=h.a3kw6tjc88v4)

[Delete Hotels 44](#_heading=h.rmz2n3711t2b)

[Edit Hotels 45](#_heading=h.573md1bid9sr)

[3.7.10 Use Case Description 10: Manage Promotion Offers by Ng Zhi Kai 45](#_heading=h.h9pavqm6cm0b)

[3.7.11 Use Case Description 11: Manage Payments by Ng Zhi Kai 46](#_heading=h.r4xmrzqqu5bt)

[Find Payments by User 46](#_heading=h.bcmr8cvsoejz)

[Find Payments by Booking 46](#_heading=h.8kfa5wpwb42j)

[3.8 Design Problems and Patterns 47](#_heading=h.ruao69t4hmkh)

[3.8.1 Design Problem for Payment Service by Ng Zhi Kai 47](#_heading=h.u9iq4oei9hs)

[Initial Implementation 47](#_heading=h.hxfn4c6imvio)

[Candidate Design Pattern(s) 47](#_heading=h.lqt01cod7mxi)

[Motivation for Strategy Pattern 47](#_heading=h.uira31stz6zz)

[Sequence Flow 47](#_heading=h.15v03gobn1cc)

[3.8.2 Design Problem for Create Notification by Ng Zhi Kai 48](#_heading=h.ygdpgzase2t7)

[Initial Implementation 48](#_heading=h.zeyjvo6xc13b)

[Candidate Design Pattern(s) 49](#_heading=h.kn8wxy3t9ha)

[Motivation for Observer Pattern 49](#_heading=h.u3a1ouqmz626)

[Sequence Diagram Flow 49](#_heading=h.n2zixkc6pkpn)

[3.8.3 Design Problem for Create Room by Ng Zhi Kai 51](#_heading=h.bqctn1hpm7zx)

[Initial Implementation 51](#_heading=h.segzve9be5tp)

[Candidate Design Pattern(s) 51](#_heading=h.jenlniyx56dc)

[Motivation for Factory Pattern 52](#_heading=h.voqhcw92eonr)

[Sequence Diagram Flow 52](#_heading=h.1wbp72xlhkhz)

[3.8.4 Design Problem for Room States by Ng Zhi Kai 53](#_heading=h.lp6s6h8j4le7)

[Initial Implementation 53](#_heading=h.tjaoeejuj52l)

[Candidate Design Pattern(s) 54](#_heading=h.2nd4fenwtmoh)

[Motivation for State Pattern 54](#_heading=h.csoywi70xgak)

[Sequence Diagram Flow 54](#_heading=h.tqml2te9bera)

[3.9 Database Schemas 58](#_heading=h.l9xjhf3cq42f)

[3.9.1 Overview 58](#_heading=h.9dn8nms2j2cp)

[3.9.2 Users 58](#_heading=h.h71ukgwrli57)

[3.9.3 Hotels 58](#_heading=h.3raf1idwolj4)

[3.9.4 Room 59](#_heading=h.yr5e7rcmlukv)

[3.9.5 Booking 59](#_heading=h.43ym1kdp7byl)

[3.9.6 Reviews 59](#_heading=h.44nn3vugjtdi)

[3.9.7 Payments 60](#_heading=h.mxsiddqo7au8)

[3.9.8 Payments 60](#_heading=h.o5vimfv1pdlt)

[3.9.9 Entity Relationships 61](#_heading=h.m72u07k4b7pv)

[4. DevSecOps and Development Lifecycle 62](#_heading=h.i1di5ibetp1o)

[4.1 Source Control Strategy 62](#_heading=h.8mixs9qhidyi)

[4.2 Repository and Project Structure 62](#_heading=h.g5w3uxr0fp7f)

[4.3 Branching Strategy 63](#_heading=h.5e6f5vkd285)

[4.4 Authentication and Access Control 64](#_heading=h.fr4rl5gtscqc)

[4.5 Continuous Integration (CI) 65](#_heading=h.tq1c1xaxa89v)

[4.5.1 Developer Workflow 65](#_heading=h.frvjbmdqlsjz)

[4.5.2 Pipeline Jobs 65](#_heading=h.ewuiqedrhtt1)

[4.6 Continuous Delivery (CD) 66](#_heading=h.p3yvvpcygoyc)

[4.6.1 Deployment Workflow 66](#_heading=h.tnd91kfi8cv)

[4.6.2 Environment 68](#_heading=h.wje83bun0iuz)

[5. Performance Testing 69](#_heading=h.jwa1lu116l14)

[5.1 Target Metrics 69](#_heading=h.o9nfkj7v58iu)

[5.1.1 Infrastructure Considerations 69](#_heading=h.aaxa0aa7wbnq)

[5.2 Business Processes 70](#_heading=h.7et19534e7fl)

[5.3 Load Testing 70](#_heading=h.kdit2ur0z9gj)

[5.4 Load Testing (Results) 71](#_heading=h.pj6w4paqxw44)

[Summary 71](#_heading=h.n6m6du6mt915)

[Statistics 71](#_heading=h.eiup9d6m7l21)

[Errors 72](#_heading=h.30zuxx79ne7)

[Response Time over Time Graph 73](#_heading=h.761msvi9lrj6)

[6. Conclusion 73](#_heading=h.2slzu48vvo3l)

[Appendix A: Static Application Security Testing (SAST) Results 75](#_heading=h.aoldndgjn91k)

[Appendix B: Dynamic Application Security Testing (DAST) – OWASP ZAP Results 77](#_heading=h.1bh00w89scbz)

[Appendix C: Linting Report 79](#_heading=h.59znrncr91dr)

[Appendix D: JaCoCo Code Coverage Report 80](#_heading=h.x0h8wgtpvzqs)

[Appendix E: Team Members Contributions 81](#_heading=h.heo5xqtu7533)

[Work Breakdown Structure 81](#_heading=h.grcj7eg0csha)

[Appendix F: Progress Reports 83](#_heading=h.l647ff867xbj)

[Appendix G: Links 83](#_heading=h.swmakqcmk05l)

# Introduction

## Background

**StayMate** is a user-centric hotel booking platform designed to meet the evolving needs of business travelers, tourists, and local users. It offers a seamless booking experience through an intuitive interface, real-time hotel availability, transparent pricing, and secure online payments.

To address key market pain points, the platform focuses on simplicity, responsiveness, and reliability. Throughout the development lifecycle, the team adopted the **Agile Scrum methodology**, which enabled iterative progress, rapid feedback, and adaptability to changing requirements.

Over multiple sprints, the team delivered essential backend modules—including user management, hotel and room booking systems, and a mock payment gateway—while simultaneously developing a responsive frontend for hotel search, filtering, and booking confirmation.

The system architecture emphasizes **scalability** and **maintainability**, with a clear separation of concerns across the frontend, backend, and database layers. Geolocation integration further enriches the user experience. The result is a robust, production-ready platform capable of supporting diverse hotel booking scenarios with efficiency and reliability.

## Business Needs

StayMate was developed to meet the demand for a seamless, user-first hotel booking experience. The core business goals include:

* **Efficiency**: Real-time hotel availability and streamlined booking workflows
* **User Experience**: Easy search, filter, and booking through a responsive, intuitive interface
* **Market Differentiation**: Transparent pricing, location-based search, and secure payments
* **Scalability**: Architecture designed to support growth in both users and hotel listings

These goals aim to drive user satisfaction, increase booking volume, and position StayMate as a competitive player in the travel tech market.

## Stakeholders

* **Product Owner** Defines the product vision, manages priorities, and ensures the platform aligns with user and business needs.
* **Project Manager** Oversees sprint planning, team coordination, and ensures timely delivery of project milestones.
* **Development Team** Builds backend services, APIs, and frontend components in alignment with user stories and acceptance criteria.
* **UI/UX Designer** Designs a consistent and user-friendly interface to enhance the overall user journey.
* **QA Engineers** Conduct functionality, integration, and stability testing to ensure product reliability and performance.

## Project Scope

The StayMate project involved the end-to-end development of a **web-based hotel booking platform**, encompassing the entire software development lifecycle—from **requirements analysis** and **system design** to **implementation**, **testing**, and **deployment**.

### Functionality in Scope

* **Responsive Frontend (React)** Enables hotel browsing, user registration, booking, and account management through a clean, mobile-friendly interface.
* **Comprehensive Backend Services** Includes user authentication, hotel and room management, booking processing, and mock payment gateway integration.
* **Relational Database Design (MySQL)** Supports core entities such as users, hotels, rooms, bookings, payments, reviews, and notifications with normalized schemas.
* **RESTful APIs** Facilitate efficient and scalable communication between the frontend and backend layers.
* **Design Pattern Integration** Implements modular backend logic using patterns like **Factory** (for room creation) and **State** (for room status management).
* **Geolocation API Integration** Supports location-based hotel searches to enhance user relevance and convenience.
* **CI/CD Pipeline Setup** Automated testing, building, and deployment using continuous integration and delivery tools.
* **Testing Strategy** Covers both **unit** and **integration** testing for backend services and frontend components to ensure system stability.

### Functionality out of Scope

* Native mobile applications (iOS and Android)
* Integration with live third-party hotel inventory providers
* Advanced pricing features such as dynamic pricing algorithms
* Real-time analytics dashboards and performance monitoring
* Multilingual support and advanced accessibility features
* Enhanced security measures like two-factor authentication and audit logging
* Deep security hardening (e.g., SSL enforcement, rate limiting, intrusion detection)

### Quality Attributes

* **Performance** Optimized to handle up to **100 concurrent users**, with response times under **3000 ms** during load testing and login operations under **5000 ms**.
* **Availability** Designed for **99.5% uptime**, with seamless deployments enabled by automated CI/CD pipelines.
* **Scalability** While currently monolithic, the system architecture supports future transition to **microservices** to accommodate growth.
* **Security** Includes baseline protections such as **input validation**, **session management**, and safeguards against common vulnerabilities (e.g., SQL injection, XSS).
* **Maintainability** Built using **clean code principles** and documented **design patterns** to support ongoing enhancements and ease of maintenance.
* **Usability** Offers a modern, intuitive interface for both guests and administrators, ensuring an efficient and satisfying user experience.
* **Testability** Core features are covered by **automated tests**, facilitating continuous delivery and early defect detection.
* **Deployability** CI/CD automation supports **rapid, low-risk deployments** with minimal manual intervention.

# Project Conduct

## Project Plan

### **Work Breakdown Structure**

| **Phase** | **Task** | **Estimated Effort (man-days)** | **Done By** |
| --- | --- | --- | --- |
| **Project Planning and Initiation**  **(1 week)** | Define project scope and objectives | 1 | All |
|  | Set up project tools and communication channels | 1 | All |
|  | Develop project schedule and resource allocation | 1 | All |
| **Requirements Gathering and Analysis (1 week)** | Document functional and non-functional requirements | 2 | Ju Khang |
|  | Define user stories and acceptance criteria | 2 | Ju Khang |
|  | Finalize project requirements | 2 | Ju Khang |
| **System Architecture and Design (1 week)** | Define system architecture design | 2 | All |
|  | Create detailed technical design | 3 | Ju Khang, Zhi Kai |
|  | Design database schema and data flow | 2 | Zhi Kai |
| **Frontend Development**  **(2 weeks)** | Set up frontend environment and tools | 2 | Jun Heng, Ju Khang |
|  | Develop core UI components | 10 | Ju Khang |
|  | Integrate frontend with backend APIs | 8 | Zhi Kai |
|  | Conduct frontend unit and integration testing | 6 | Ju Khang, Zhi Kai |
| **Backend Development (2 weeks)** | Set up backend environment and tools | 2 | Zhi Kai |
|  | Develop core backend functionality | 10 | Zhi Kai |
|  | Implement API endpoints | 8 | Zhi Kai |
|  | Integrate third-party services | 6 | Zhi Kai |
|  | Implement business logic and data validation | 6 | Zhi Kai  Jonathan |
| **Testing and Quality Assurance (2 weeks)** | Develop test cases for all requirements | 3 | Ju Khang, Zhi Kai |
|  | Conduct manual testing | 8 | Ju Khang, Zhi Kai |
|  | Automate tests | 10 | Ju Khang, Zhi Kai |
|  | Perform load and performance testing | 6 | Ju Khang |
|  | Bug fixing and validation after testing | 6 | Zhi Kai Jonathan |
| **DevOps and Deployment**  **(1 week)** | Set up CI/CD pipeline | 5 | Zhi Kai |
|  | Deploy application to production | 3 | Zhi Kai |

### 

### Use Case Story

| **Use Case** | **Team Member(s)** | **Design Problem Identified** | **Design Pattern Used** |
| --- | --- | --- | --- |
| Login | Ng Zhi Kai | Need to handle authentication logic for multiple user roles | Strategy Pattern for flexible login flows |
| Browse Hotels | Ng Zhi Kai | Rendering dynamic hotel listings efficiently | MVC Pattern for UI separation |
| Search Hotels | Ng JuKhang | Allowing flexible and multi-criteria hotel search | Specification Pattern for filter logic |
| Save Hotel | Lee Jun Heng | Managing user's saved hotels with proper database association | Repository Pattern |
| Update Profile | Ng Zhi Kai | Allowing real-time updates while enforcing validation | Observer Pattern for real-time UI updates |
| Book Hotel | Ng Zhi Kai | Managing booking state and consistency between room availability and booking confirmation | State Pattern for booking lifecycle |
| Payment Service | Ng Zhi Kai | Supporting different payment methods with unified interface | Strategy Pattern for gateway handling |
| Manage Accounts | Ng Zhi Kai | Role-based permissions and dynamic user management | Role-Based Access Control (RBAC) Pattern |
| Manage Hotels | Ng Zhi Kai | CRUD operations across multiple hotel records | MVC + Factory Pattern for scalability |
| Manage Promotion Offers | Ng Zhi Kai | Associating promotions with target hotels and enforcing activation conditions | Decorator Pattern (optional, if dynamic pricing) |
| Generate Report | Ng JuKhang | Formatting reports dynamically with different data views | Builder or Template Method Pattern |
| Manage Payments | Ng JuKhang | Managing payment records without processing transactions | Repository + Factory Pattern |
| Email Verification | Ng Zhi Kai | Sending confirmation emails upon user registration | **Observer Pattern** to trigger email service upon registration |
| Notifications | Ng Zhi Kai | Informing users of events like bookings and payments | **Observer Pattern** to decouple event handling from services |

## Project Status

The **StayMate Hotel Booking Platform** has reached its initial **production-ready milestone**, with core features successfully implemented, tested, and deployed. These include:

* **User Registration and Authentication** Enables users to sign up, log in, and manage their accounts securely.
* **Hotel and Room Browsing** Provides real-time availability and detailed room information with support for search and filtering.
* **Booking Management** Allows users to create, view, and manage bookings with a streamlined workflow.
* **Mock Payment Integration** Simulates payment transactions to validate booking flow and system behavior.
* **Notifications** Automatically sends confirmation messages for key actions such as booking creation.

### Deployment Overview

* **Frontend** Deployed via **GitHub Pages**, featuring a responsive UI and real-time booking capabilities.
* **Backend** Hosted on an **AWS EC2 instance**, exposing RESTful APIs for all essential platform operations.
* **CI/CD Pipelines** Implemented for both frontend and backend, automating builds, testing, and deployments to accelerate delivery and reduce manual errors.

### Outstanding Issues

* **Error Handling** Some edge cases require more robust error handling to improve reliability and user experience.
* **Test Coverage** Additional testing is planned, particularly for **integration scenarios** and complex edge cases.
* **Security Enhancements** Future updates will address **rate limiting**, **SSL certificate integration**, and **secure secrets management** for production environments.
* **Mobile Optimization** While mobile responsiveness is functional, further **optimization and QA** across a broader range of devices is planned.

## Project Metrics

### Project Milestones Achieved

| **Milestone** | **Completion Date** | **Remarks** |
| --- | --- | --- |
| **Project Planning Completed** | Week 1 | Scope, schedule, and tools defined |
| **Requirements Finalized** | Week 2 | Functional and non-functional requirements set |
| **System Design Completed** | Week 3 | Architecture, DB schema, and data flows defined |
| **Frontend Core UI Completed** | Week 5 | UI components built with Tailwind & React |
| **Backend Core APIs Implemented** | Week 5 | User, booking, payment, and hotel services |
| **Integration of Frontend & Backend** | Week 6 | End-to-end flows connected via REST APIs |
| **Testing & Bug Fixing** | Week 7 | Manual and automated tests completed |
| **CI/CD and Deployment to EC2 & GitHub Pages** | Week 8 | Frontend live on GitHub Pages, backend on EC2 |

### 

### **Effort Expended per Team Member (Estimated Man-**Hour**s)**

| **Team Member** | **Total Effort (Man-Hours)** | **Key Contributions** |
| --- | --- | --- |
| **Zhi Kai** | 200 | Backend development, Frontend development, API integration, CI/CD setup, deployment, system design, testing |
| **Ju Khang** | 100 | Requirements gathering, Frontend development, System design, System Testing, Performance Testing |
| **Jonathan** | 80 | Enhancement of functionality with testing. Fixing of bugs. Frontend development. |
| **Jun Heng** | 100 | Setting up of project, frontend development, enhancement of functionality with testing, fixing of bugs. |
| **Phillip** | 80 | Integrating testing framework, system testing, writing unit testing for API. |

### 

# 

# System Design

## Software Architecture Overview

The **StayMate platform** follows a **layered and modular architecture** to support scalability, maintainability, and a clear separation of concerns. The system is organized into three main layers:

* **Presentation Layer (Frontend)** Built with **React**, it handles user interactions and communicates with backend services via REST APIs.
* **Business Logic Layer (Backend)** Developed using **Spring Boot**, it processes requests, enforces business rules, and connects the frontend with the database.
* **Data Access Layer (Database)** Uses **MySQL** to manage data related to users, hotels, rooms, bookings, and more.

### External Integrations

* **Mock Payment Gateway** for simulating transactions
* **Email Notification Service** for sending booking-related emails via REST APIs

## Architectural Layers and Components

### Presentation Layer (Frontend)

Developed using **React.js**, this layer manages the user interface, client-side routing, and dynamic component rendering. It focuses on **responsiveness**, **reusability**, and delivering a **user-friendly experience**.  
Users interact with this layer to:

* Browse and search for hotels
* Make and manage bookings
* Access and update their account/profile

### Business Logic Layer (Backend)

Built with Spring Boot, this layer handles the core application logic and orchestrates communication between the frontend, database, and external services. It follows a modular structure, with services organized by domain:

* **userService**: User registration, login, and profile management
* **hotelService**: Hotel and room management
* **bookingService**: Booking creation, modification, and cancellation
* **paymentService**: Transaction handling via mock payment gateway
* **notificationService**: Email notifications for bookings and system events

### Data Access Layer

This layer uses **Spring Data JPA** and **Hibernate ORM** to manage persistence with a **MySQL database**. Each entity (e.g., User, Hotel, Booking) has a corresponding repository interface for CRUD operations, enabling seamless mapping between Java objects and database tables.

### 

## Component Overview

## Frontend (React App)

Provides the interface for hotel search, booking, and user account management. Communicates with backend services via RESTful APIs.

### Backend (Spring Boot Services)

Encapsulates domain logic through modular services:

* **Authentication Service**: Manages login, registration, and session handling
* **Hotel Management Service**: Handles hotel creation and room availability
* **Booking Management Service**: Manages bookings and cancellations
* **Payment Service**: Simulates payments via a mock gateway
* **Notification Service**: Sends transactional and system emails

### MySQL Database

Stores data for users, hotels, rooms, bookings, payments, reviews, and notifications.

### Mock Payment Gateway

Simulates payment flows to validate booking logic without real transactions.

### SMTP/Email Service

Sends automated emails for booking confirmations and system notifications.

## Technology Stack

| **Layer** | **Technology** |
| --- | --- |
| **Frontend** | React.js, Tailwind CSS, React Router |
| **Backend** | Java, Spring Boot, Spring Security, Spring Data JPA |
| **Database** | MySQL |
| **CI/CD** | GitHub Actions for automated builds and deployments |
| **Testing** | JUnit (backend), Jest + React Testing Library (frontend), JMeter (Performance Testing) |
| **Integration** | Email Service |
| **Deployment Platform** | AWS EC2 (Ubuntu VM with NGINX and Java runtime) |

## Transition from Analysis to Design

To ensure a smooth transition from **requirements analysis** to **system design**, several structured strategies were applied. These techniques ensured that business needs captured in user stories and use cases were consistently translated into technical design and implementation. This systematic approach helped maintain **traceability**, **modularity**, and **alignment** across the development lifecycle.

| **Strategy** | **Description** |
| --- | --- |
| **Use Case → Sequence Diagrams** | For each use case (e.g., Book Hotel, Manage Hotels), sequence diagrams were created to visualize interactions between actors and system components. These diagrams helped identify key classes, methods, and service boundaries necessary to implement each use case. |
| **Domain Model Mapping** | Entities identified during the analysis phase (e.g., User, Hotel, Room, Booking, Payment) were directly mapped to Java classes with their corresponding attributes and relationships, ensuring consistency between the data model and object-oriented design. |
| **Layered Architecture** | Each use case was mapped to the appropriate layer of the software architecture: Presentation Layer for UI components, Business Logic Layer for service operations and validations, and Data Access Layer for persistence logic, supporting modularity and separation of concerns. |
| **Controller-Service-Repository** | The backend followed the Controller-Service-Repository pattern, where controllers were derived from use cases to handle HTTP requests, services encapsulated business logic, and repositories managed CRUD operations through JPA, ensuring clean separation of concerns. |
| **Design Patterns** | Design patterns such as Factory (for user roles like Customer and Admin), Strategy (for different payment methods), and Observer (for triggering notifications) were applied to solve common design problems and promote code reusability. |
| **Traceability Matrix** | A traceability matrix was created to link user stories and use cases to their corresponding UI components, controller/service methods, database entities, and test cases, ensuring full coverage from analysis through to design and implementation. |
| **API Contract (Swagger)** | For any user stories requiring frontend-backend communication, OpenAPI (Swagger) specifications were defined, outlining API endpoints, request/response formats, and expected behaviors, ensuring clear contracts and synchronized development between teams. |

## Transition Strategy 1: Use Case Realization through Sequence Diagrams

* **Analysis Objects Impacted**: Actors (User, Registered User, Admin), Use Cases
* **Impacted Use Cases**: Book Hotel, Manage Hotels, Manage Accounts, Login, Search Hotels, Manage Promotions
* **Class Diagram Changes**:  
  + Identification of controller classes (e.g., **BookingController, HotelController**).
  + Addition of service classes (**BookingService, UserService**).
  + Mapping of entity relationships (e.g., **User → Booking, Hotel → Room**).
  + **Rationale**: These classes are core to the architectural layers and tied to user-initiated system behaviors.
* **Sequence Diagram Changes**:  
  + Addition of method calls and response flows between actors, controllers, services, and repositories.
  + **Rationale**: Visualizes how the system responds to actions like "book hotel" or "manage accounts", ensuring clear process understanding.

## Transition Strategy 2: Domain Model Mapping

* **Analysis Objects Impacted**: Domain entities (e.g., User, Hotel, Room, Booking, Payment)
* **Impacted Use Cases**: All use cases involving data operations
* **Class Diagram Changes**:  
  + Class definitions with attributes based on database schema.
  + Cardinality and associations (e.g., 1 Hotel has many Rooms, 1 Booking linked to 1 Room and 1 User).
  + **Rationale**: Ensures consistency between data models and software representations.
* **Sequence Diagram Changes**:  
  + Interactions between service methods and entities (e.g., **bookingService.createBooking()** calls **bookingRepository.save(booking)**).
  + **Rationale**: Clarifies object persistence and lifecycle during user operations.

## Transition Strategy 3: Layered Architectural Alignment

* **Analysis Objects Impacted**: System components, Use Cases, Sequence flows
* **Impacted Use Cases**: All use cases, especially backend processing
* **Class Diagram Changes**:  
  + Introduction of layers: Presentation, Service, Repository.
  + Clear separation between DTOs, entity classes, and controllers.
  + **Rationale**: Promotes clean, maintainable, and scalable architecture.
* **Sequence Diagram Changes**:  
  + Sequence flow must go through appropriate layers (UI → Controller → Service → Repository).
  + **Rationale**: Enforces architectural discipline and ensures proper responsibility distribution.

## Transition Strategy 4: Controller-Service-Repository Pattern

* **Analysis Objects Impacted**: Functional Requirements, Entity-Related Use Cases
* **Impacted Use Cases**: Login, Book Hotel, Manage Payments, Manage Hotels, Update Profile
* **Class Diagram Changes**:  
  + Creation of controller classes for HTTP request handling.
  + Services encapsulate business logic.
  + Repositories abstract DB operations.
  + **Rationale**: Isolates business logic from routing and persistence.
* **Sequence Diagram Changes**:  
  + Clear flow from controller → service → repository, indicating responsibility boundaries.
  + **Rationale**: Promotes a reusable, testable method for implementing use case behavior.

## Transition Strategy 5: Design Pattern Application

* Analysis Objects Impacted: Complex behaviors in use cases
* Impacted Use Cases: Payment Service (Strategy Pattern), Notification Trigger (Observer Pattern), User Role Management (Factory Pattern)
* Class Diagram Changes:  
  + Additional interfaces and concrete classes: **PaymentStrategy** with **CreditCardPayment, PayPalPayment, UserFactory** for generating roles.
  + Rationale: Promotes flexibility, scalability, and adherence to SOLID principles.
* Sequence Diagram Changes:  
  + Dynamic delegation of behaviors (e.g., paymentService.setStrategy(payPal).processPayment()).
  + Rationale: Supports interchangeable strategies and event-driven interactions without modifying core logic.

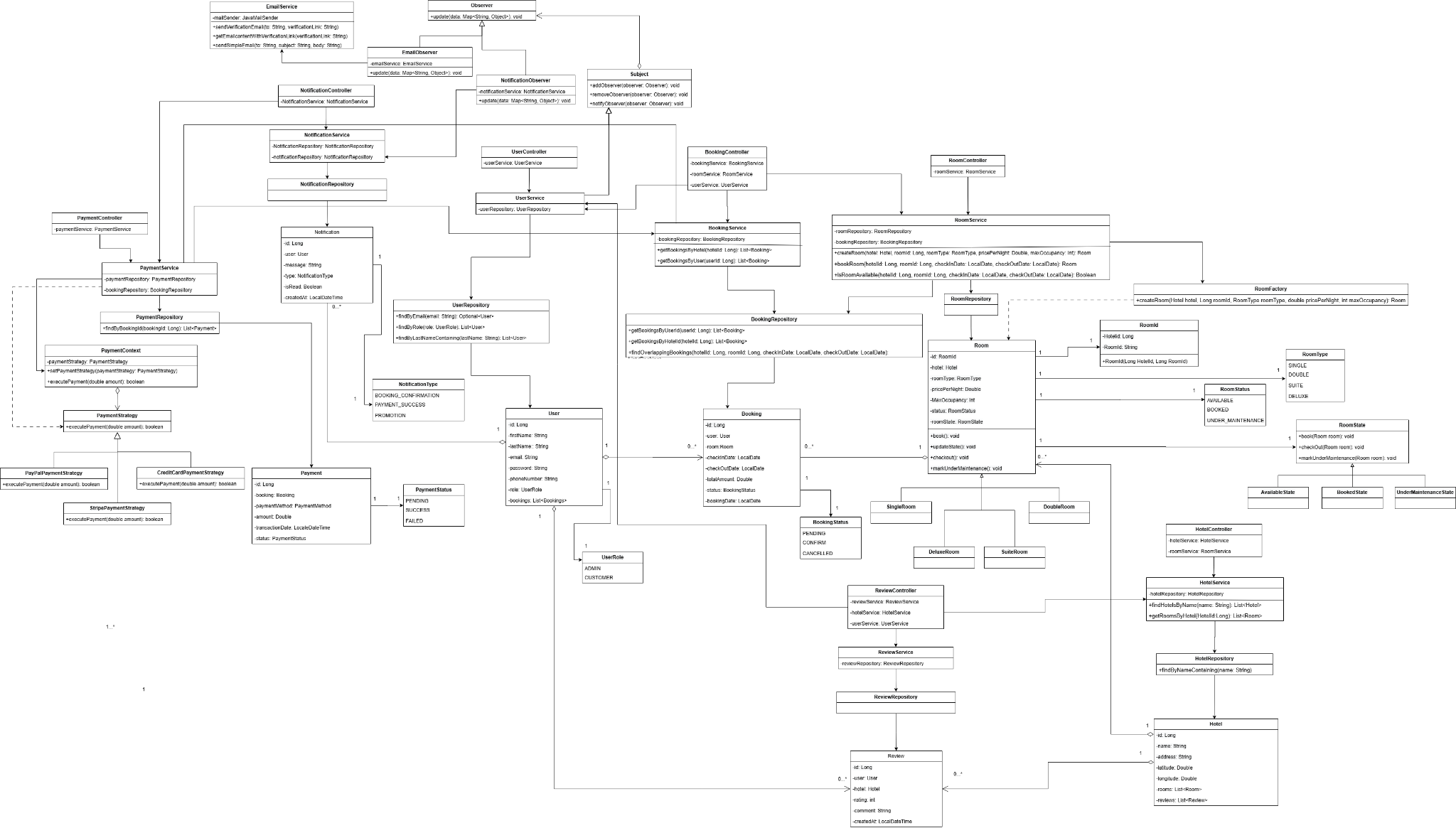
## Transition Strategy 6: API Contract Design

* Analysis Objects Impacted: Frontend-Backend Interaction Specifications
* Impacted Use Cases: Search Hotels, Book Hotel, Login, Update Profile
* Class Diagram Changes:  
  + Definition of Data Transfer Objects (DTOs) representing JSON payloads for requests/responses.
  + Rationale: Ensures separation between domain models and API exposure layers.
* Sequence Diagram Changes:  
  + Inclusion of frontend calls to backend endpoints and expected responses.
  + Rationale: Ensures synchronization between frontend and backend during integration phases.

## 

## Behavioral and Structural Diagrams

### UML Class Diagram

**

| **Feature** | **Components** | **Description** |
| --- | --- | --- |
| **Booking** | BookingController, BookingService, BookingRepository, Booking | Manages hotel bookings (create, update, cancel, retrieve). |
| **Hotel** | HotelController, HotelService, HotelRepository, Hotel | Manages hotel data like name, location, and check-in/out times. |
| **User** | UserController, UserService, UserRepository, User | Handles user registration, login, profiles, and verification. |
| **Room** | RoomController, RoomService, RoomRepository, Room | Manages rooms in hotels including type, price, and availability. |
| **Notification** | NotificationController, NotificationService, NotificationRepository, Notification | Sends and manages system notifications for users. |
| **Review** | ReviewController, ReviewService, ReviewRepository, Review | Handles user reviews for hotels and rooms. |
| **Payment** | PaymentController, PaymentService, PaymentRepository, Payment | Processes and tracks payments for bookings. |
| **Bookmark** | BookmarkController, BookmarkService, BookmarkRepository, Bookmark | Lets users bookmark their favorite hotels. |
| **Email** | EmailService | Sends email verifications and other email notifications. |
| **Main App** | StaymateApplication | Spring Boot app’s main entry point. |

### Use Case Diagram

### 

### Use Case Description 1: Login

| **Item** | **Description** |
| --- | --- |
| **Actor** | User |
| **Precondition** | User has an existing account. |
| **Postcondition** | User is logged in and redirected to the homepage or dashboard. |
| **Trigger** | User clicks "Log in" on the login page. |
| **Priority** | High |

### Flow of Events

#### Normal Flow

* + User accesses the login page.
  + User enters their email and password.
  + System verifies the credentials against stored user data.
  + If the credentials are valid, the system logs the user in.
  + The user is redirected to the dashboard or homepage.

#### Exceptional Flow

* + If the credentials are **invalid**:
    - The system displays an **error message** (e.g., "Invalid email or password").
    - The user is prompted to **retry login**.

### 

### 

### Use Case Description 2: Browse Hotels

| **Item** | **Description** |
| --- | --- |
| **Actor** | User / Registered User |
| **Precondition** | User has access to the application (login not mandatory). |
| **Postcondition** | User is able to view a list of hotels or detailed information about a hotel. |
| **Trigger** | User selects the option to browse hotels. |
| **Priority** | High |

### Flow of Events

#### Normal Flow

* + User navigates to the hotel browsing page.
  + System displays a list of all available hotels with basic information (e.g., name, location, price, rating).
  + Users can click on a specific hotel to view detailed information (e.g., rooms, amenities, reviews).

#### Exceptional Flow

* + If there are no hotels available, the system displays a message like: “No hotels found. Please try again later.”

### 

### Use Case Description 3: Search Hotels

| **Item** | **Description** |
| --- | --- |
| **Actor** | User / Registered User |
| **Precondition** | User is on the hotel browsing or search page. |
| **Postcondition** | System displays a list of hotels matching the search criteria. |
| **Trigger** | User enters a search query (e.g., hotel name) and initiates the search. |
| **Priority** | High |

### Flow of Events

#### Normal Flow

* + User navigates to the hotel browsing or search page.
  + User enters search criteria (e.g., hotel name, location).
  + System processes the search query.
  + System displays a list of hotels that match the criteria with basic information (e.g., name, location, price, rating).
  + Users can click on a specific hotel to view detailed information (e.g., rooms, amenities, reviews).

#### Exceptional Flow

* + If no hotels match the search criteria:
  + The system displays a message like: *“No matching hotels found. Please try different keywords.”*

### 

### 

### 

### 

### Use Case Description 4: Bookmark Hotel

| **Item** | **Description** |
| --- | --- |
| **Actor** | Registered User |
| **Precondition** | User is logged in and browsing hotels. |
| **Postcondition** | Hotel is added to the user’s saved list. |
| **Trigger** | User clicks the “bookmark” button on a hotel. |
| **Priority** | Medium |

### Flow of Events

#### Normal Flow

* + User clicks the “bookmark” button on a hotel.
  + System stores the hotel in the user’s saved list.
  + Users can view saved hotels from their profile.

#### Exceptional Flow

* + If there’s an issue with saving, the system shows an error message.

### 

### Use Case Description 5: Update Profile

| **Item** | **Description** |
| --- | --- |
| **Actor** | Registered User / Admin |
| **Precondition** | User is logged in. |
| **Postcondition** | User’s profile is updated successfully. |
| **Trigger** | Users navigate to their profile to edit. |
| **Priority** | High |

### Flow of Events

#### Normal Flow

* + User navigates to their profile.
  + User edits details like name, phone number, or password.
  + System validates and updates the profile.

#### Exceptional Flow

* + If validation fails (e.g., invalid phone number), the system prompts for correction.

### 

### Use Case Description 6: Book Hotel

| **Item** | **Description** |
| --- | --- |
| **Actor** | Registered User |
| **Precondition** | User has selected a hotel and room. |
| **Postcondition** | Booking is confirmed and payment proceeds. |
| **Trigger** | User initiates the booking process. |
| **Priority** | High |

### Flow of Events

#### Normal Flow

* + User selects a hotel and room.
  + User provides booking details (dates, guests).
  + System calculates the total amount and confirms availability.
  + Booking proceeds to the Payment Service.

#### Exceptional Flow

* + If a room becomes unavailable, the system notifies and suggests alternatives.

### 

### Use Case Description 7: Payment Service (Included by Book Hotel)

| **Item** | **Description** |
| --- | --- |
| **Actor** | System |
| **Precondition** | User has confirmed booking and selected a payment method. |
| **Postcondition** | Payment is processed and booking is confirmed. |
| **Trigger** | User enters payment details to confirm booking. |
| **Priority** | High |

### Flow of Events

#### Normal Flow

* + User enters payment method and details.
  + System processes the transaction via the mock payment gateway.
  + On success, system confirms the booking.

#### Exceptional Flow

* + If payment fails, the system aborts booking and notifies the user.

### 

### Use Case Description 8: Manage Accounts

| **Item** | **Description** |
| --- | --- |
| **Actor** | Admin |
| **Precondition** | Admin is logged in. |
| **Postcondition** | User account details are updated. |
| **Trigger** | Admin selects a user from the list to update. |
| **Priority** | Medium |

### Flow of Events

#### Normal Flow

* + Admin logs into the dashboard.
  + Admin navigates to the user account management section.
  + Admin selects a user from the list of existing accounts.
  + Admin updates user details such as name, email, phone number.
  + System validates and applies the updates.

#### Exceptional Flow

* + If the admin inputs invalid data (e.g., incorrectly formatted email, empty fields), the system displays an error message and prevents the update until corrected.

### 

### 

### 

### Use Case Description 9: Manage Hotel

| **Item** | **Description** |
| --- | --- |
| **Actor** | Admin |
| **Precondition** | Admin is logged in and accessing the hotel management page. |
| **Postcondition** | Hotel and room details are updated or deleted. |
| **Trigger** | Admin decides to add, edit, or delete a hotel listing or room details. |
| **Priority** | High |

### 

### Flow of Events

#### Normal Flow

* + Admin navigates to the hotel management page.
  + Admin adds, edits, or deletes hotel listings and room details.

#### Exceptional Flow

* + System restricts deletion if a hotel has active bookings.

### 

### Use Case Description 10: Manage Promotion Offers

| **Item** | **Description** |
| --- | --- |
| **Actor** | Admin |
| **Precondition** | Admin is logged in. |
| **Postcondition** | Promotional offers are applied to relevant hotels/rooms. |
| **Trigger** | Admin creates or updates promotional offers. |
| **Priority** | Medium |

### Flow of Events

#### Normal Flow

* + Admin creates or updates promotional offers.
  + System validates and stores offer details.
  + Offers are applied to relevant hotels/rooms.

#### Exceptional Flow

* + If offer conditions are invalid (e.g., date range mismatch), the system rejects the input.

### 

### Use Case Description 11: Manage Payments

| **Item** | **Description** |
| --- | --- |
| **Actor** | Admin |
| **Precondition** | Admin is logged in and has access to payment management. |
| **Postcondition** | Admin reviews transaction history and processes refunds if necessary. |
| **Trigger** | Admin selects the payment management section. |
| **Priority** | High |

### 

### Flow of Events

#### Normal Flow

* + Admin accesses the payment management section.
  + Admin views transaction history and statuses.
  + Admin can trigger refunds or investigate failed transactions.

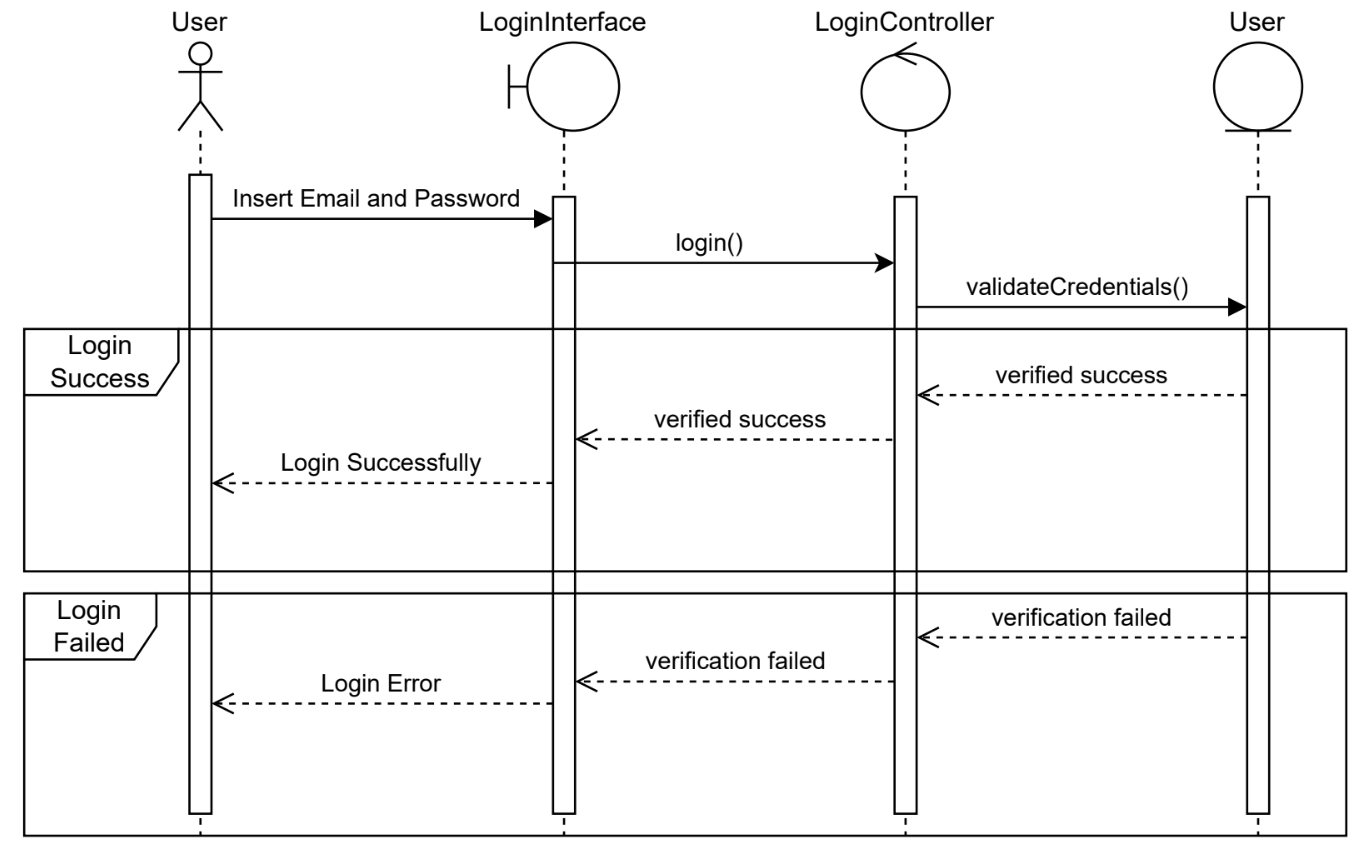
#### Exceptional Flow

* + Admin cannot manually process payments due to exclusion of Payment Service, maintaining separation of concerns.

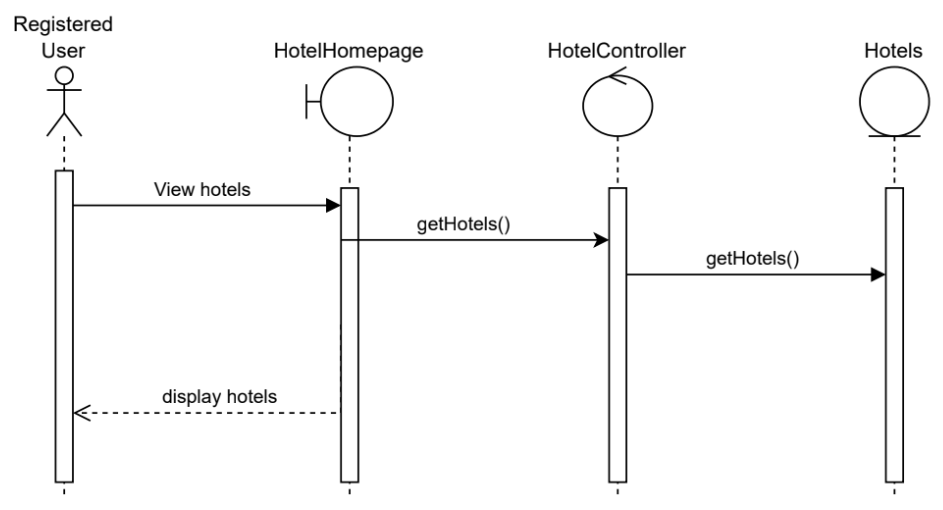
## 

## Sequence Diagrams

### Use Case 1: Login by Ng Ju Khang

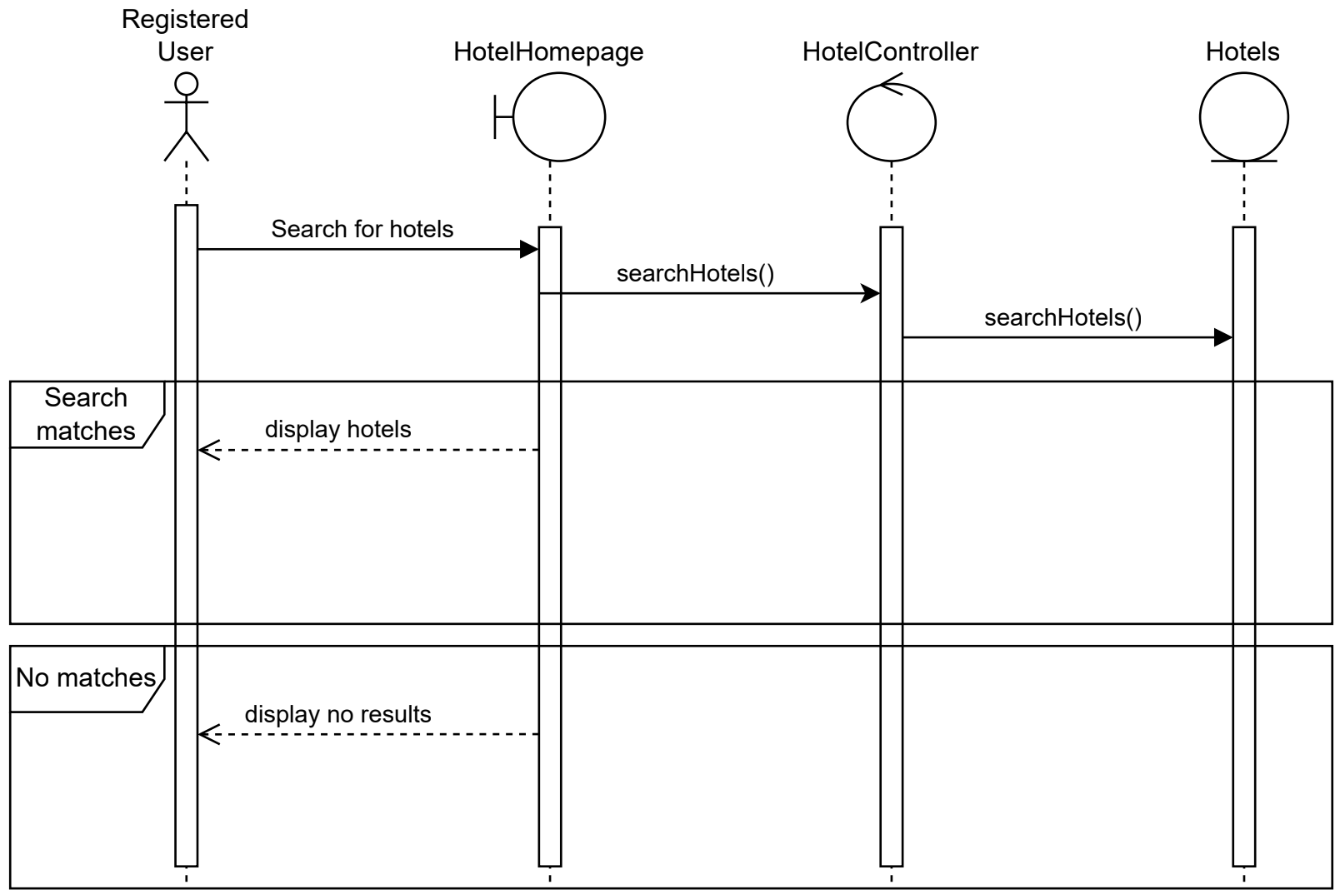


### Use Case 2: Browse Hotels by Ng Ju Khang

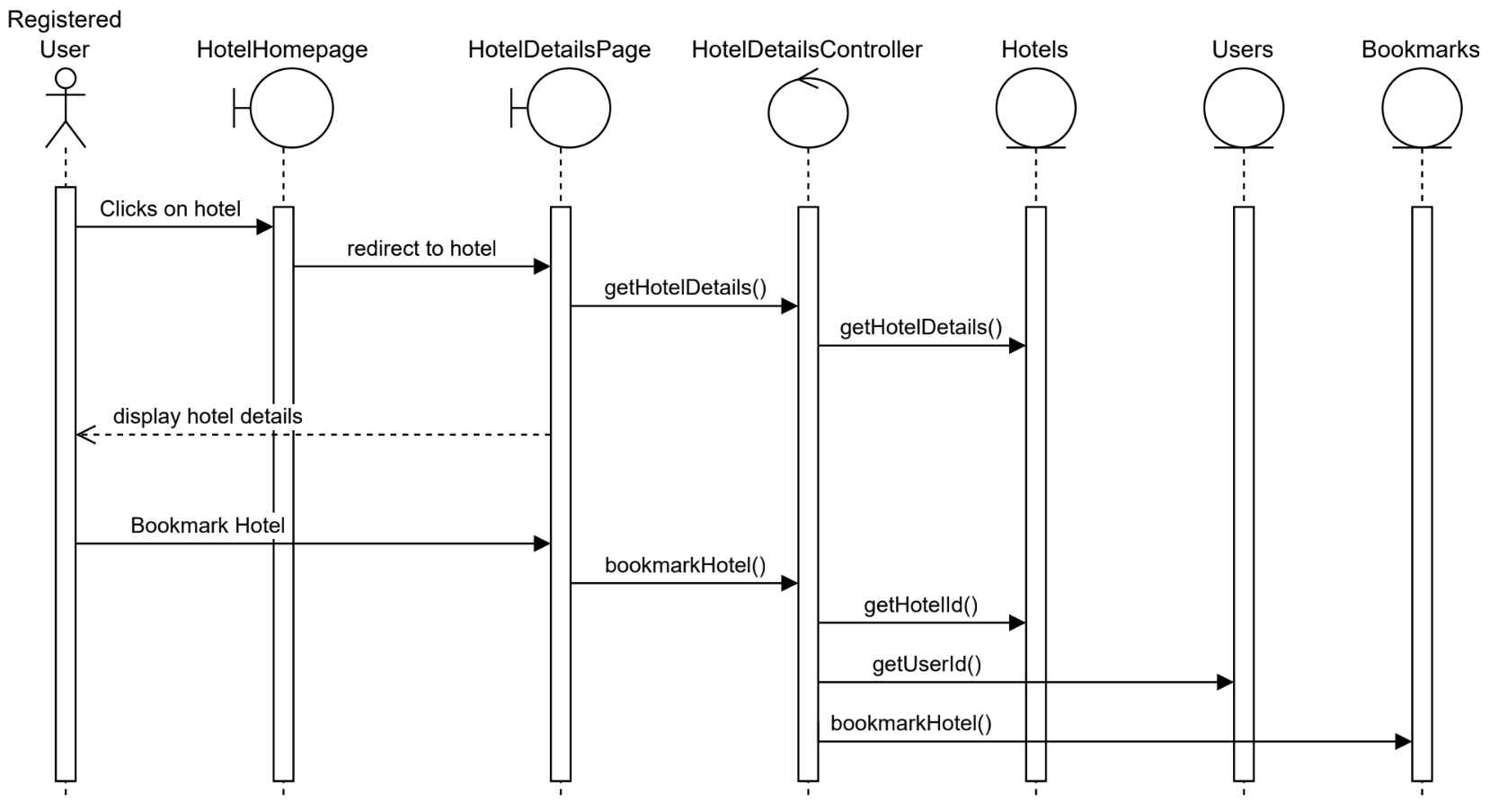


### Use Case 3: Search for Hotels by Ng Ju Khang

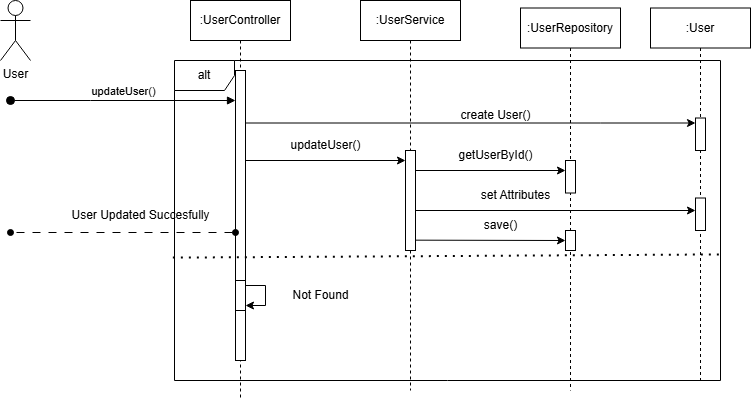
Sequence Diagram



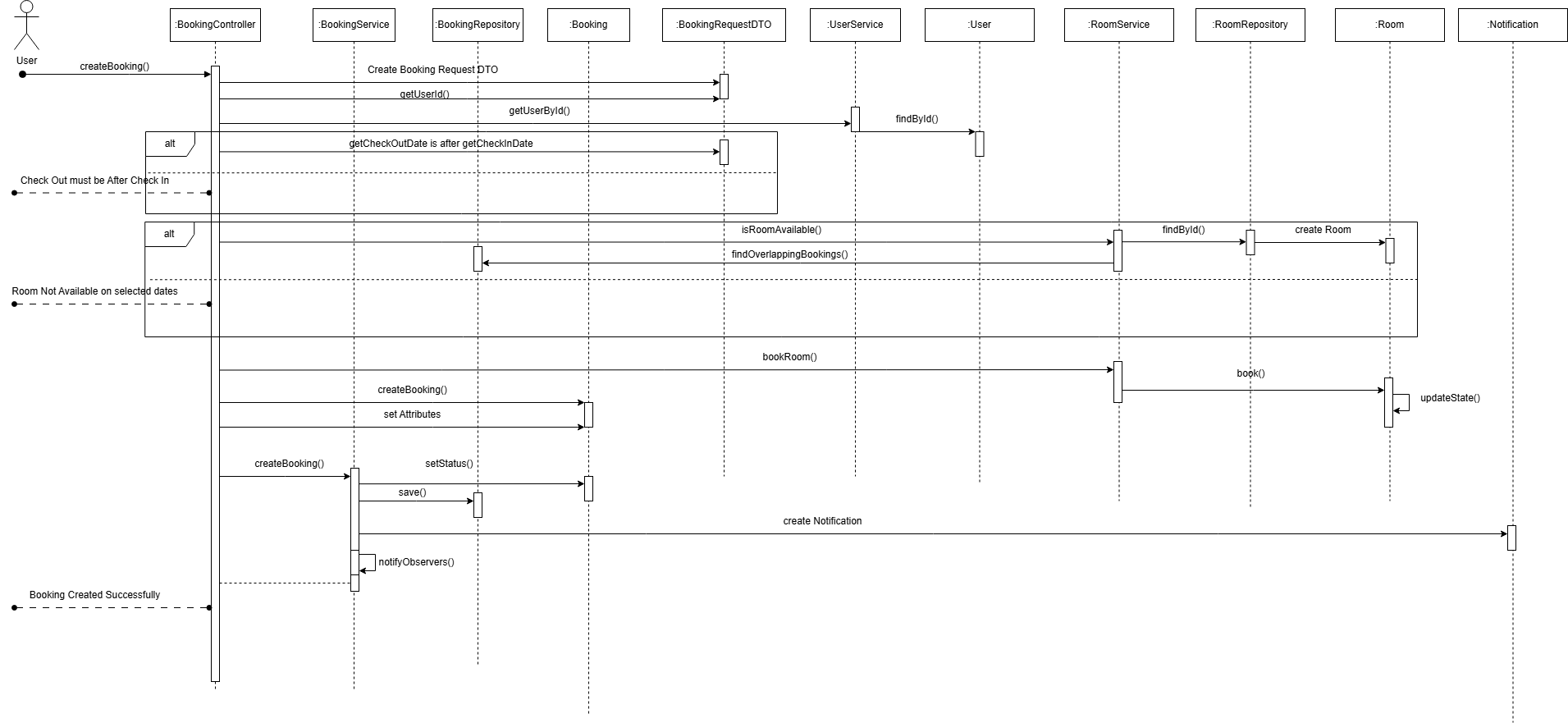
### Use Case 4: Bookmark Hotel by Ng Ju Khang



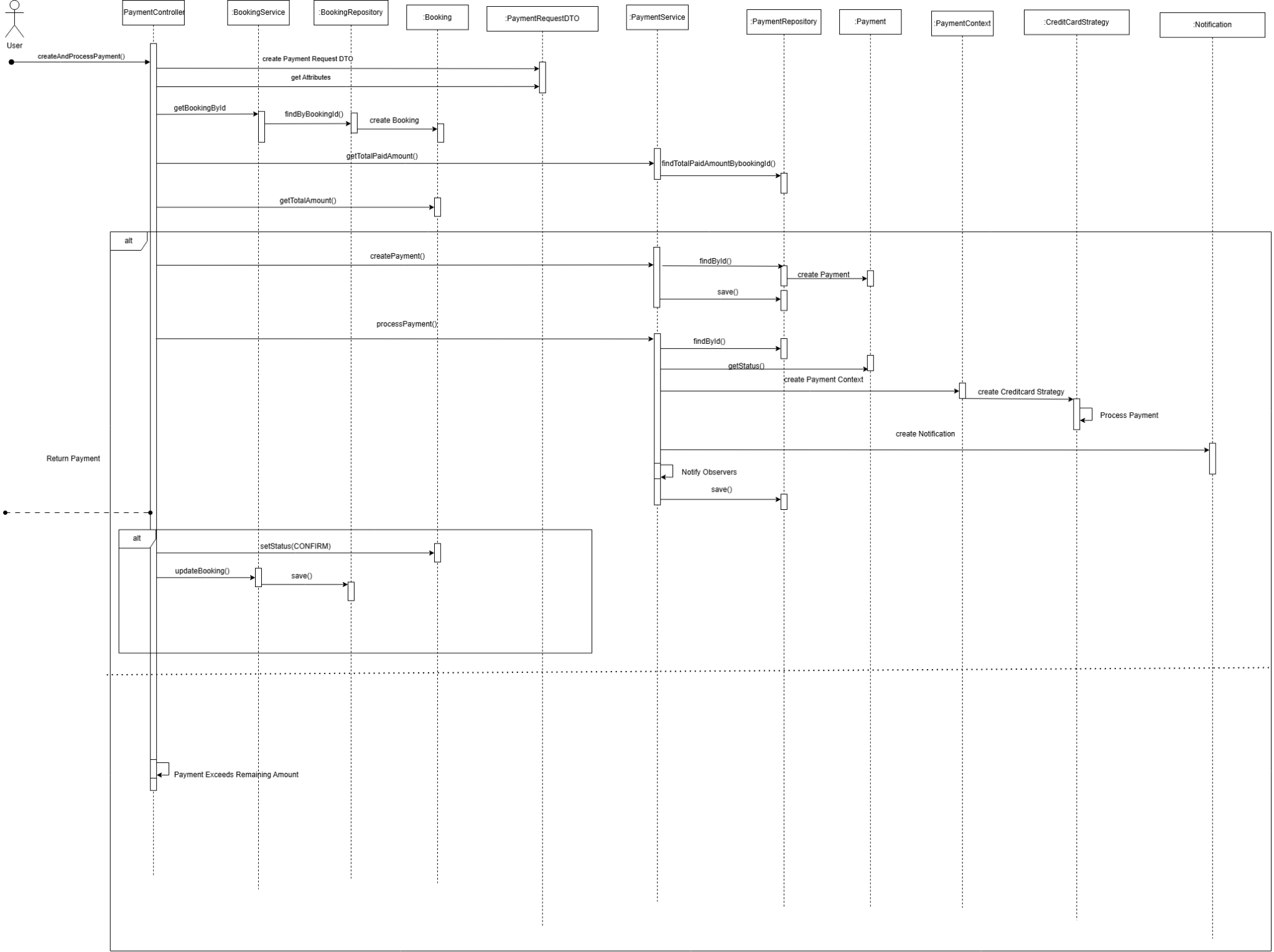
### Use Case 5: Update Profile by Ng Zhi Kai



### Use Case 6: Book Hotel by Ng Zhi Kai

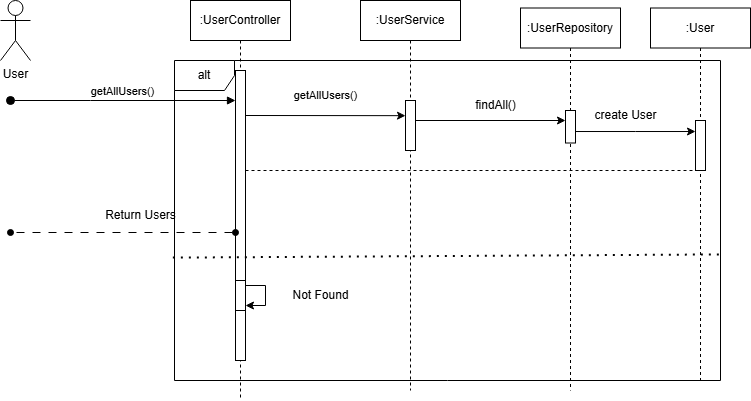


### Use Case 7: Make Payment by Ng Zhi Kai

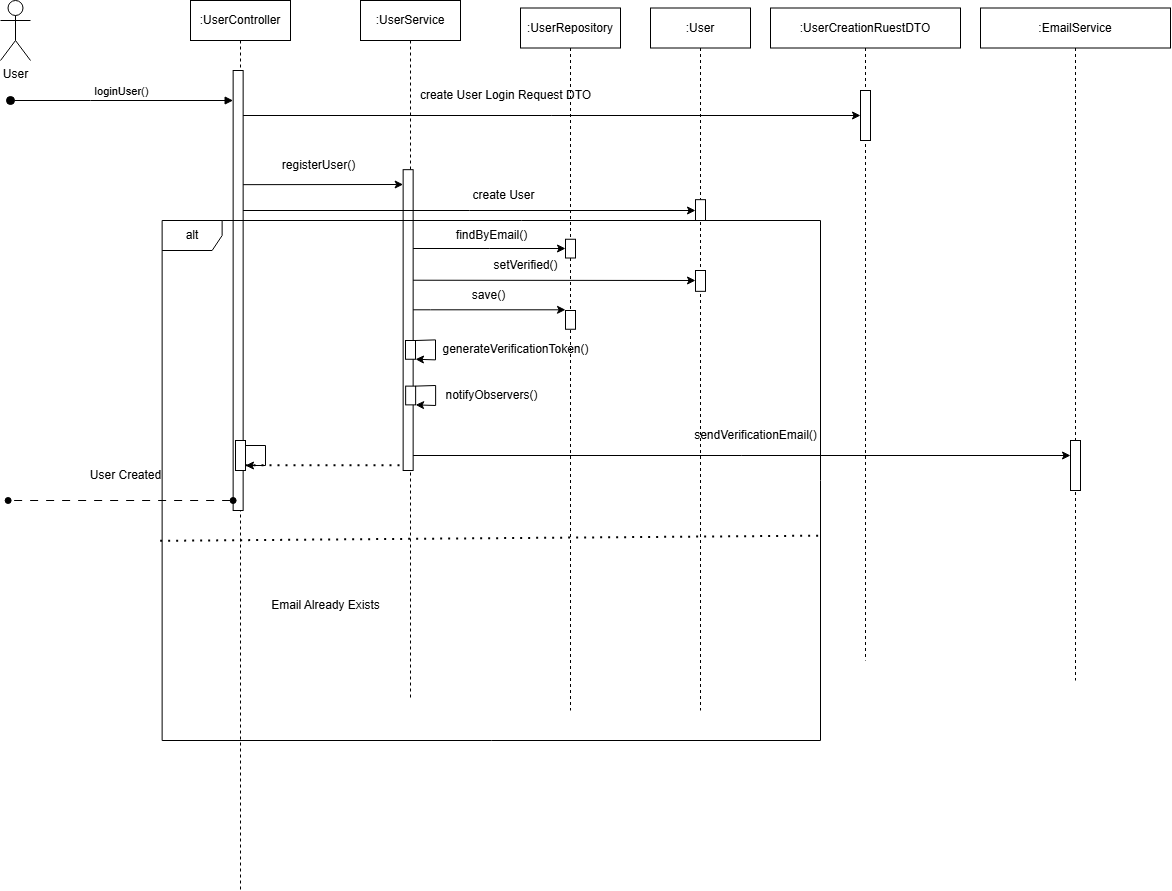


### Use Case 8: Manage Users by Ng Zhi Kai

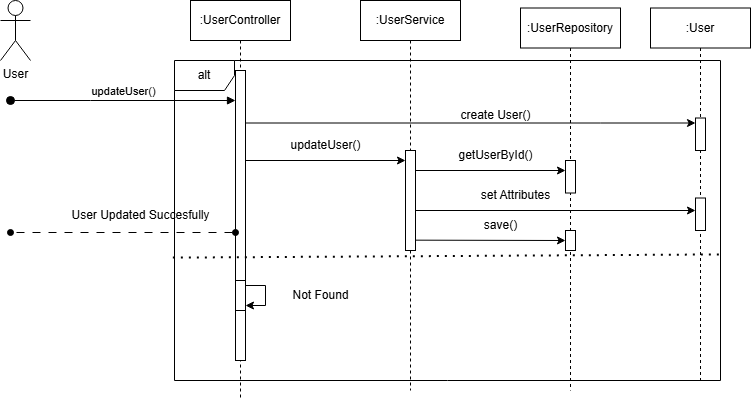
#### View All Users



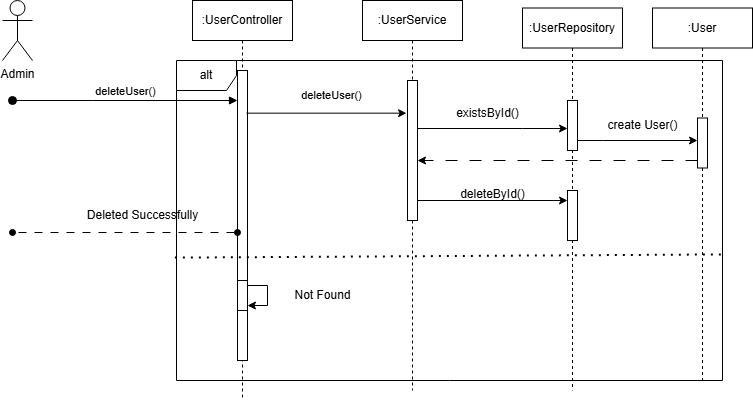
#### Register Users



#### Edit Users

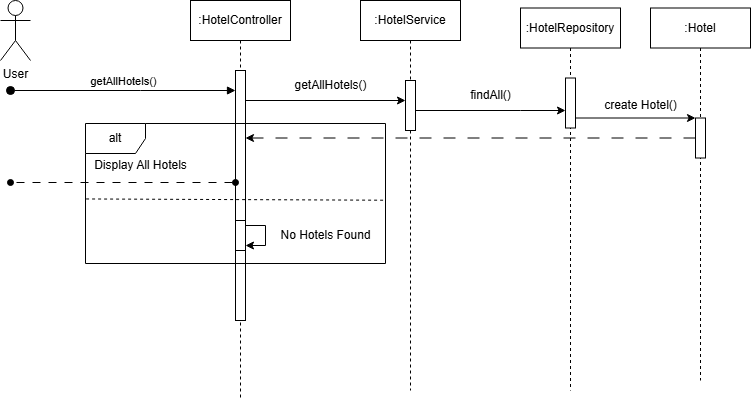


#### Delete Users

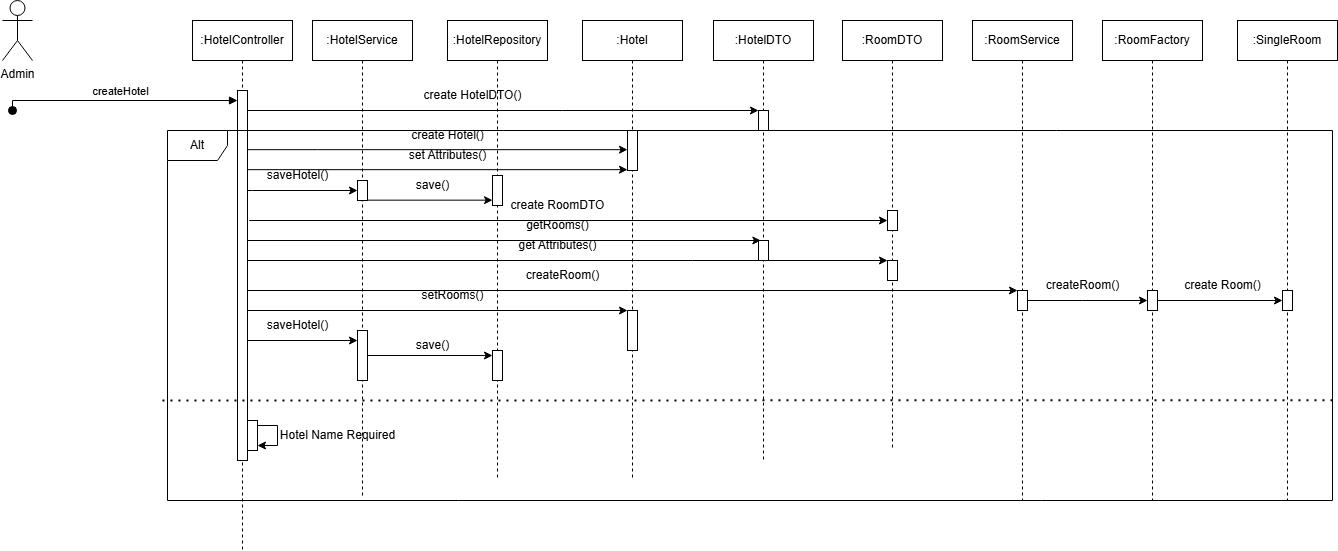


### Use Case 9: Manage Hotel by Ng Zhi Kai

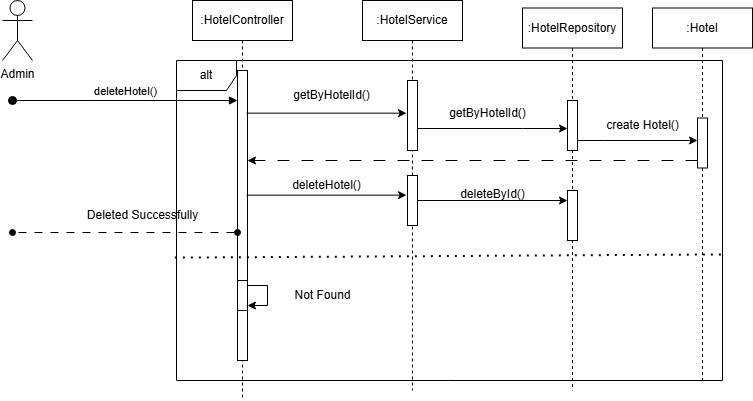
#### View All Hotels



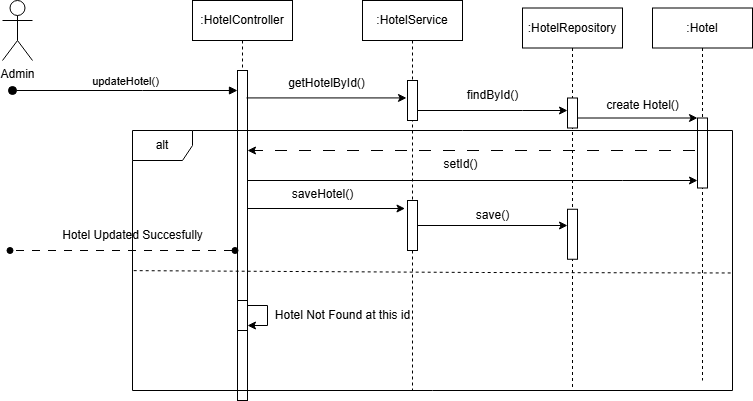
#### Create Hotels



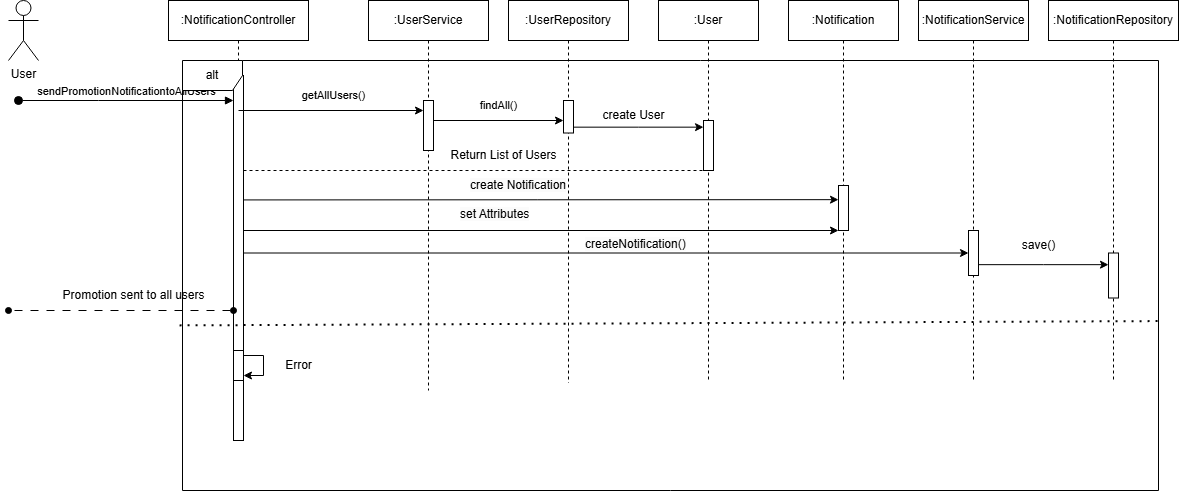
#### Delete Hotels



#### Edit Hotels

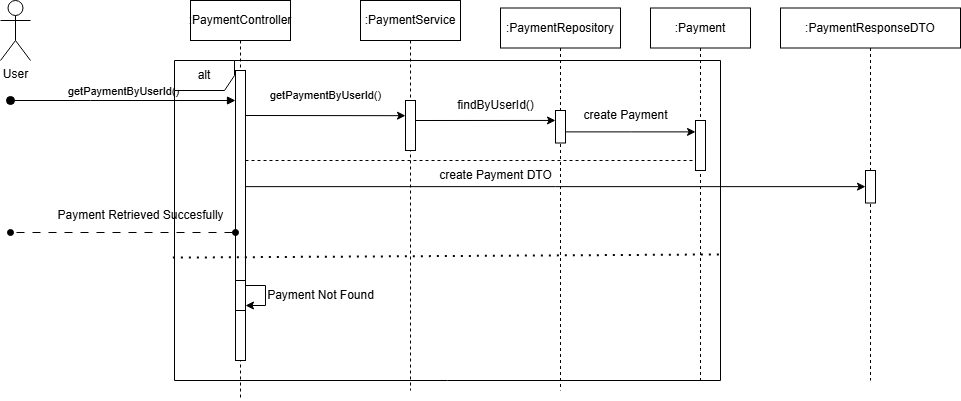


### Use Case Description 10: Manage Promotion Offers by Ng Zhi Kai

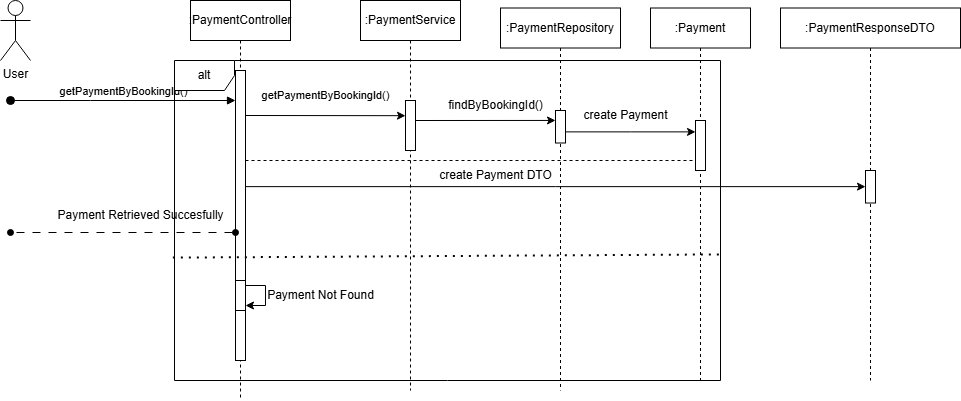


### Use Case Description 11: Manage Payments by Ng Zhi Kai

#### Find Payments by User



#### Find Payments by Booking



## 

## Design Problems and Patterns

### Design Problem for Payment Service by Ng Zhi Kai

### Initial Implementation

In Staymate, users can pay via Credit Card, PayPal, or Bank Transfer. Initially, **PaymentService** handled only Credit Card payments using hardcoded **if-else** logic:

*if (method == "CARD") { ... }*

*else if (method == "PAYPAL") { ... }*

*else if (method == "BANK") { ... }*

This approach is rigid, violates the **Open/Closed Principle**, and complicates testing and future extension when new payment methods are introduced.

### Candidate Design Pattern(s)

* **Strategy Pattern (Chosen):** Best fits the need to encapsulate different payment methods and make them interchangeable.
* **Template Method Pattern (Rejected):** Less dynamic and not suitable for runtime strategy switching.

### Motivation for Strategy Pattern

The **Strategy Pattern** is chosen for the following reasons:

* **Modular Design:** Each payment method is a separate class implementing a shared interface.
* **Easy to Extend:** New methods can be added without modifying existing logic.
* **Open/Closed Compliance:** Core logic remains unchanged when adding new strategies.
* **Simplified Testing:** Strategies can be tested independently, and **PaymentService** simply delegates.

### Sequence Flow

1. User selects a payment method.
2. **PaymentController** calls **PaymentService**.**setPaymentStrategy**().
3. **PaymentService** delegates payment processing to the chosen strategy.
4. **PaymentStrategy**.**processPayment**() handles the transaction.

| **Code** | **Diagram** |
| --- | --- |
| *public class PaymentContext {*  *private PaymentStrategy paymentStrategy;*  *public void setPaymentStrategy(PaymentStrategy paymentStrategy) {*  *this.paymentStrategy = paymentStrategy;*  *}*  *public boolean executePayment(double amount) {*  *if (paymentStrategy == null) {*  *System.err.println("Error: Payment method not set.");*  *return false;*  *}*  *System.out.println("Processing payment using: " + paymentStrategy.getClass().getSimpleName());*  *return paymentStrategy.processPayment(amount);*  *}*  *}* |  |

This refactor enables dynamic payment method selection and promotes flexibility. Adding a new method only requires a new class implementing **PaymentStrategy**, keeping the design clean and extensible.

### 

### Design Problem for Create Notification by Ng Zhi Kai

### Initial Implementation

In Staymate, users receive notifications for events like bookings and payments. Initially, **NotificationService** used hardcoded **if-else** logic to handle event types, making the system rigid and hard to extend or maintain.

*public void sendNotification(String eventType, String message) {*

*if (eventType.equals("BOOKING")) {*

*System.out.println("Booking: " + message);*

*} else if (eventType.equals("PAYMENT")) {*

*System.out.println("Payment: " + message);*

*} else {*

*System.out.println("Invalid event type.");*

*}*

*}*

This violates the **Open/Closed Principle**, making it hard to support new event types like cancellations or refunds.

### Candidate Design Pattern(s)

* **Observer Pattern (Chosen)**: Allows **BookingService**, **PaymentService**, etc., to notify observers like **NotificationService** dynamically.
* **Strategy Pattern (Rejected)**: Better for encapsulating interchangeable algorithms, not for event-based notification flows.

### Motivation for Observer Pattern

* **Decoupling:** Services notify observers without knowing how notifications are handled.
* **Modularity & Extensibility:** New event types can be added via new observers, without touching existing code.
* **Open/Closed Principle:** Add functionality without modifying source.
* **Testability:** Each observer can be tested independently.

### Sequence Diagram Flow

1. **Booking Event**:
   * **BookingService** generates a booking event.
   * The **NotificationService** (observer) receives the event and sends a booking-related notification.
2. **Payment Event**:
   * **PaymentService** processes a payment and generates a payment event.
   * The **NotificationService** (observer) receives the event and sends a payment-related notification.

| **Code** | **Diagram** |
| --- | --- |
| *public class BookingService {*  *private List<Observer> observers = new ArrayList<>();*  *public void addObserver(Observer observer) {*  *observers.add(observer);*  *}*  *public void confirmBooking(String bookingId) {*  *System.out.println("Booking confirmed: " + bookingId);*  *for (Observer observer : observers) {*  *observer.update("BOOKING", "Your booking has been confirmed.");*  *}*  *}*  *}*  *public class PaymentService {*  *private List<Observer> observers = new ArrayList<>();*  *public void addObserver(Observer observer) {*  *observers.add(observer);*  *}*  *public void processPayment(String paymentId, double amount) {*  *System.out.println("Payment processed: " + paymentId + " Amount: " + amount);*  *for (Observer observer : observers) {*  *observer.update("PAYMENT", "Your payment has been processed.");*  *}*  *}*  *}* |  |

With this refactor, the **NotificationService** becomes a flexible observer of various events (like booking or payment), and new event types can be handled by adding new observer classes without touching the existing event generation code.

### Design Problem for Create Room by Ng Zhi Kai

### Initial Implementation

In the Staymate system, **RoomService** initially created rooms directly using **RoomRepository**. As room types grew (e.g., Deluxe, Suite), this approach became rigid, violating the **Open/Closed Principle** and making extension difficult.

*public class RoomService {*

*public createRoom(Room room){*

*return roomRepository.save(room);*

*}*

*}*

Any changes or additions to room types required modifications to **RoomService**, reducing flexibility.

### Candidate Design Pattern(s)

* **Factory Pattern (Chosen)**: Decouples creation logic, promotes scalability, easy to add new room types
* **Builder Pattern (Rejected)**: Although the Builder Pattern can help in constructing complex objects, it’s more suited for situations where the object creation process is complicated. In this case, a simple factory approach is more appropriate.

### Motivation for Factory Pattern

* **Separation of Concerns**: Creation logic moves to **RoomFactory**, service stays clean
* **Scalability**: Add new room types without touching **RoomService**
* **Open/Closed Principle**: Extendable without modifying existing logic
* **Testability**: Easier to mock and isolate creation logic during testing

### Sequence Diagram Flow

1. User requests to create a room.
2. RoomService calls RoomFactory to create the room based on type.
3. RoomFactory instantiates the appropriate Room subclass.
4. RoomService receives the Room and proceeds with business logic.

| **Code** | **Diagram** |
| --- | --- |
| *public class RoomFactory {*  *public Room createRoom(Hotel hotel, Long roomId, RoomType roomType, double pricePerNight, int maxOccupancy) {*  *switch (roomType) {*  *case SINGLE: return new SingleRoom(hotel, roomId, pricePerNight, maxOccupancy);*  *case DOUBLE: return new DoubleRoom(hotel, roomId, pricePerNight, maxOccupancy);*  *case DELUXE: return new DeluxeRoom(hotel, roomId, pricePerNight, maxOccupancy);*  *case SUITE: return new SuiteRoom(hotel, roomId, pricePerNight, maxOccupancy);*  *default: throw new IllegalArgumentException("Unknown room type");*  *}*  *}*  *}* |  |

With this refactor, the **RoomService** becomes a flexible and focused component that delegates room creation to the **RoomFactory**. This decouples creation logic from business logic, enabling easier maintenance, testing, and extension of new room types without modifying existing service code.

### Design Problem for Room States by Ng Zhi Kai

### Initial Implementation

In the Staymate system, room states (e.g., available, booked, cleaned) were managed directly in the **Room** class, with state transitions embedded in business methods. This approach became rigid and error-prone as the system grew and making it difficult to extend or modify behavior without altering the core room class.

*public class Room {*

*private String state;*

*public void setState(String state) {*

*this.state = state;*

*}*

*public void bookRoom() {*

*if (state.equals("available")) {*

*setState("booked");*

*} else {*

*// Handle invalid state transition*

*}*

*}*

*}*

### Candidate Design Pattern(s)

* **State Pattern (Chosen)**: The State Pattern is ideal for handling state transitions where each state can have its own behavior and the context (e.g., **Room**) delegates the state-specific behavior to state objects.
* **Strategy Pattern (Rejected)**: The Strategy Pattern is not as suited to managing state transitions as the State Pattern, which explicitly handles different behaviors based on the current state.

### Motivation for State Pattern

* **Separation of Concerns:** Each state (e.g., **AvailableState**, **BookedState**) is encapsulated in its own class, ensuring the Room class remains focused on its core functionality.
* **Open/Closed Principle:** New states or transitions can be added by introducing new state classes, without modifying the Room class.
* **State-Specific Behavior:** Each state can implement its own behavior (e.g., **bookRoom()** can behave differently depending on the current state of the room).
* **Simplified Maintenance:** Changes to the logic for a particular state are localized to the corresponding state class, making it easier to maintain and extend the system.

### Sequence Diagram Flow

1. The user interacts with the **Room** (e.g., trying to book a room).
2. **Room** delegates the action to the current state (e.g., **AvailableState**).
3. The state performs the action (e.g., changing state to "booked").
4. The **Room** object reflects the new state.

| **Code** | **Diagram** |
| --- | --- |
| *public interface RoomState {*  *void bookRoom(Room room);*  *void checkOut(Room room);*  *void markUnderMaintenance(Room room);*  *}*  *public class AvailableState implements RoomState {*  *public void bookRoom(Room room) {*  *room.setState(new BookedState()); // Transition to BookedState*  *System.out.println("Room booked.");*  *}*  *public void checkOut(Room room) {*  *System.out.println("Room is already available. No need to check out.");*  *}*  *public void markUnderMaintenance(Room room) {*  *room.setState(new UnderMaintenanceState()); // Transition to UnderMaintenanceState*  *System.out.println("Room marked under maintenance.");*  *}*  *}*  *public class BookedState implements RoomState {*  *public void bookRoom(Room room) {*  *System.out.println("Room is already booked.");*  *}*  *public void checkOut(Room room) {*  *room.setState(new AvailableState()); // Transition to AvailableState after checkout*  *System.out.println("Checked out of the room.");*  *}*  *public void markUnderMaintenance(Room room) {*  *room.setState(new UnderMaintenanceState()); // Transition to UnderMaintenanceState*  *System.out.println("Room marked under maintenance.");*  *}*  *}*  *public class UnderMaintenanceState implements RoomState {*  *public void bookRoom(Room room) {*  *System.out.println("Room is under maintenance. Cannot be booked.");*  *}*  *public void checkOut(Room room) {*  *System.out.println("Room is under maintenance. Cannot checkout.");*  *}*  *public void markUnderMaintenance(Room room) {*  *System.out.println("Room is already under maintenance.");*  *}*  *}*  *public class Room {*  *private RoomState state;*  *public Room() {*  *state = new AvailableState(); // Default state*  *}*  *public void setState(RoomState state) {*  *this.state = state;*  *}*  *public void bookRoom() {*  *state.bookRoom(this);*  *}*  *public void checkOut() {*  *state.checkOut(this);*  *}*  *public void markUnderMaintenance() {*  *state.markUnderMaintenance(this);*  *}*  *}* |  |

With this refactor, the **Room** class delegates state-specific behavior to the state objects, ensuring better maintainability, flexibility, and ease of testing. New states or transitions can be added without altering the core room management logic.

## Database Schemas

### Overview

The StayMate platform uses a centralized **relational database (MySQL)** to store and manage all data. The schema is organized around key entities that support core application features such as user management, bookings, payments, and notifications.

### Users

Stores information for both customers and administrators.

* **id**: Unique identifier
* **firstName**, **lastName**, **email**, **phoneNumber**: Personal details
* **role**: Defines the user's role (e.g., *Customer*, *Admin*)
* **isVerified**: Indicates if the user's email is verified

### Hotels

Holds hotel-specific data including location and geolocation.

* **hotelId**: Unique identifier
* **name**: Hotel name
* **address**, **city**, **state**, **country**: Location details
* **latitude**, **longitude**: Geolocation coordinates

### Room

Represents individual rooms available in each hotel.

* **roomId**: Unique identifier
* **hotelId**: Foreign key linking to the hotel
* **roomType**: Type of room (e.g., *Single*, *Double*, *Suite*)
* **price**: Nightly rate
* **status**: Current state (e.g., *Available*, *Booked*)

### Booking

Tracks user reservations for hotel rooms.

* **bookingId**: Unique identifier
* **userId**: Foreign key linking to the user
* **hotelId**, **roomId**: Identify the booked hotel and room
* **checkInDate**, **checkOutDate**: Stay duration
* **totalAmount**: Total price for the booking
* **status**: Booking state (e.g., *Confirmed*, *Pending*, *Cancelled*)

### Reviews

Captures feedback submitted by users about hotels.

* **reviewId**: Unique identifier
* **userId**: Reviewer
* **hotelId**: Reviewed hotel
* **rating**: Numeric score (e.g., 1–5)
* **comment**: User’s written feedback

### Payments

Records payment details for bookings.

* **paymentId**: Unique identifier
* **bookingId**: Related booking
* **paymentMethod**: Method used (e.g., *Credit Card*, *PayPal*)
* **amount**: Payment amount
* **transactionDate**: Timestamp of the transaction
* **status**: Payment result (e.g., *Success*, *Failed*)

### Payments

Manages messages sent to users about system events.

* **notificationId**: Unique identifier
* **userId**: Recipient
* **message**: Notification content
* **type**: Notification type (e.g., *Email*, *In-App*)
* **isRead**: Read status

### 

### Entity Relationships

| **Relationship** | **Type** | **Description** | **Implementation** |
| --- | --- | --- | --- |
| **Users ⇄ Booking** | One-to-Many (1:N) | A user can make multiple bookings, but each booking is tied to only one user. | **Booking.user\_id** is a foreign key referencing **Users.id**. |
| **Hotel ⇄ Room** | One-to-Many (1:N) | A hotel can contain many rooms, but each room belongs to only one hotel. | **Room.hotel\_id** references **Hotel.id**. Composite primary key (**hotel\_id**, **room\_id**) ensures room uniqueness within each hotel. |
| **Booking ⇄ Room (via Hotel)** | Many-to-One (M:1) | A booking is made for one specific room in a specific hotel. | **Booking.hotel\_id**, **Booking.room\_id** form a composite foreign key referencing Room(**hotel\_id, room\_id**). |
| **Users ⇄ Review ⇄ Hotel** | Many-to-Many (M:N) | A user can leave many reviews, and a hotel can have reviews from many users. | **Review.user\_id** references **Users.id**, **Review.hotel\_id** references **Hotel.id**. Review acts as a junction table holding review content and metadata. |
| **Booking ⇄ Payment** | One-to-One (1:1) | Each booking may have one associated payment record. | **Payment.booking\_id** references **Booking.id**. The Unique constraint on **booking\_id** ensures a one-to-one mapping. |
| **Users ⇄ Notification** | One-to-Many (1:N) | A user may receive many notifications, each tied to specific actions like bookings or promotions. | **Notification.user\_id** references Users.id. |
| **Admins ⇄ Hotels, Bookings, Users, Promotions** | Role-based access | Admins manage users, hotels, bookings, and promotions through application logic, not foreign key relationships. | Role-based access control is managed through the Users.role field rather than direct foreign key relationships. Admins have the authority based on their role. |

# 

# DevSecOps and Development Lifecycle

## Source Control Strategy

To ensure smooth collaboration, traceability, and secure integration of code changes, our team adopts a structured source control strategy centered around **Git** and **GitHub**. This strategy supports version control, branch management, and collaboration, enabling seamless development and deployment processes.

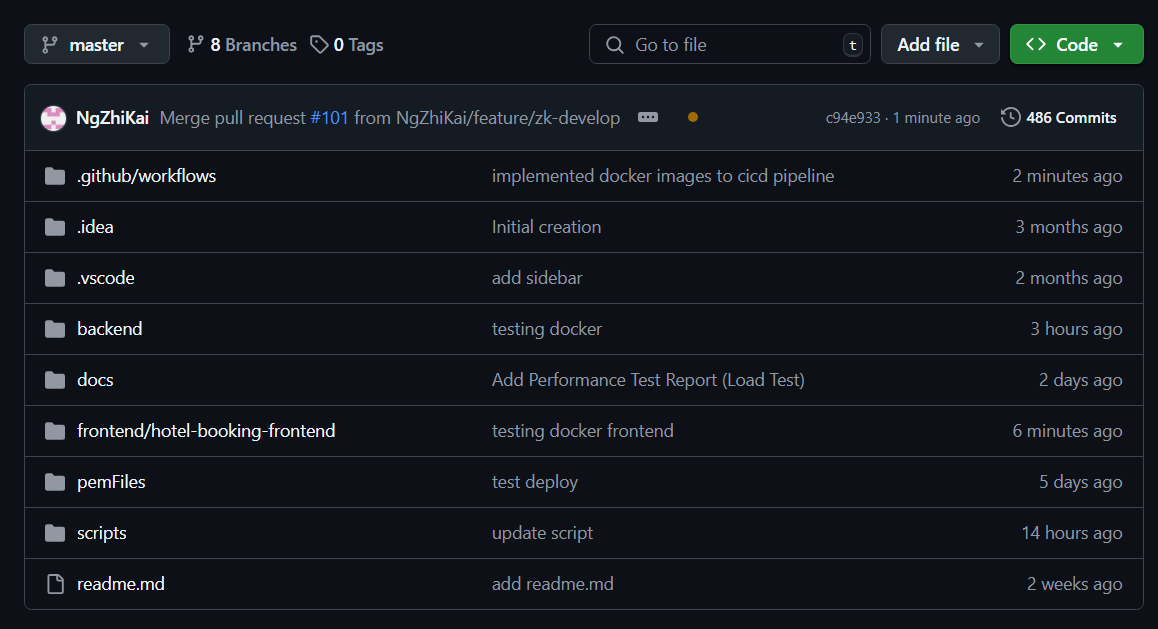
## Repository and Project Structure

The project is hosted on **GitHub** under a centralized repository that includes the following components:

* **Source Code**: All frontend and backend code, including business logic and components.
* **Documentation**: Design documents, project specifications, and API documentation.
* **Configuration Files**: Environment configurations, database setups, and CI/CD pipeline settings.
* **CI/CD Pipeline Definitions**: YAML files and scripts for continuous integration and deployment.

The repository follows a modular structure with clear separation of concerns:

* **/frontend**: React-based frontend code.
* **/backend**: Backend API server and business logic.
* **/docs**: Design and project documentation.
* **/scripts**: Utility and deployment scripts.



## Branching Strategy

To maintain a structured and organized development workflow, we follow a **feature-branch model**, allowing individual developers to work independently on different tasks while maintaining a stable **master branch**.

**Feature Branch:**

* Each developer works on their own feature branch, which is derived from the master branch.
* Branch names follow the convention **feature/<username-develop>**, e.g., **feature/zk-develop**.

**Master Branch:**

* The **master** branch is considered the stable version of the application.
* It always contains production-ready code and serves as the base for new development work.

**Pull Requests (PRs):**

* All changes must go through a **pull request** to be reviewed by at least one teammate before being merged into **master**.
* A pull request must pass automated tests and conflict resolution checks before being merged.

**Merging and Conflict Resolution:**

* Developers must pull the latest changes from **master** before starting work on their feature branch to minimize conflicts.
* If conflicts occur, developers are responsible for resolving them locally and testing the changes to ensure integration doesn’t break existing functionality.

| **Branch Strategy** | **Pull Request Process** |
| --- | --- |
|  |  |

## Authentication and Access Control

* **Authentication**: Managed via individual **GitHub** accounts. Each team member is required to authenticate through their GitHub credentials for access to the repository.
* **Access Control**: Only authorized team members have write access to the repository. Read access is granted to stakeholders.
* **Role-Based Permissions**: The repository uses **GitHub's role-based access control (RBAC)** for different levels of access for developers, project managers, and other stakeholders.

## Continuous Integration (CI)

### Developer Workflow

The **StayMate** project follows a well-defined Continuous Integration (CI) pipeline that ensures efficient code integration and deployment processes. This pipeline is automated and triggered whenever code is pushed to feature branches or when a pull request (PR) is created to merge changes into the master branch.

* **Push to Feature Branch**: Developers begin by creating feature branches and pushing their changes to GitHub. This allows for isolated work on specific features without impacting the main codebase.
* **Pull Request(PR)**: After completing a feature or change, developers create a pull request (PR) from their feature branch to the master branch. This PR is then reviewed to ensure code quality and consistency before merging into the main codebase.

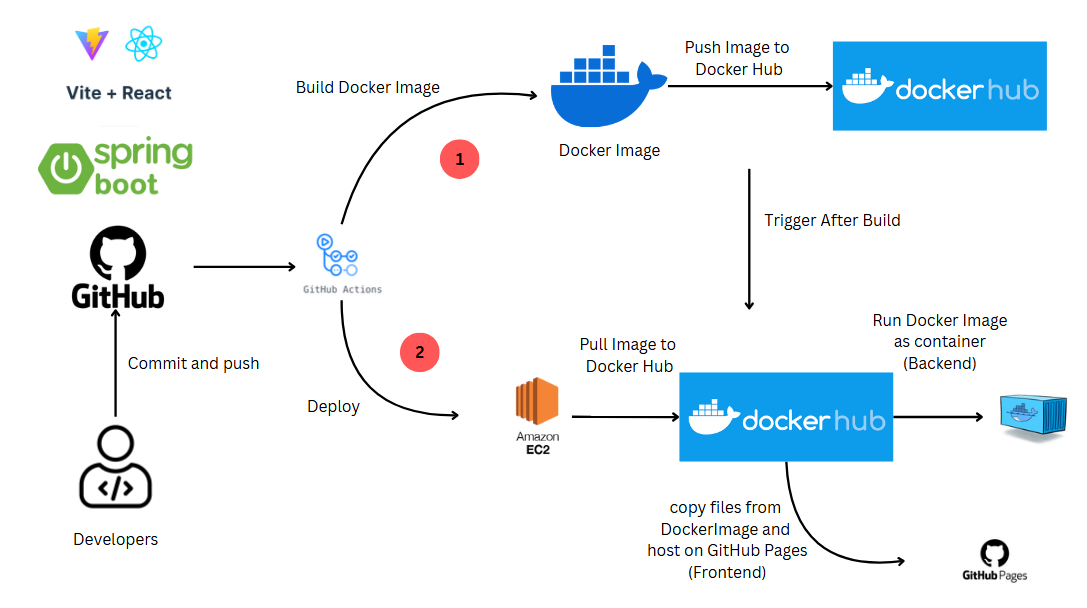
### Pipeline Jobs

The CI pipeline jobs are executed based on the folder that has been modified in the codebase, with specific tasks for both the backend and frontend.

* **Backend Changes Detected**:

When changes are detected in the backend folder, the following steps are executed:

* + **Unit Tests**: The pipeline runs JUnit tests to ensure the backend code functions correctly and adheres to test standards.
  + **Linting**: Linting tools are run on the backend code to check for and resolve any coding standard violations.
  + **Build & Packaging**: The backend code is built and packaged using **Maven**. This step ensures that the backend is ready for deployment.
  + **Dockerization**: The backend application is containerized into a Docker image, ensuring portability and consistency across environments.
  + **Deploy to EC2**: The packaged backend is deployed to a staging EC2 instance for further testing and validation.
* **Frontend Changes Detected**:  
  + **Linting**: Linting is run on the frontend React.js code to ensure consistency and adherence to coding standards.
  + **Build & Packaging**: The frontend code is bundled using **Webpack** or React scripts to prepare it for deployment.
  + **Dockerization**: The frontend application is containerized into a Docker image, ensuring consistent deployment across environments.
  + **Deploy to GitHub Pages**: Deploys the updated frontend code to GitHub Pages.
* **End-to-End Testing**:  
  + Once deployed, end-to-end tests validate the overall functionality of the app (UI interactions, booking flow, user authentication, and payment processing).

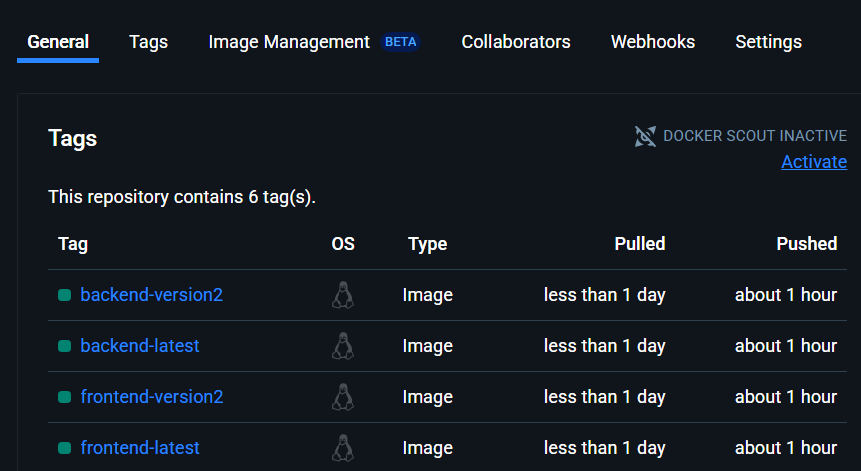


## Continuous Delivery (CD)

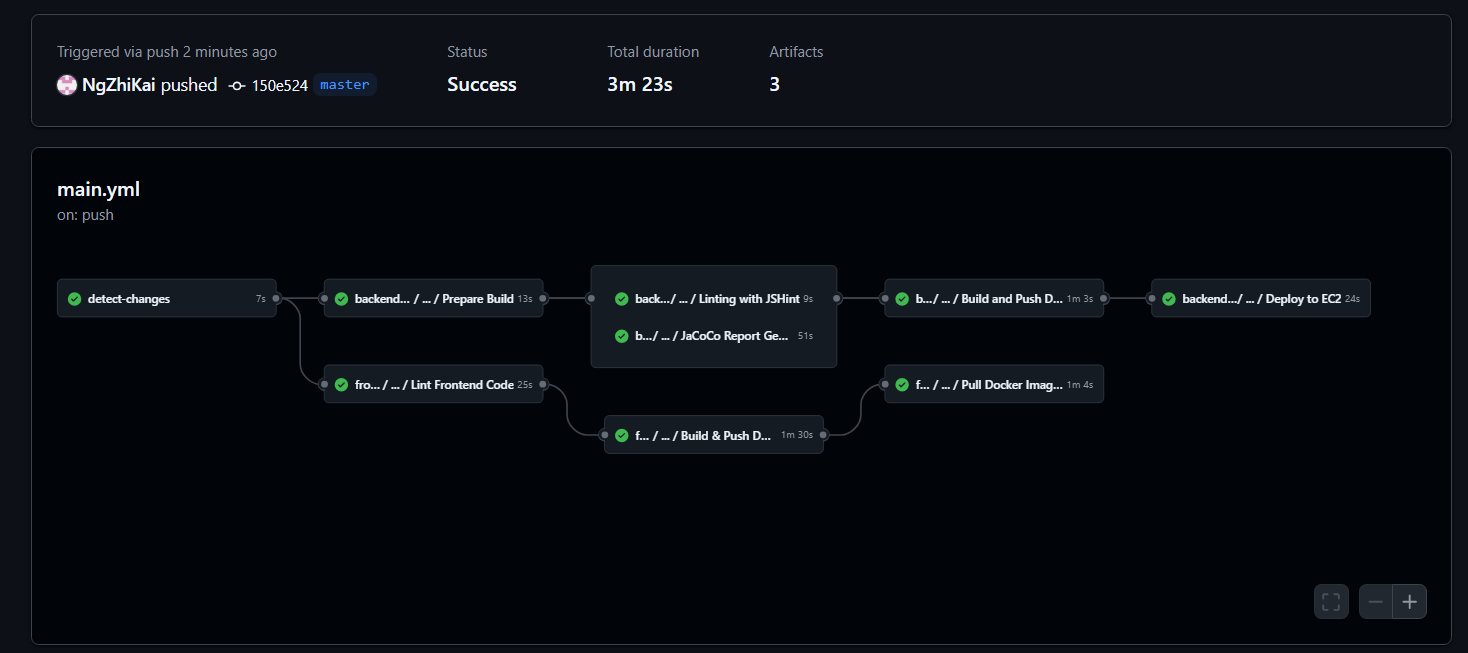
### Deployment Workflow

The StayMate project uses an automated deployment pipeline to streamline the release of application updates:

* **Code Merge Trigger**: When changes are merged into the master branch, the CD pipeline initiates automatically.
* **Build & Package**: Backend and frontend code are built and packaged using Maven and Webpack respectively.
* **Dockerization**: Both backend and frontend are containerized using Docker to ensure consistent deployments.



* **Artifact Delivery**:  
  + Backend Docker images are prepared for deployment.
  + Frontend static files are bundled and readied for GitHub Pages.
* **Deployment**:  
  + The **frontend** is deployed to **GitHub Pages** for immediate user access.
  + The **backend** is deployed to a **EC2 instance** for further testing.



This process ensures that every change passing CI is quickly and safely delivered to users or testable by the team.

### Environment

* **Development Environment**
  + **Purpose**: Facilitates active development and internal testing.
  + **Usage**:  
    - Developers commit changes to feature branches, which trigger Continuous Integration (CI) jobs for automated testing and validation.
    - Upon successful review and merging into the **master** branch, the development environment is automatically updated.
    - Connected to a testing database populated with mock data to support realistic simulations without affecting production integrity.
* **Production Environment**
  + **Purpose**: Hosts the live application accessed by end users.
  + **Usage**:  
    - Changes are deployed only after thorough testing and manual approval in the development environment.
    - Only stable, production-ready code is released to maintain reliability.
    - Connected to live production data, with strict security, access controls, and data integrity safeguards in place.

# Performance Testing

To ensure the system can support real-world usage and maintain reliability under load, we conducted performance testing using **Apache JMeter**. The following test is performed:

| **Test Type** | **Concurrent Users** | **Duration** | **Purpose** |
| --- | --- | --- | --- |
| Load Test | 50 users | 40 Mins | To assess normal expected traffic performance. |

## 

## Target Metrics

* Average response time under load (50 users): **≤ 3000 ms**
* Login operations under load: **≤ 5000 ms**

### Infrastructure Considerations

The system is hosted on **cost-effective, free-tier platforms**, consisting of:

* **Frontend**: Deployed on GitHub Pages (static hosting)
* **Backend & API**: Hosted on a low-cost compute instance
* **Database**: AWS RDS (Free Tier), 1 GB RAM, 20 GB storage, max ~60 concurrent connections

Due to these **resource constraints**, we have adjusted the number of virtual users expectations accordingly. The following factors contribute to the increased latency under higher load:

* Limited **database connection pool size**
* **Shared CPU** resources leading to degraded performance under high concurrent load
* Lack of horizontal scaling or caching layers in free-tier environments

Due to these limitations, we expect some number of failed cases as a result of the infrastructure specifications; however, the system remains viable for small-scale deployments such as early-stage testing, demos, or MVP use cases.

## Business Processes

| **Business Processes (BPs)** | **Percentage (%) of Total VUs** |
| --- | --- |
| View Hotel (Non-Login) | 40 |
| View Hotel (Login) | 30 |
| View Notifications (Users) | 10 |
| Book Hotel (Users) | 20 |

## Load Testing

**Objective**: Evaluate the system’s ability to handle expected user traffic.

**Setup**:

* **Number of Virtual Users (VUs)**: 50
* **Ramp-Up Time**: 5 Mins
* **Ramp-Down Time**: 5 Mins
* **Hold-Time**: 30 Mins
* **Think Time**: 5 Seconds
* **Interval Time (Time between execution of the next iteration):** 10 Seconds
* **Targeted Scenarios**:  
  + All BPs

**Expected Outcome**:

* Response time for general operations remains **below 3000 ms**
* Response time for login remains **below 5000ms**

## Load Testing (Results)

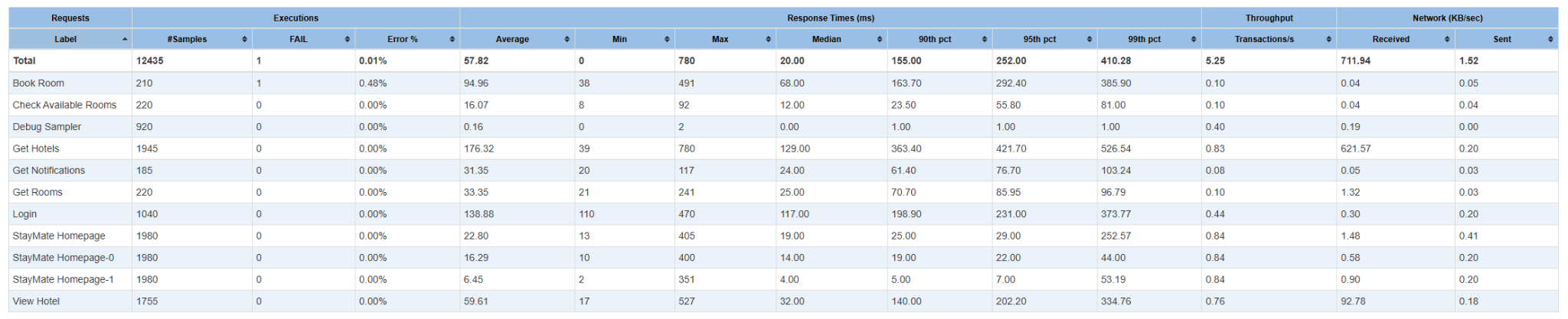
## 

### Summary

## 

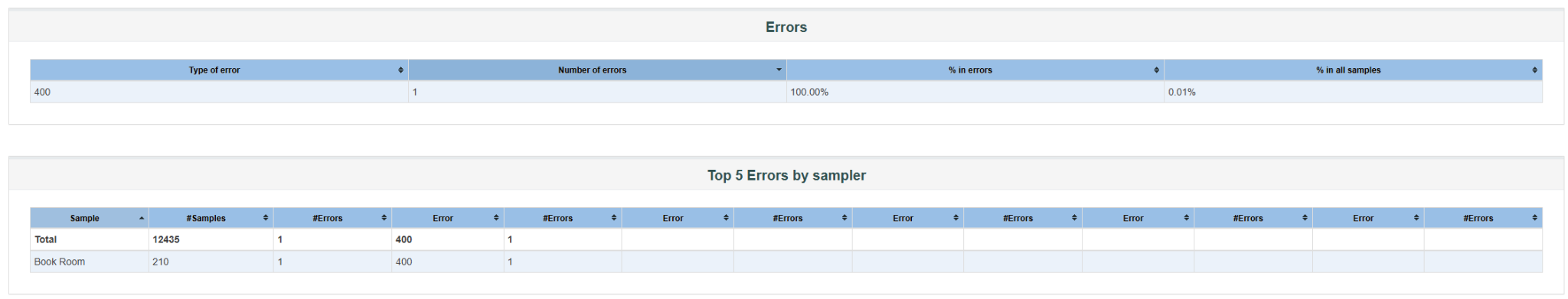
The performance testing conducted using JMeter demonstrated that the system performs reliably under normal load conditions. Out of all requests made during the test sessions, 99.99% were successfully handled within the expected response time thresholds, with only 0.01% failed. These results indicate a high level of stability and responsiveness, suitable for small to moderate user loads.

### Statistics



The JMeter performance report yielded a total of 12,435 samples, out of which only 1 request failed, resulting in an error rate of just 0.01%. The system demonstrated excellent responsiveness with an average response time of **57.82 ms**, a minimum of **2 ms**, and a maximum of **780 ms**. The **median response time** was **20 ms**, while the **90th, 95th, and 99th percentiles** were **155 ms**, **252 ms**, and **410.28 ms** respectively, indicating strong consistency even under load. The system also maintained a steady throughput of **5.25 transactions per second**, confirming its ability to handle concurrent users efficiently within the limits of the hosting infrastructure.

### Errors

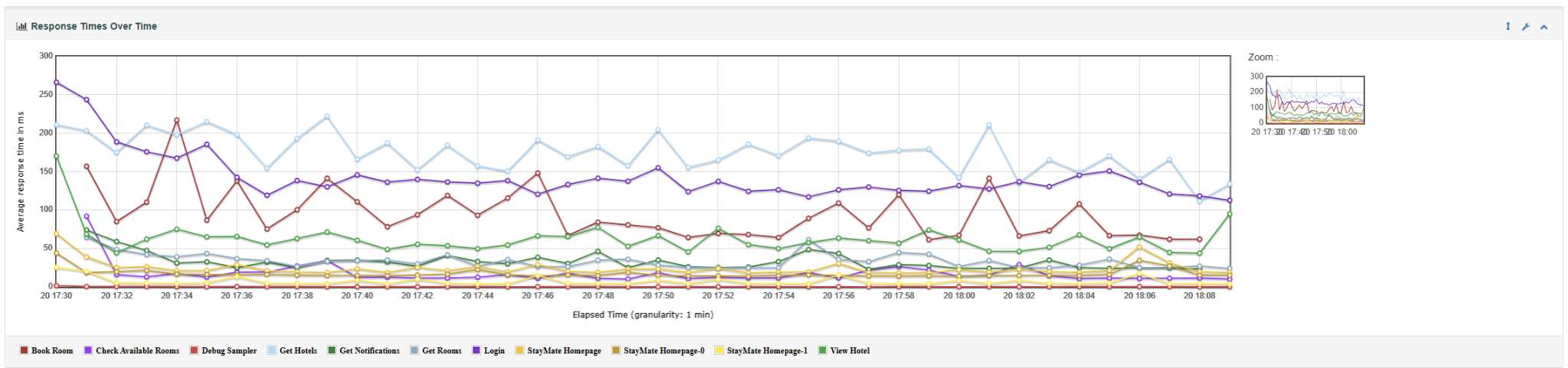


During the performance testing, only **1 error** was encountered out of **12,435 samples**, resulting in a **0.01% error rate**. The error recorded was an **HTTP 400 Bad Request**. This type of error typically indicates that the request sent to the server was malformed or invalid. Possible reasons for this could include:

* **Improperly formatted parameters** in the request (e.g., missing required fields or invalid JSON syntax).
* **Encoding issues** during high concurrency, causing occasional malformed requests.
* **Timing-related glitches** where a request might have been sent before the system was fully ready to process it (e.g., during ramp-up).
* **Manual configuration error** in one of the virtual users or JMeter threads that generated a bad payload.

Given the extremely low occurrence rate, the error appears to be an isolated case and not indicative of systemic issues. However, it highlights the importance of input validation and robust error handling in real-world scenarios.

### Response Time over Time Graph



The response time trends observed throughout the test demonstrate **strong system performance**, with the **majority of business processes averaging between 50 ms to 100 ms**. This indicates that the system is well-optimized for responsiveness, even under sustained load.

Notably, the operations with **higher response times** include:

* **Login** – due to authentication checks.
* **Get Hotels** – as it involves fetching a list of all hotels and its data.
* **Book Room** – involving availability checks, data consistency, and database transactions.

Despite these naturally heavier processes, their response times remained well within acceptable thresholds, with spikes still comfortably below the maximum acceptable latency (sub-3000 ms). This confirms that the system can efficiently handle real-world user interactions under moderate to high load.

# Conclusion

The Staymate application reflects a well-rounded and disciplined effort to design and implement a modern, full-stack hotel booking system that addresses real-world business needs while upholding strong software engineering principles. Starting with a clear understanding of requirements and stakeholder expectations, the team advanced through structured planning, in-depth system design, and consistent implementation.

The system architecture adopts a layered structure, ensuring clean separation of concerns across the frontend, backend, and data access layers. Core design patterns—Strategy, Factory, Observer, and State—were purposefully employed to resolve design complexities and enhance maintainability and scalability. Key external integrations, such as a mock payment gateway and email service, contribute to a realistic and production-ready system experience.

Comprehensive modeling with use cases and sequence diagrams, combined with DevSecOps practices like CI/CD pipelines and automated testing, demonstrate the team’s proficiency with modern development workflows. Performance evaluations and metrics tracking further confirmed the system’s reliability and efficiency under load.

Looking forward, future iterations could explore enhanced analytics, real-time communications, and deeper third-party service integration. Overall, Staymate stands as a robust, secure, and extensible solution that delivers meaningful value to users while showcasing best practices in full-stack software development.

# 

# Appendix A: Static Application Security Testing (SAST) Results

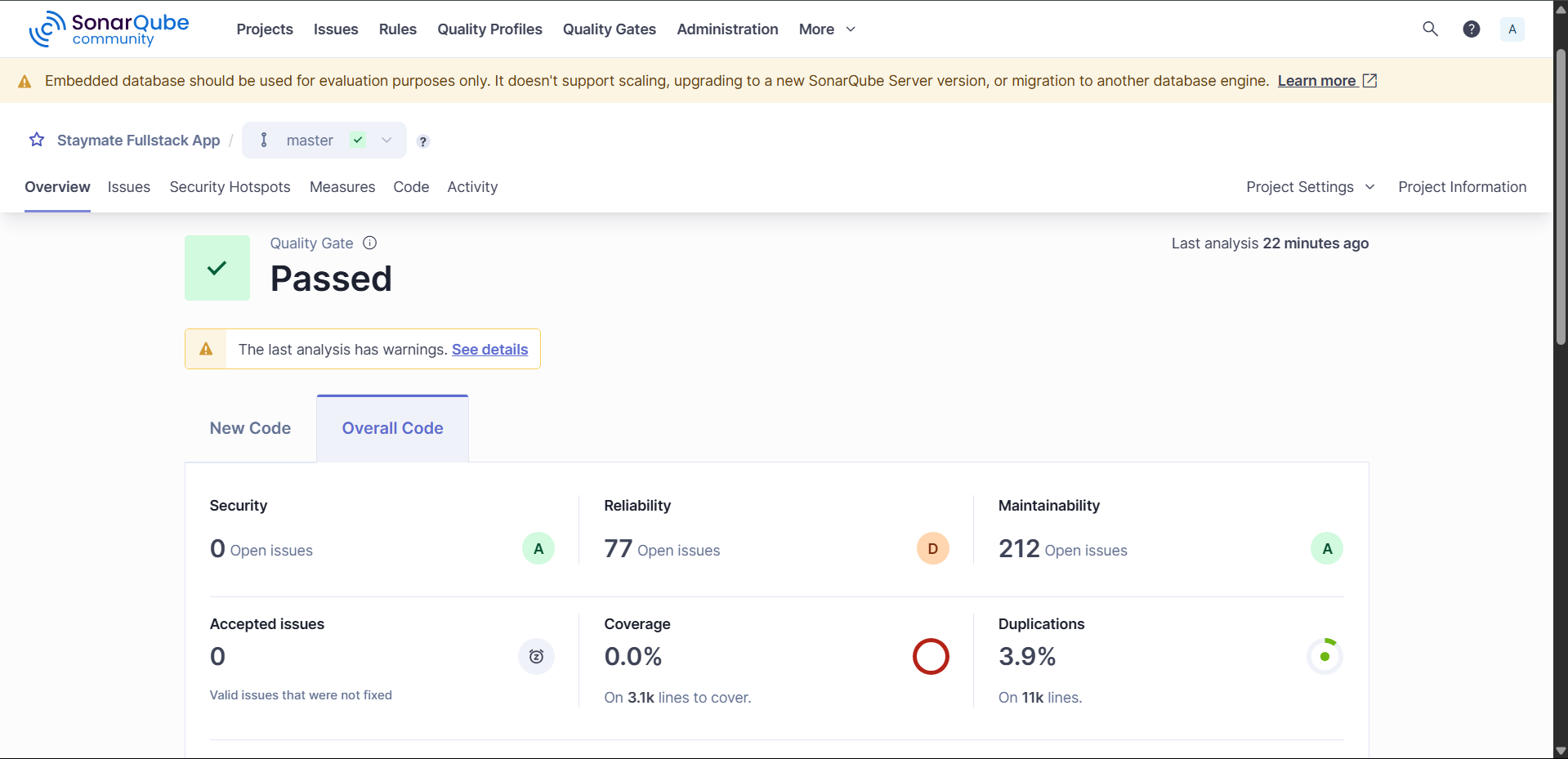
**Tool Used:** SonarQube

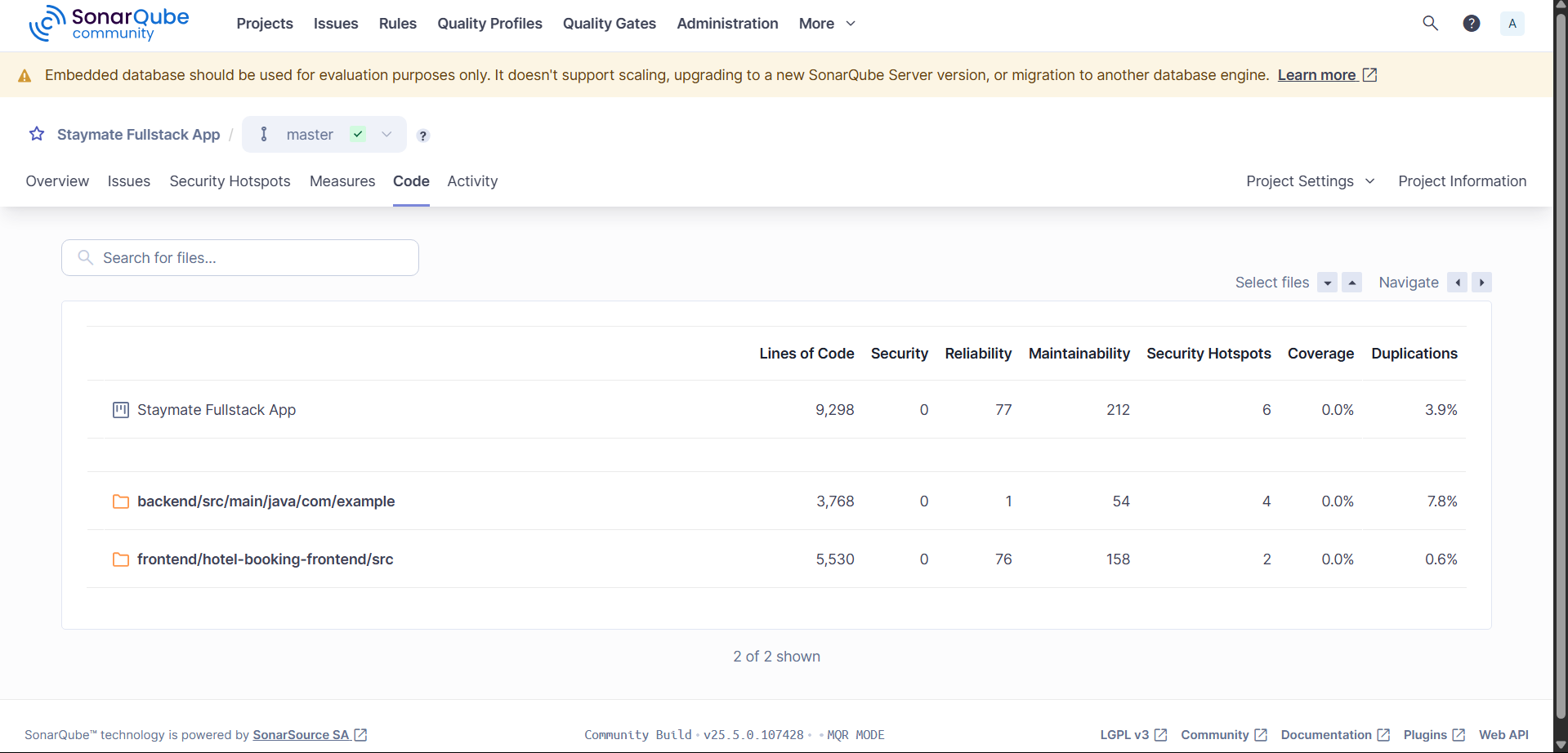
**Scan Date:** May 10, 2025

**Scope:** Full project source code

**Summary:**

| **Risk Level** | **Count** |
| --- | --- |
| Blocker | 0 |
| High | 19 |
| Medium | 144 |
| Low | 126 |
| Info | 0 |

****

****

# 

# Appendix B: Dynamic Application Security Testing (DAST) – OWASP ZAP Results

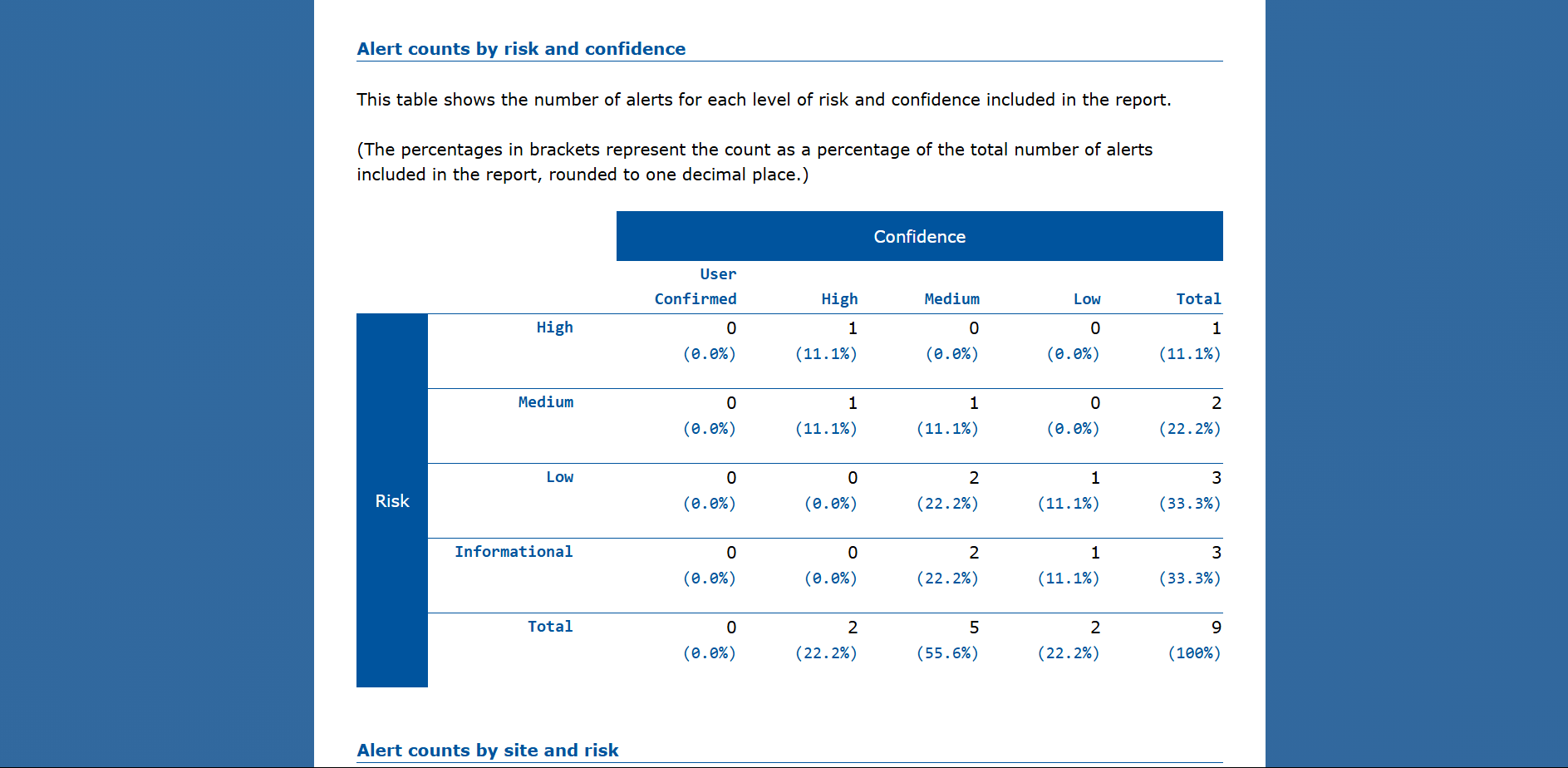
**Scan Date:** May 10, 2025

**Target Application:** https://ngzhikai.github.io/stayMate/

**Scan Type:** Spider

**Summary:**

| **Risk Level** | **Count** |
| --- | --- |
| High | 1 |
| Medium | 2 |
| Low | 3 |
| Informational | 3 |



# 

# 

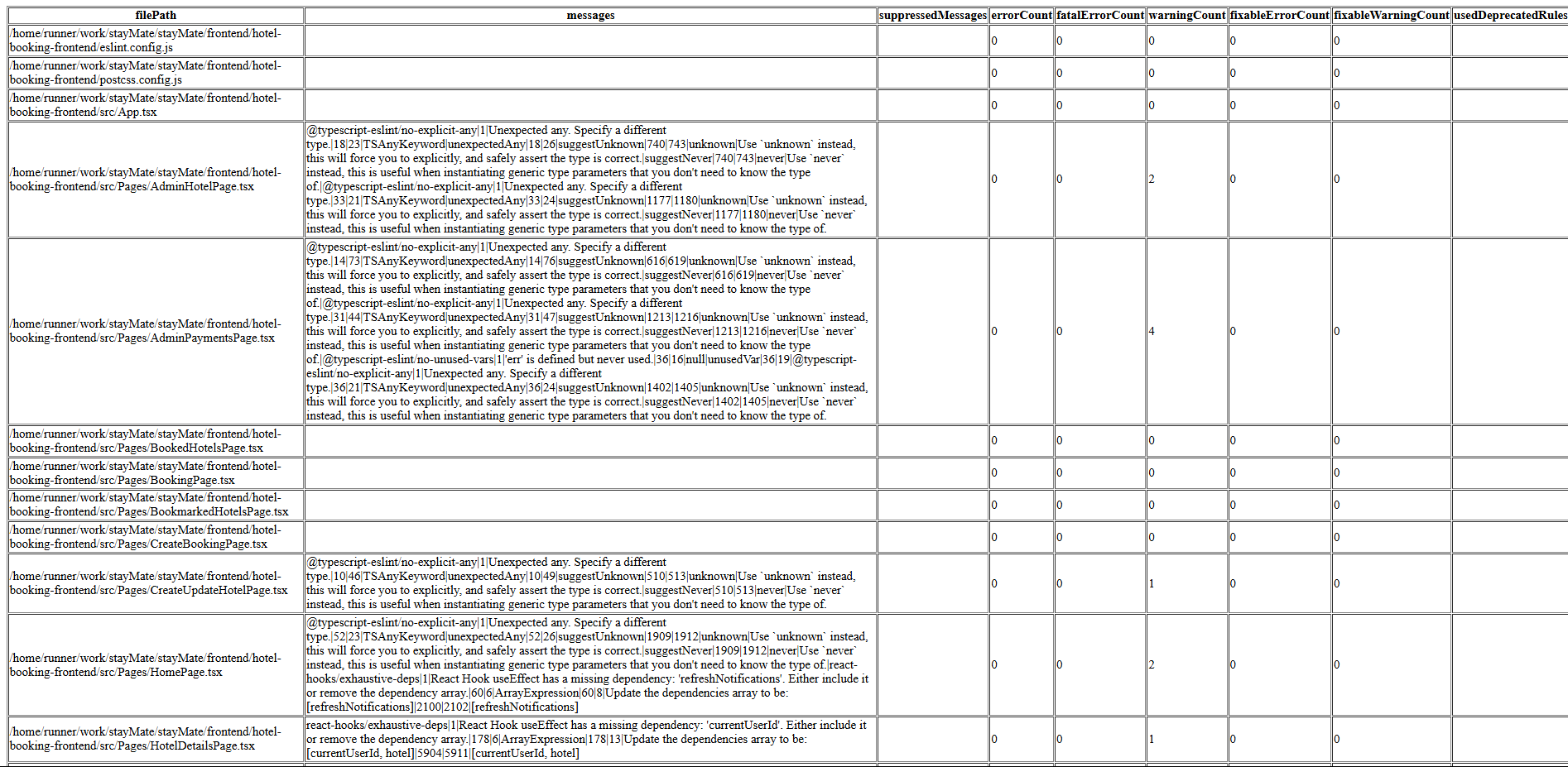
# Appendix C: Linting Report

**Tool Used:** ESLint

**Scan Date:** May 10, 2025

**Summary:**

| **File** | **Error Count** | **Warning Count** |
| --- | --- | --- |
| **Pages/AdminHotelPage.tsx** | 0 | 2 |
| **Pages/AdminPaymentsPage.tsx** | 0 | 4 |
| **services/paymentApi.ts** | 0 | 7 |
| **Pages/HomePage.tsx** | 0 | 2 |

****

### 

# 

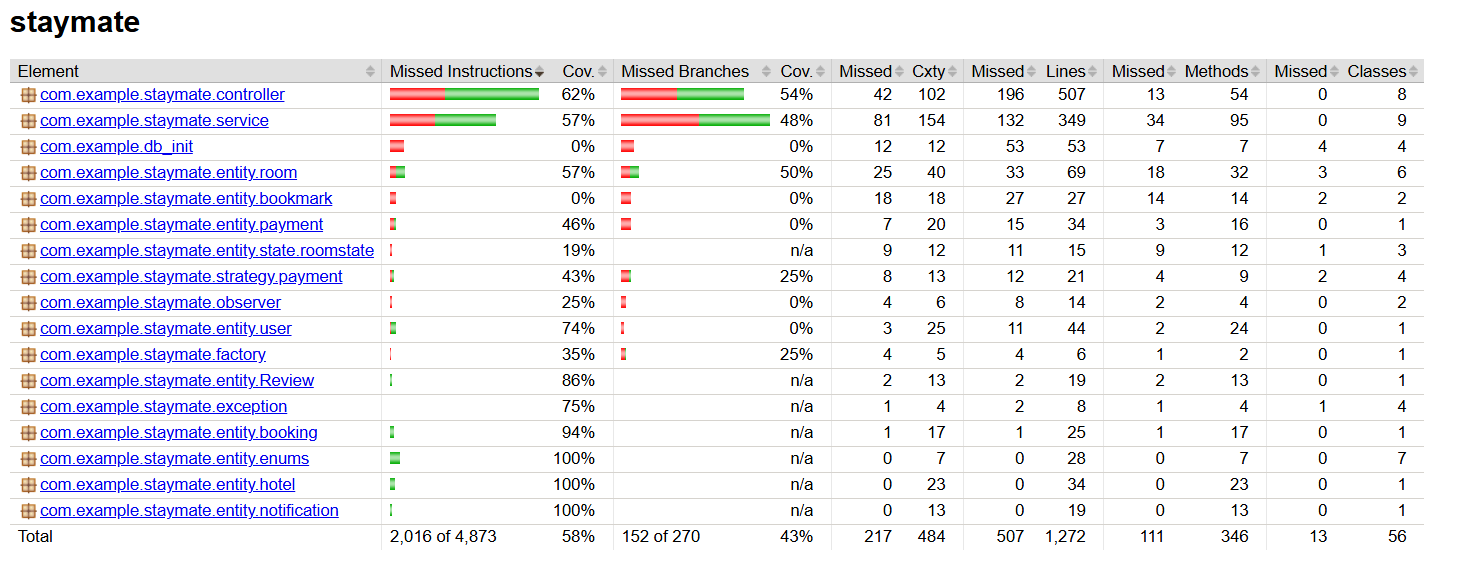
# Appendix D: JaCoCo Code Coverage Report

**Tool Used:** JaCoCo

**Scan Date:** May 5, 2025

**Summary:**

| **Metric** | **Covered** | **Missed** | **Coverage(%)** |
| --- | --- | --- | --- |
| Instructions | 2857 | 2016 | 58 |
| Branches | 118 | 152 | 43 |
| Lines | 765 | 507 | 60.14 |
| Methods | 235 | 111 | 67.91 |
| Classes | 43 | 13 | 76.79 |



# 

# Appendix E: Team Members Contributions

[Effort Expended per Team Member (Estimated Man-Hours)](#_heading=h.jnsugparupp7)

| **Team Member** | **Total Effort (Man-Hours)** | **Key Contributions** |
| --- | --- | --- |
| **Zhi Kai** | 200 | Backend development, Frontend development, API integration, CI/CD setup, deployment, system design, testing |
| **Ju Khang** | 100 | Requirements gathering, Frontend development, System design, System Testing, Performance Testing |
| **Jonathan** | 80 | Enhancement of functionality with testing. Fixing of bugs. Frontend development. |
| **Jun Heng** | 100 | Setting up of project, frontend development, enhancement of functionality with testing, fixing of bugs. |
| **Phillip** | 80 | Integrating testing framework, system testing, writing unit testing for API. |

### [Work Breakdown Structure](#_heading=h.7tv9wjiqw1by)

| **Phase** | **Task** | **Estimated Effort (man-days)** | **Done By** |
| --- | --- | --- | --- |
| **Project Planning and Initiation**  **(1 week)** | Define project scope and objectives | 1 | All |
|  | Set up project tools and communication channels | 1 | All |
|  | Develop project schedule and resource allocation | 1 | All |
| **Requirements Gathering and Analysis (1 week)** | Document functional and non-functional requirements | 2 | Ju Khang |
|  | Define user stories and acceptance criteria | 2 | Ju Khang |
|  | Finalize project requirements | 2 | Ju Khang |
| **System Architecture and Design (1 week)** | Define system architecture design | 2 | All |
|  | Create detailed technical design | 3 | Ju Khang, Zhi Kai |
|  | Design database schema and data flow | 2 | Zhi Kai |
| **Frontend Development**  **(2 weeks)** | Set up frontend environment and tools | 2 | Jun Heng, Ju Khang |
|  | Develop core UI components | 10 | Ju Khang |
|  | Integrate frontend with backend APIs | 8 | Zhi Kai |
|  | Conduct frontend unit and integration testing | 6 | Ju Khang, Zhi Kai |
| **Backend Development (2 weeks)** | Set up backend environment and tools | 2 | Zhi Kai |
|  | Develop core backend functionality | 10 | Zhi Kai |
|  | Implement API endpoints | 8 | Zhi Kai |
|  | Integrate third-party services | 6 | Zhi Kai |
|  | Implement business logic and data validation | 6 | Zhi Kai  Jonathan |
| **Testing and Quality Assurance (2 weeks)** | Develop test cases for all requirements | 3 | Ju Khang, Zhi Kai |
|  | Conduct manual testing | 8 | Ju Khang, Zhi Kai |
|  | Automate tests | 10 | Ju Khang, Zhi Kai |
|  | Perform load and performance testing | 6 | Ju Khang |
|  | Bug fixing and validation after testing | 6 | Zhi Kai Jonathan |
| **DevOps and Deployment**  **(1 week)** | Set up CI/CD pipeline | 5 | Zhi Kai |
|  | Deploy application to production | 3 | Zhi Kai |

# Appendix F: Progress Reports

# Appendix G: Links

GitHub Repo - <https://github.com/NgZhiKai/stayMate>

StayMate Application Page - <https://ngzhikai.github.io/stayMate/>