Team-13 : PitchPlease

Team Members :

- Chirag Dhamija : 2022101039
- Tejas Cavale : 2022101087
- Aditya Mishra : 2022101047
- Jay Mupiddi : 2021101035
- Hardik Kalia : 2022101092

PitchPlease is a platform designed to streamline the discovery and booking of sports facilities for end users, while empowering facility providers to manage their listings, schedules, and pricing with ease. The system is built with a microservices architecture and adheres to best practices in scalability, modularity, and security. The platform supports end-to-end workflows including registration, facility search, booking coordination, secure payments, and real-time availability checks—ensuring a smooth and reliable user experience for both customers and providers.

Tech Stack

- **Backend**: Java (Spring Boot)
- **Frontend**: Vanilla HTML, CSS, JavaScript
- **Database**: PostgreSQL

Functional Requirements

Below listed are the functional requirements of our system which defines what our system must do :

- Users must be able to register, log in/out, and manage profiles.
- Users should be able to reset passwords and manage account settings.
- User activity (e.g., bookings made, listings added) must be tracked.
- Providers can create, update, and manage their facilities.
- Support for reviews, ratings, and basic personalized recommendations.

- Customers must be able to search facilities using filters (location, sport type, rating).
- Users can check availability of facilities and make or cancel bookings.
- Booking history should be retrievable per user (both provider and customer).
- Secure payment processing must be supported via cards/wallets.
- Refunds and cancellation charges must be handled based on booking status.
- Invoices/receipts must be generated after each transaction.

Non Functional Requirements

Below listed are the non functional requirements of our system which define how well the system will perform under some constraints:

- Performance
  - Facility search should return results in reasonable time (high availability demand)
  - Booking confirmation (including payment) should complete within few seconds.
- Scalability
  - The system must handle growing numbers of users and facilities without performance degradation.
  - Services should support horizontal scaling independently (e.g., spike in bookings should not affect user login)
- Availability & Reliability
  - System uptime should be high.
  - Booking and payment services must support retry mechanisms and idempotency for fault tolerance.
- Security
  - All sensitive information such as password must be encrypted (at rest and in transit).
  - Token-based authentication must be enforced.

- Maintainability
  - Each microservice is independently deployable
  - Versions are clearly mentioned in the dependency files for all the microservices.
- Usability :
  - Users should have a smooth experience even during failures (e.g., retry options on payment failure).
  - Actions like login, booking, and browsing should require minimal steps and provide helpful feedback.
- Fault Tolerance :
  - The system should handle temporary failures gracefully without crashing or losing user data.
  - Critical operations like payment should support retry mechanisms or allow users to retry without re-entering details.

Architectural Significant Requirements

These are the subset of functional and non functional requirements which will have a strong impact on system architecture . These are listed below :

1. **Workflow Coordination for Booking and Payment** : Booking and payment span multiple services and must be coordinated reliably. If payment succeeds but booking fails (or vice versa), the system must handle rollback or compensation.
2. **Idempotency in Booking and Payment** : Users might retry due to network failure or UI issues. Reprocessing should not cause double bookings or duplicate charges.
3. **High Performance and Low Latency Search** : Efficient search is critical to user experience and directly impacts response time and system load.
4. **Data Security (Encryption + Access Control)** : Payment data and user PII require compliance with standards.
5. **Deployability and Maintainability** : Enables team autonomy, easier rollbacks, and service-specific scaling.

6. **Scalability Across User and Facility Growth** : Future growth in users or providers must not degrade performance or availability.
7. **User Activity Tracking** : Tracking needs a cross-service logging strategy or event stream, as actions happen across multiple domains.

Subsystems :

Based on the all the functional requirements we have 4 distinct subsystems in our system which are mainly :

1. User Subsystem

The User Subsystem is responsible for managing user identities and access control within the application. It provides functionalities such as user registration, login, logout, password reset, profile updates, and session tracking. Additionally, it tracks user activity to support audit logging or analytics. This subsystem serves purely as the authentication and identity management layer of the platform.

2. Facility Subsystem

The Facility Subsystem handles the full lifecycle of sports facility management and discovery. For providers, it offers tools to create and manage facility listings, including, specifying amenities, and configuring pricing plans with calendar-based availability. For customers, it supports discovery features such as searching and filtering facilities by location, sport type, and rating. It also incorporates reviews and basic recommendation capabilities. This subsystem is solely focused on managing facility data.

3. Booking Subsystem

The Booking Subsystem is in charge of managing the reservation workflow for facilities. It coordinates with the Facility Subsystem to check real-time availability and allows users to create or cancel bookings accordingly. It maintains a comprehensive history of bookings for both customers and providers and handles notifications related to booking status updates. While it plays a crucial role in scheduling, and is designed to operate independently with clear API contracts.

4. Payment Subsystem

The Payment Subsystem is responsible for all financial operations associated with bookings. It facilitates secure transaction processing through multiple payment options like cards or digital wallets, manages refunds and cancellation fees, and generates invoices and receipts for completed transactions. The subsystem ensures the protection of sensitive payment data using encryption and adheres to security standards and compliance practices. It focuses strictly on financial interactions.

Stakeholder Identification :

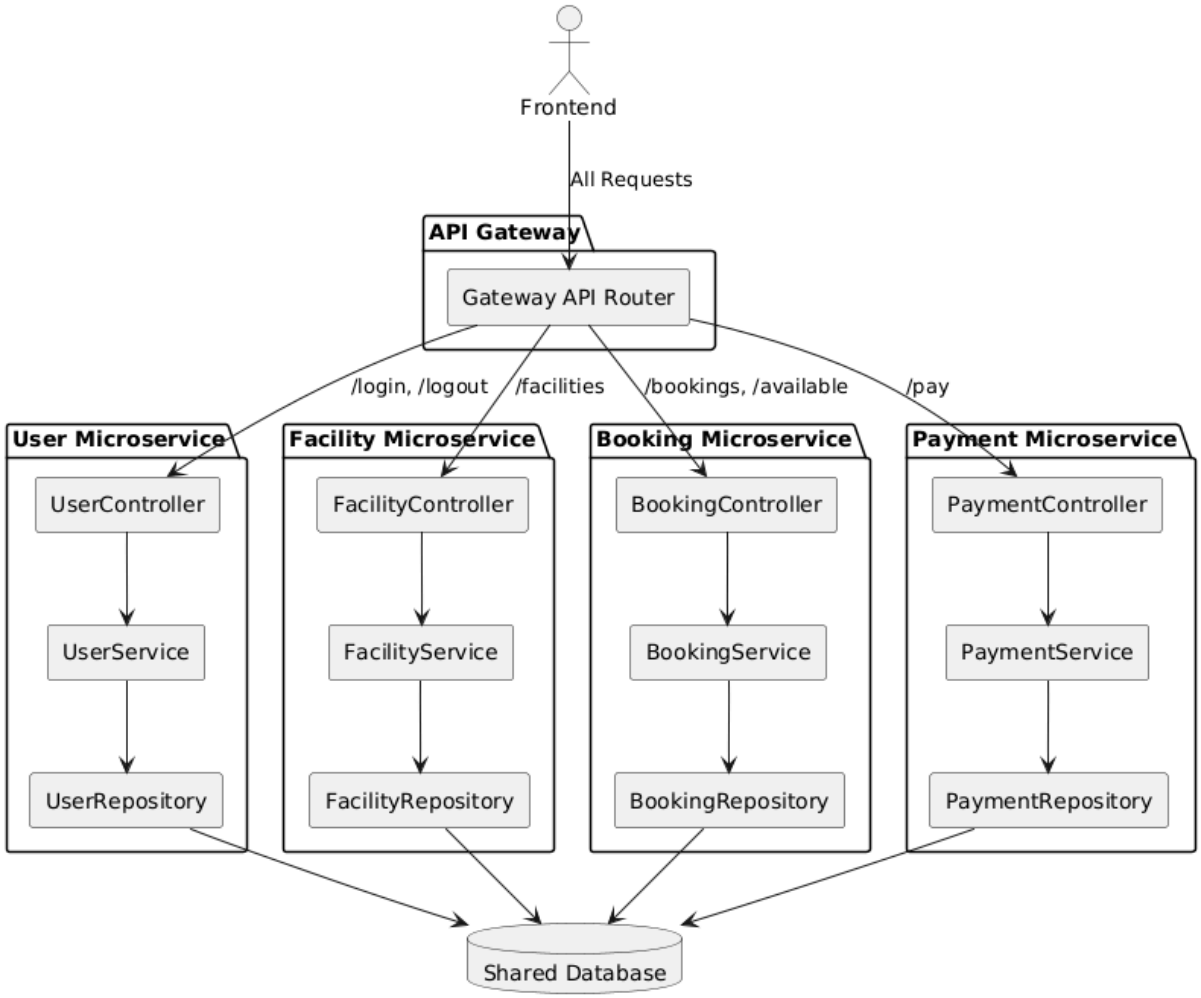
Below is the table listing relevant stakeholders and their primary concerns and viewpoints with PitchPlease as a their entity of interest.

Stakeholder	Role & Justification	Primary Concerns	Relevant Viewpoints
System Architect(s)	Responsible for overall system structure, service decomposition, and NFR compliance	Clean modular design, Microservices , separation, High availability, Idempotency handling, Scalability and reliability	Architecture Overview View, Deployment View
Frontend Developer(s)	Builds user-facing interfaces for booking, registration, search, etc.	Responsive and intuitive UI, Efficient API consumption, Error feedback, Secure login/logout, Easy booking flow	User Interface View, Interaction Flow View
Backend Developer(s)	Implements business logic, APIs, database models, and service coordination	API consistency, Secure authentication, Retry and rollback for booking/payment, Service orchestration, Logging and activity tracking	Service Interaction View, API Design View, Operational View
End Users (Customers)	Real users who may use the platform to find and book sports facilities	Easy registration and login, Search with filters, Smooth booking flow, View booking history, Secure and quick payments	User Experience View, Booking Workflow View

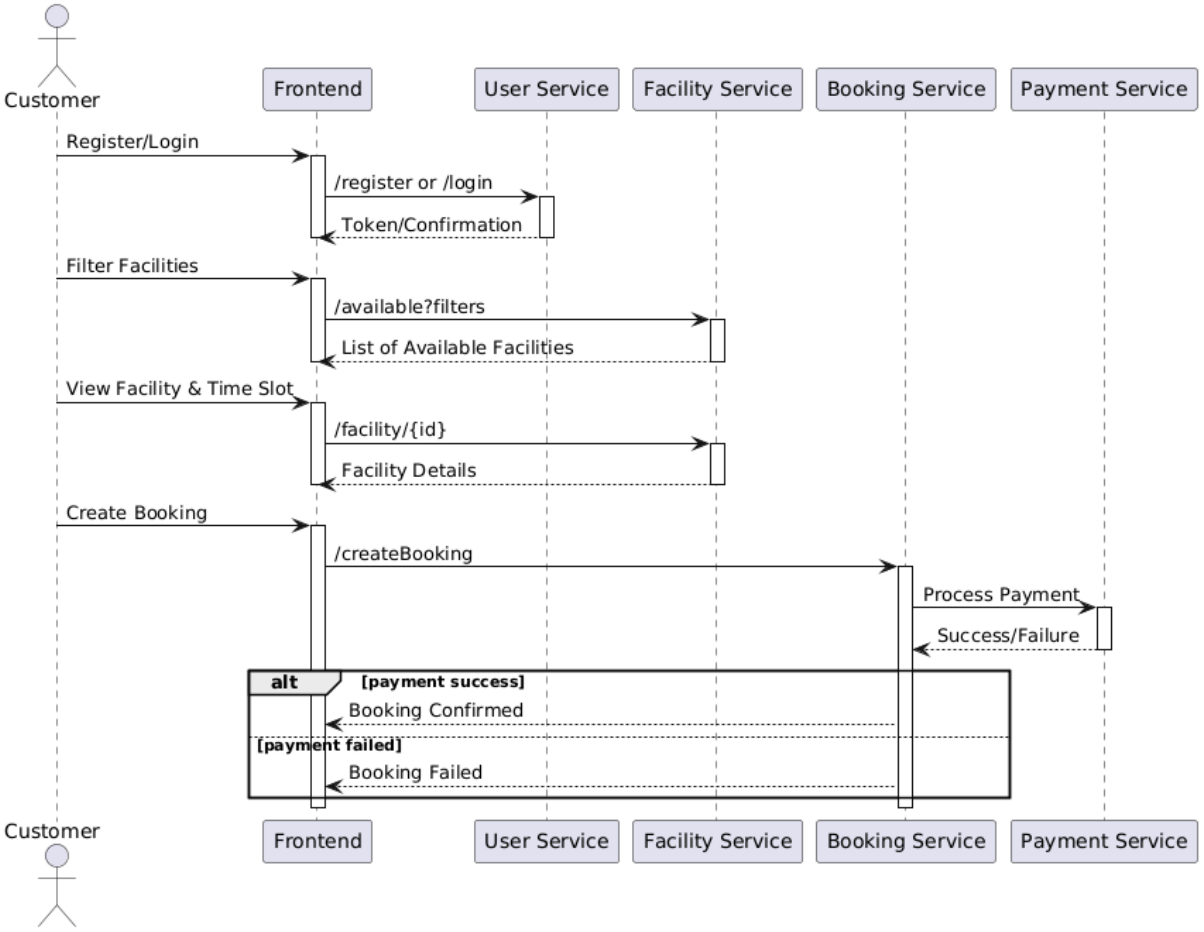
End Users (Providers)	Real facility owners or managers who list and manage their sports venues	Facility listing interface, Upload photos, set pricing and calendar, View bookings, See reviews and ratings	Facility Management View
Payment Gateway Provider	Integrated Payment Service	Reliable and secure API, Idempotent transactions, Clear refund/cancellation integration	Integration View, Payment Flow View

Major Stakeholder Views

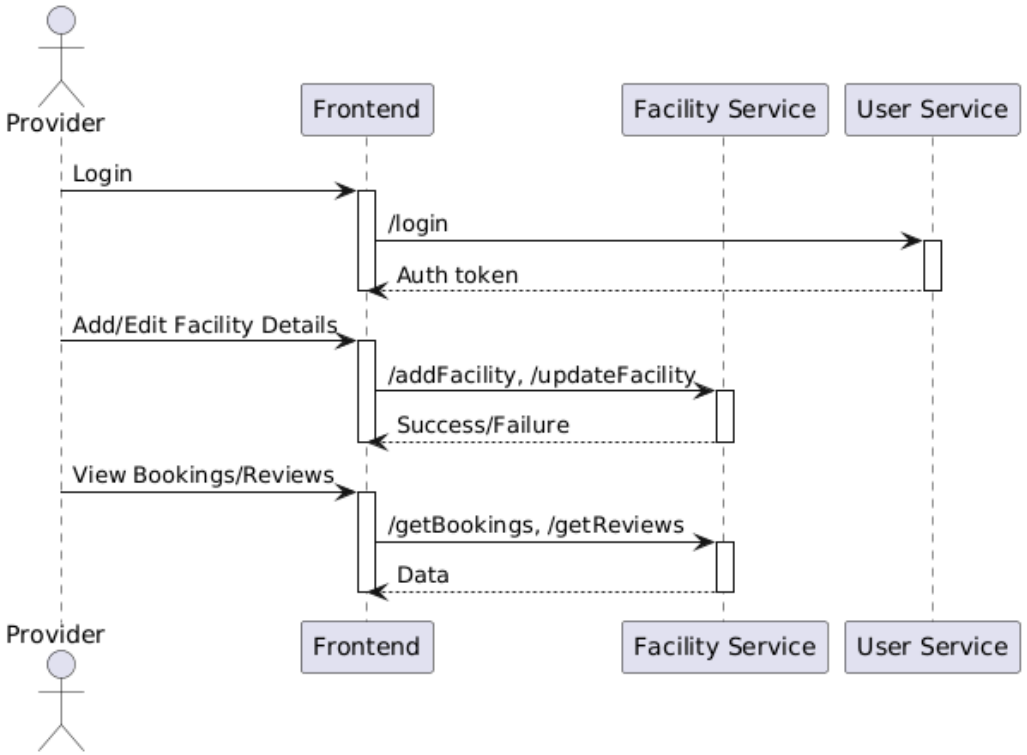
1. Architecture Overview View



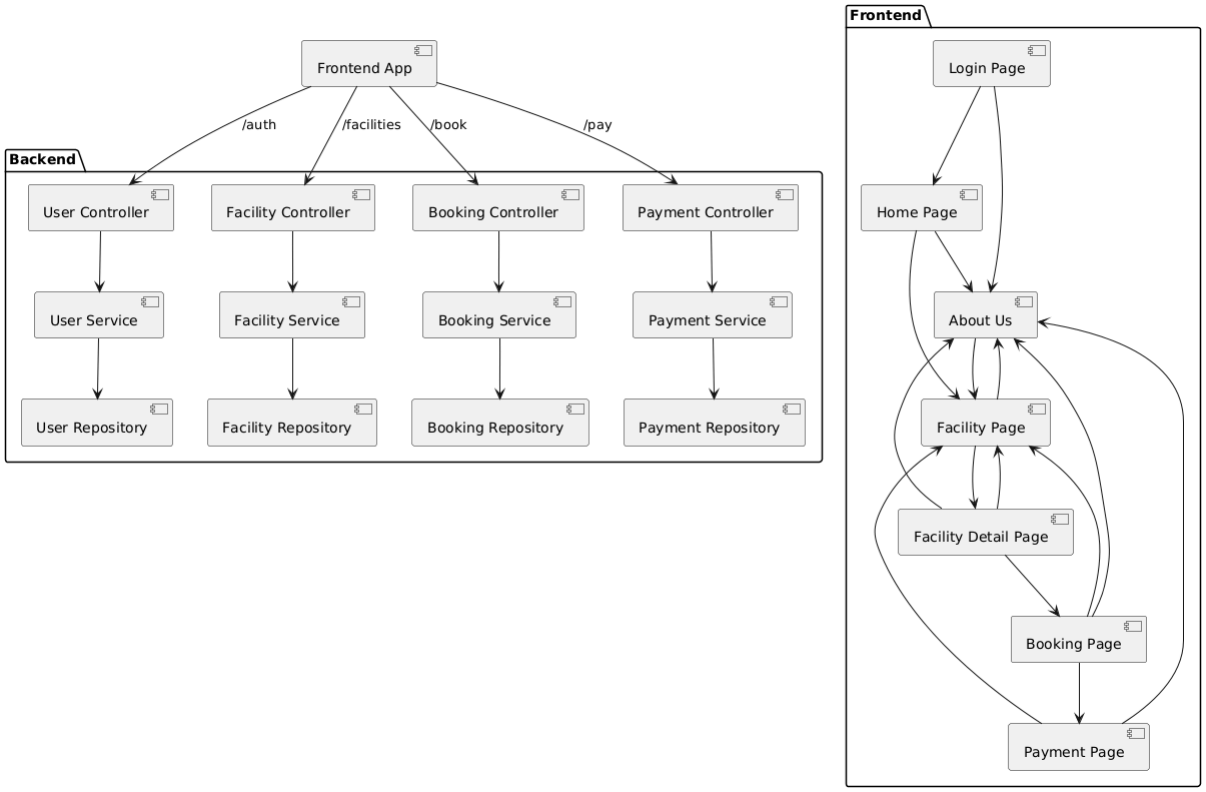
2. User Experience View + Booking and Payment Workflow View



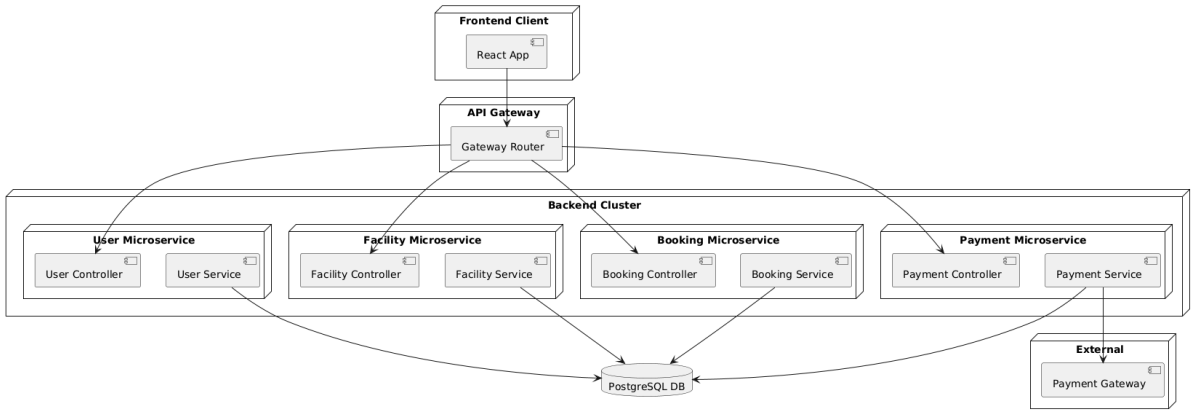
3. Facility Management View



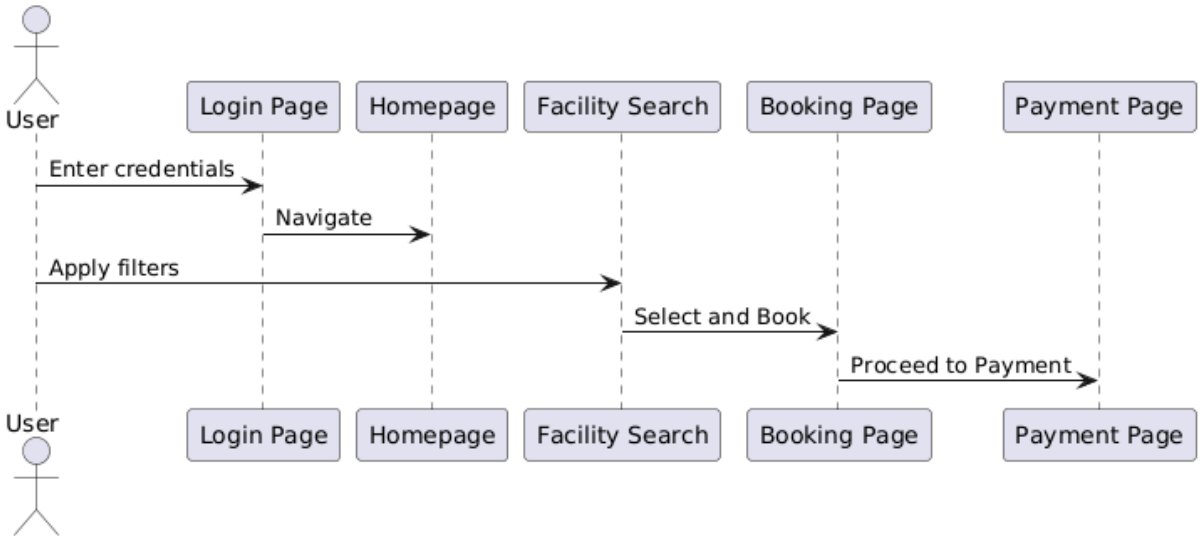
4. Backend and Frontend View



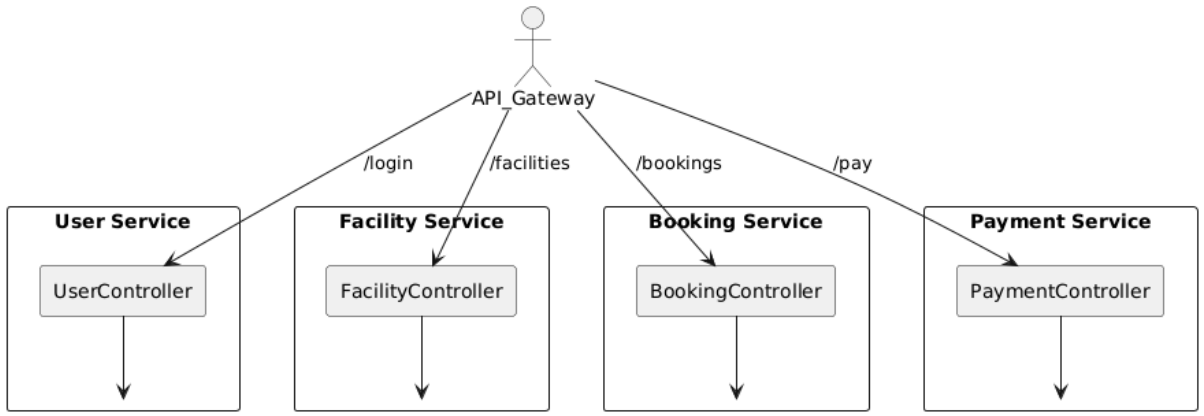
5. Deployment View for System Architech



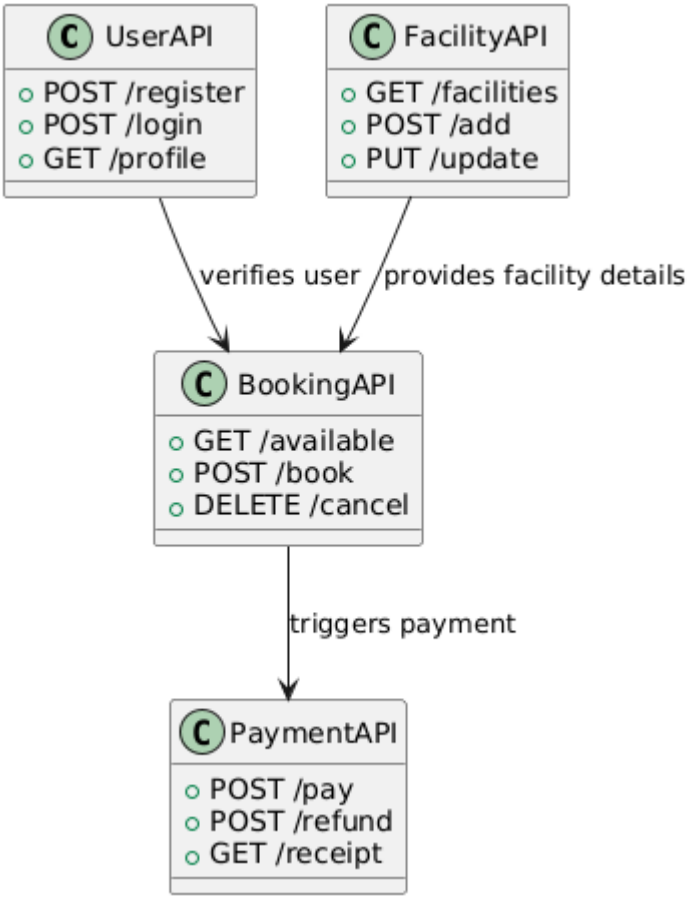
6. Interaction flow view



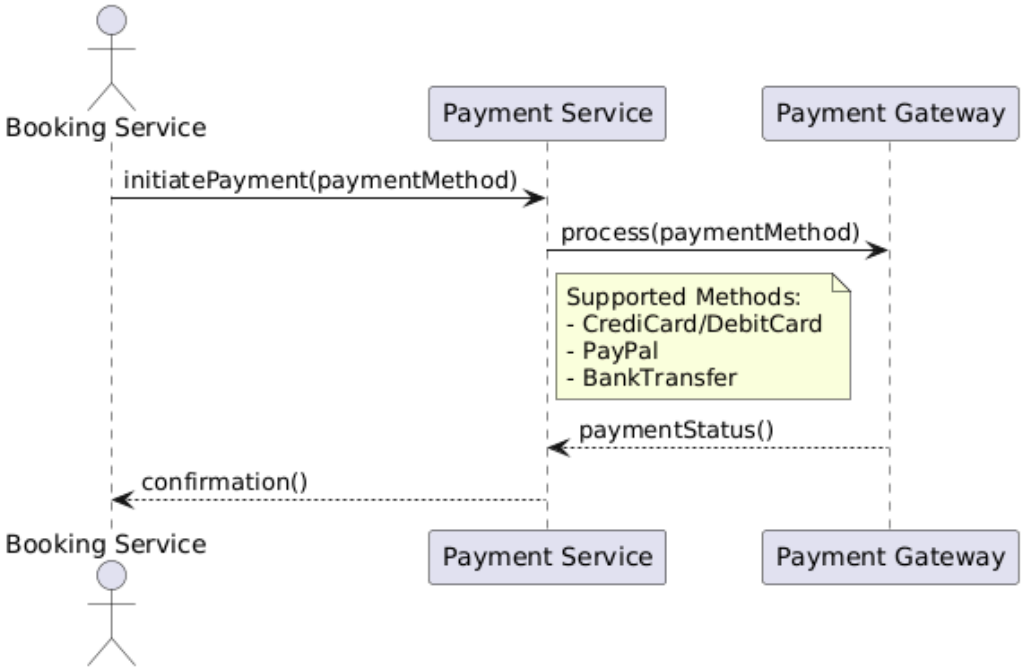
7. Service Interaction View



8. API Design View



9. Payment flow view for Payment Gateway Provider and Integration view (dev + payment gatway)



Design Decisions

Design decisions have been logged as ADR's, which have been uploaded as in separate pdf's in a separate folder. It can be found separately in this OneDrive Link

Architectural Tactics

To address the key non-functional requirements of our system—**security**, **modifiability**, **scalability**, **usability**, and **maintainability**—the following architectural tactics have been employed:

1. Token Expiry and Refresh

- **Explanation**: This tactic uses short-lived access tokens and long-lived refresh tokens to manage sessions securely while providing a smooth user experience.
- **Addresses**:
  - **Security**: Limits the impact of stolen or leaked tokens.
  - **Usability**: Enables seamless user sessions without repeated logins.

2. User Authentication

- **Explanation**: Users are authenticated via secure protocols. Every protected endpoint checks for valid authentication before allowing access.
- **Addresses**:
  - **Security**: Ensures that only authorized users can interact with protected resources.
  - **Reliability**: Prevents unauthorized operations that could affect system consistency.

3. Input Validation

- **Explanation**: Incoming data is validated against schemas or rules to prevent malformed or malicious input from entering the system.
- **Addresses**:
  - **Security**: Protects against injection attacks and other input-based vulnerabilities.

- **Reliability**: Maintains data integrity and system correctness.

4. Use Interfaces / API Abstraction

- **Explanation**: Services interact through well-defined interfaces or APIs, allowing for separation of concerns and easier evolution of individual modules.
- **Addresses**:
  - **Modifiability**: Facilitates changes or replacements in one component without affecting others.
  - **Maintainability**: Promotes clean architecture and easier testing.

5. Service Decomposition

- **Explanation**: The application is divided into modular subsystems (User, Facility, Booking, Payment), each responsible for its own set of features.
- **Addresses**:
  - **Scalability**: Each subsystem can be scaled independently based on demand.
  - **Modifiability**: Teams can develop and deploy changes in one subsystem without touching others.

6. Retry Requests

- **Explanation**: While doing a payment, If a payment fails, the system informs the user and provides a "Try Again" option to manually retry the payment without re-entering details.
- **Addresses**:
  - **Usability**: Makes it easy for users to retry failed transactions without starting over.
  - **Fault Tolerance**: Allows the system to recover from temporary issues without losing user progress.

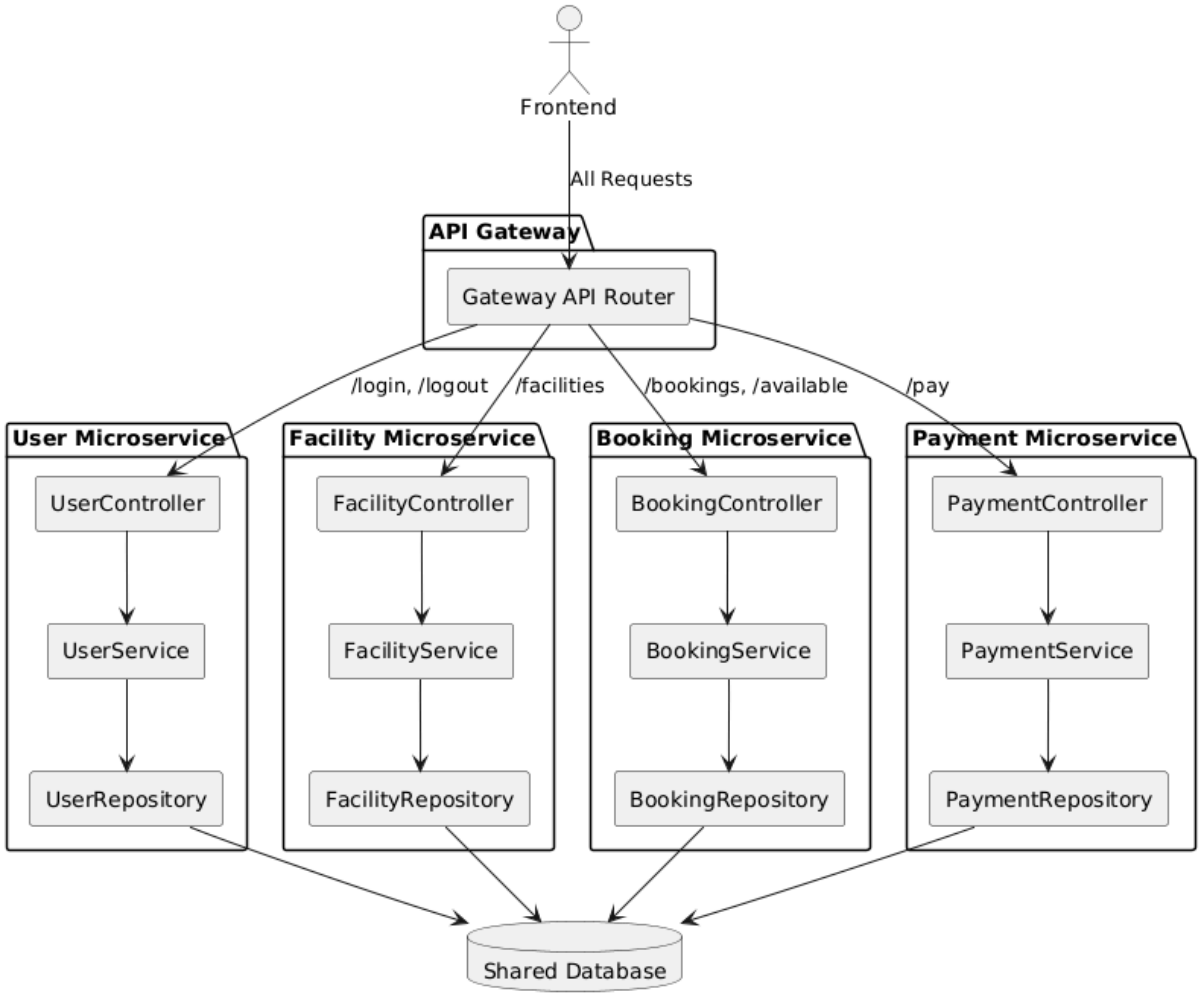
7. Load Balancing

- **Explanation**: Load balancing distributes incoming requests across multiple instances of a service such as the User Service and Auth Service to efficiently handle increased traffic and maintain responsiveness. This was

implemented using Eureka Server and Eureka Clients for service discovery and dynamic routing.

- **Addresses**:
  - **Scalability**: Supports growing user demand without performance degradation.
  - **Availability**: Maintains service uptime even if some instances go down.

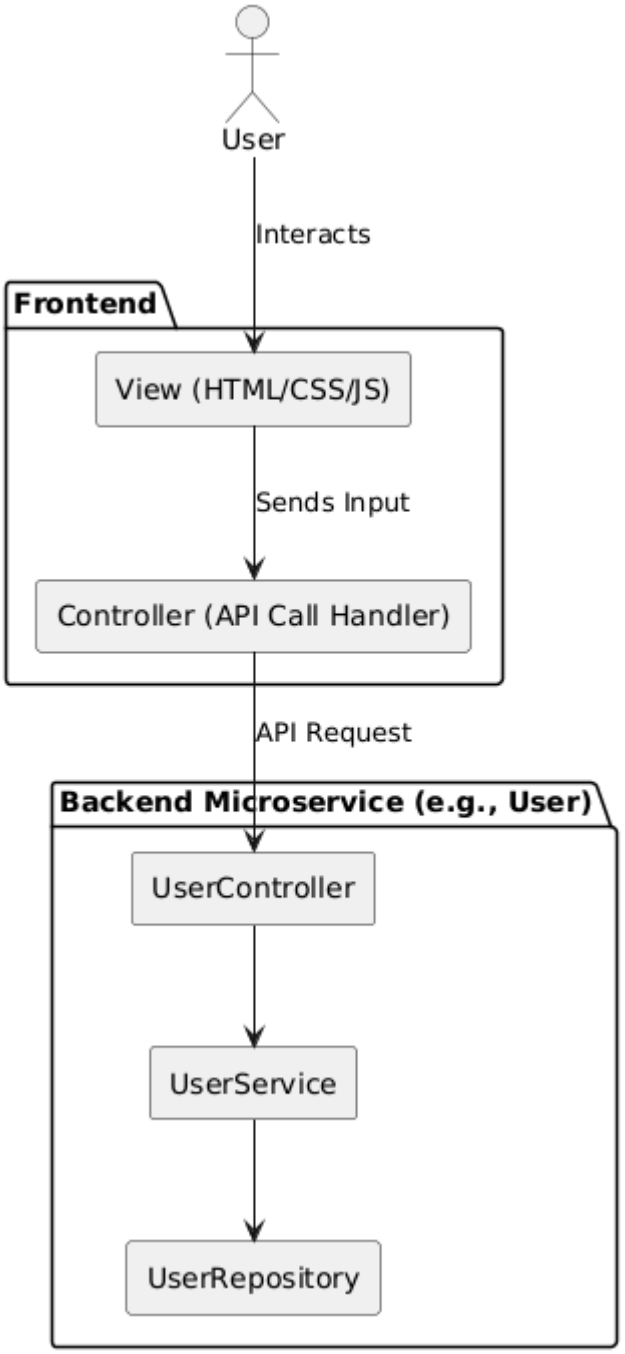
Architectural Patterns and Styles



1. Model-View-Controller (MVC) Architecture

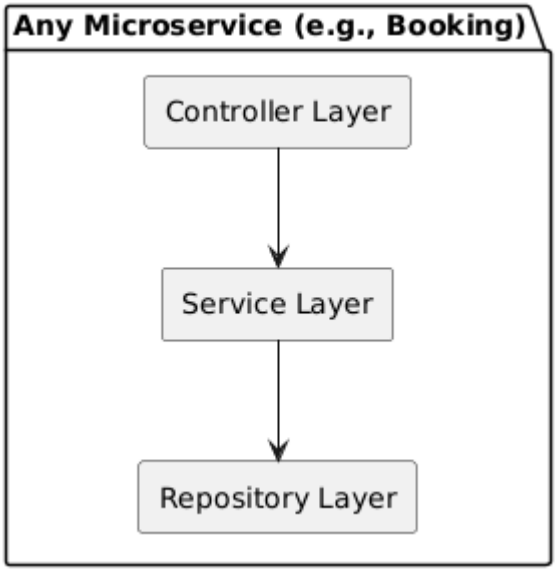
In **PitchPlease**, the Model-View-Controller (MVC) architecture is employed to structure the frontend and API interaction logic, enabling a clean separation of concerns between data, business logic, and presentation. The **Model** refers to the backend Java (Spring Boot) services, which encapsulate business logic

and data persistence for each microservice. The **View** comprises the Vanilla HTML, CSS, and JavaScript components, responsible for rendering the user interface for actions like searching facilities or viewing bookings. The **Controller** layer acts as an intermediary, handling HTTP requests, routing them to backend services, and passing responses back to the user interface. This pattern enhances modularity in the codebase and allows independent updates to frontend and backend logic, making it easier to evolve the system as user requirements change.



2. Layered Architecture

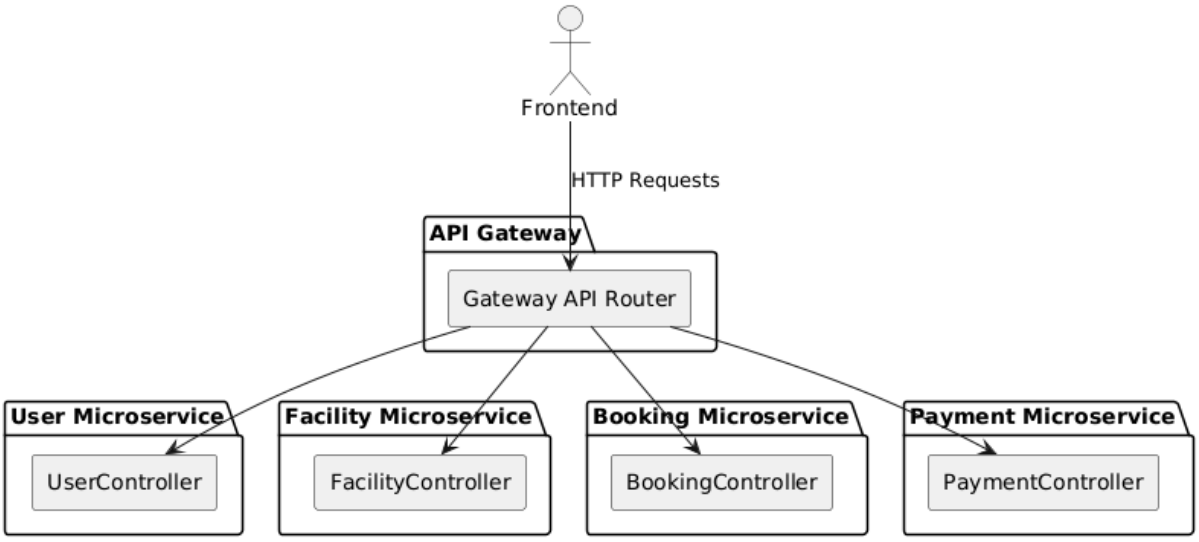
The system follows a layered architecture within each microservice to promote modularity and maintainability. In each subsystem—User, Facility, Booking, and Payment—the application is organized into three layers: a **Presentation Layer** (handling API endpoints via Spring Controllers), a **Business Logic Layer** (containing core service logic), and a **Data Access Layer** (built using Spring Data Repositories for PostgreSQL). This design enables separation of responsibilities: UI-related concerns are isolated from core application logic, and data handling is abstracted from service logic. This layering allows for clearer code organization, easier debugging, and more straightforward unit testing. It also aligns with development-centric viewpoints, addressing stakeholder concerns around maintainability, change impact, and codebase scalability.



3. Microservices Architecture

At the core of **PitchPlease** lies a Microservices Architecture, where each major domain (User, Facility, Booking, Payment) is implemented as an independent service with its own database and deployment lifecycle. These services communicate over RESTful APIs and are designed to be loosely coupled yet highly cohesive. This architecture choice was made to fulfill several architectural significant requirements, such as scalability, reliability, and fault tolerance. For example, a surge in facility search queries does not affect payment processing due to the isolation of concerns. Each service can be independently developed, scaled, and deployed, enabling faster iteration and reduced risk of system-wide failure. This architectural style aligns with

deployment, operational, and quality attribute viewpoints, ensuring that the system can evolve efficiently while meeting non-functional requirements like performance and availability.



Design Patterns Employed in the System

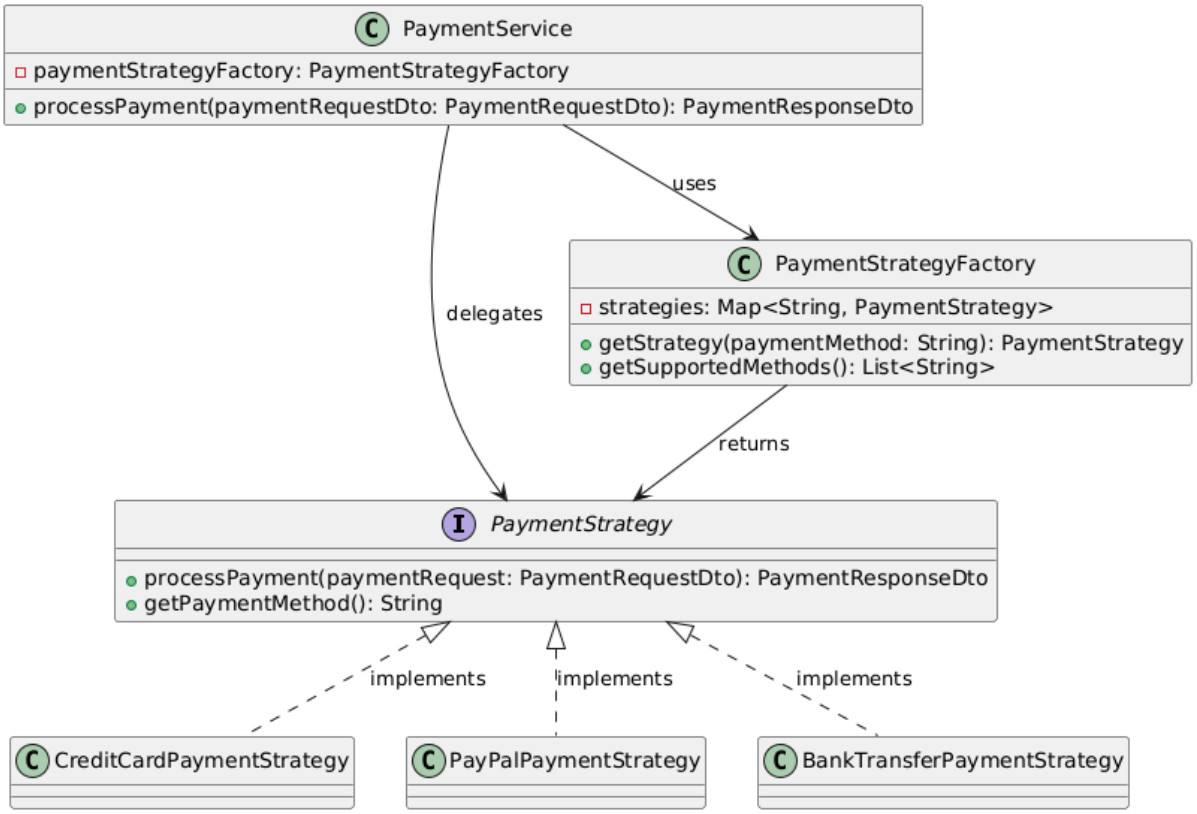
1. Factory Method Pattern – Payment Subsystem

The Factory Method Pattern is applied in the Payment subsystem to create different `PaymentStrategy` objects such as `CreditCardPaymentStrategy`, `PayPalPaymentStrategy`, or `BankTransferPaymentStrategy` without exposing the instantiation logic to the client. This design delegates the creation of specific strategy objects to a factory class named `PaymentStrategyFactory`, which determines the appropriate strategy based on the provided payment method. This encapsulation of object creation makes it easy to introduce new payment modes in the future without modifying existing code, thus adhering to the Open/Closed Principle. It also enhances clarity by abstracting processor-specific details and centralizing creation logic, leading to better maintainability.maintainability.

2. Strategy Pattern – Payment Subsystem

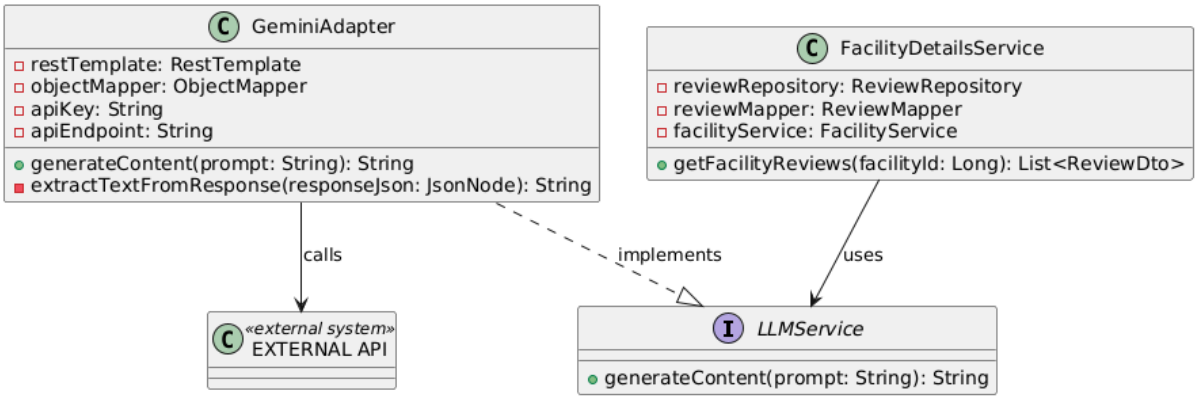
The Strategy Pattern is employed to encapsulate different payment behaviors such as credit card, PayPal, or bank transfer payments. Each payment method is implemented as a separate class (e.g., `CreditCardPaymentStrategy`, `PayPalPaymentStrategy`, `BankTransferPaymentStrategy`), all conforming to a common

`PaymentStrategy` interface. At runtime, the selected strategy is obtained from the `PaymentStrategyFactory` based on the user's input and then injected into the payment processing flow. This pattern allows seamless switching between strategies without altering the caller's logic, ensures a clean separation of concerns, simplifies testing, and provides runtime flexibility to extend behavior based on region, payment method, or policy.



3. Adapter Pattern – Review Summarization Integration

To integrate large language models (LLMs) for summarizing user reviews, the Adapter Pattern is used to unify the interface between the system's summarization logic and various external LLM providers (e.g., OpenAI, Gemini, or others). This ensures that changes in third-party APIs or libraries do not ripple through the application code. The adapter acts as a translator between the LLM's native interface and the expected format used by the review module, enabling seamless integration, improved code modularity, and support for future LLM upgrades with minimal changes.

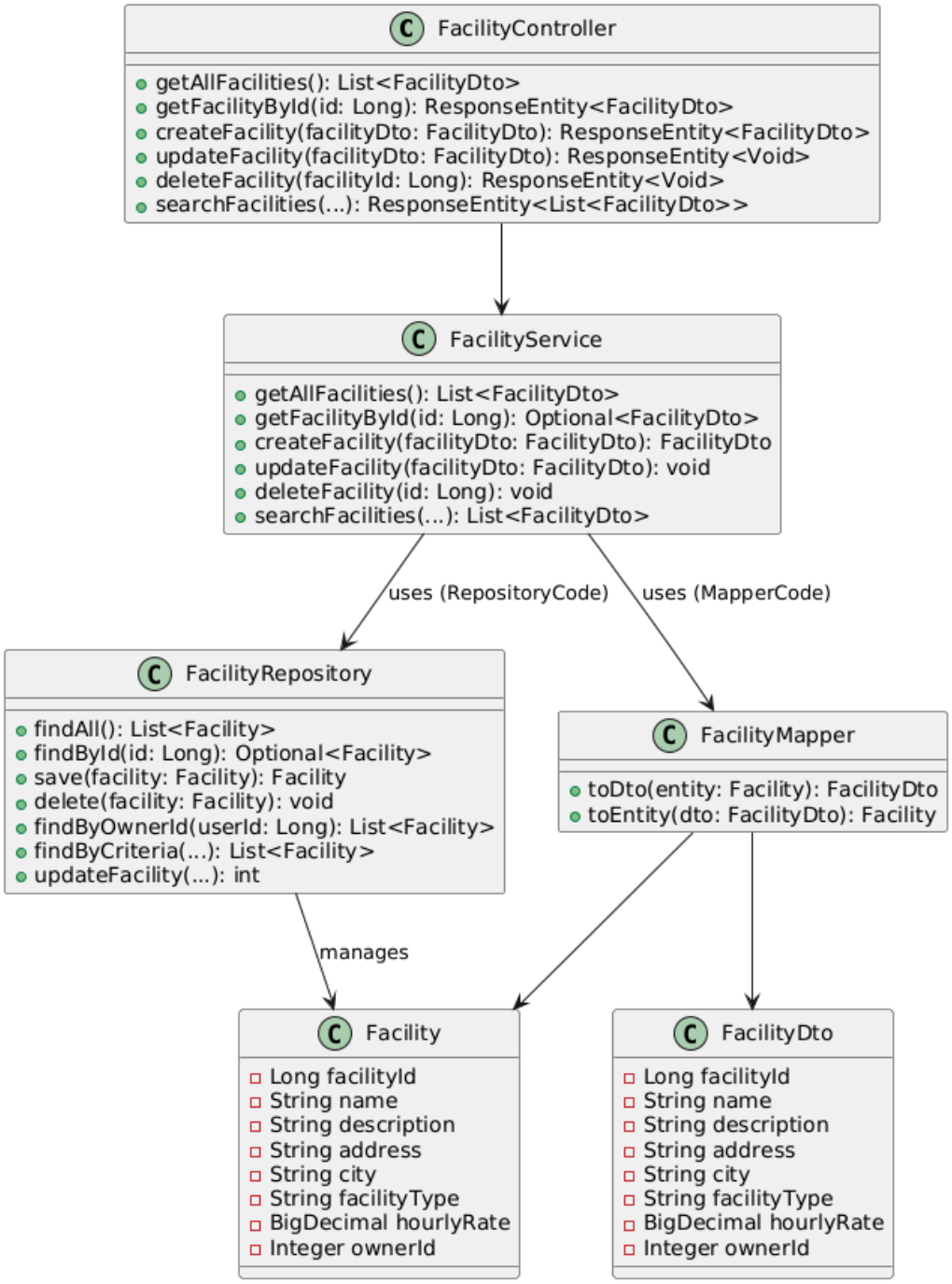


4. Repository Pattern – Backend Data Access Layer (RepositoryCode to Database)

The Repository Pattern is utilized in the backend to create a dedicated **RepositoryCode layer** that abstracts all interactions with the underlying database. Instead of embedding SQL or ORM logic directly within service classes, each repository encapsulates CRUD operations for a specific domain entity (e.g., `UserRepository`, `FacilityRepository`, etc.). This pattern acts as a **middle layer between the backend services and the database**, promoting separation of concerns. By delegating persistence logic to RepositoryCode, developers ensure that service logic remains clean and focused on business rules. Moreover, this abstraction makes it easy to swap the underlying database (e.g., switching from MySQL to PostgreSQL) or mock repositories during testing without impacting the rest of the backend.

5. Mapper Pattern – DTO ↔ Domain Model Conversion (MapperCode bridging Frontend and Backend)

The Mapper Pattern is applied to **bridge the gap between the frontend-facing DTOCode and the backend domain models**. When the frontend sends or receives API requests (typically in JSON), these payloads are first converted into Data Transfer Objects (DTOCode). The **MapperCode** is responsible for translating these DTOs into internal domain objects used by backend logic, and vice versa. This decoupling allows the backend to evolve independently of the frontend schema, ensuring that internal changes don't leak to the API layer. Similarly, MapperCode enables injecting derived fields, applying validations, or flattening nested structures when converting domain models to DTOs. This pattern ensures a consistent, modular, and clean flow of data across the frontend-backend boundary, improving maintainability and flexibility of both layers.



Architecture Analysis

Adopted Architecture: Blend of Microservices, MVC, and Layered Architecture

PitchPlease is built using a **hybrid architecture** that combines:

- **Microservices Architecture** for modularization across core domains (User, Booking, Facility, Payment)
- **Model-View-Controller (MVC)** to structure frontend-backend interaction
- **Layered Architecture** (Controller → Service → Repository) to enforce clear responsibility boundaries within each service

This combination offers the best of modular design, clear separation of concerns, and service independence—ensuring long-term maintainability and scalability.

Comparison with Monolithic Architecture

Aspect	**Hybrid Architecture (Implemented)**	**Monolithic Architecture (Alternative)**
Maintainability	**High** – Modular services, isolated responsibilities, easier refactoring	Medium – Tight coupling, difficult to isolate features or refactor
Response Time	**Slightly higher** – Due to inter-service REST API communication	**Better** – All function calls are local, no network overhead
Deployment	Independent deployment per microservice	All features redeployed together
Fault Isolation	High – Failures in one service don't crash others	Low – One module crash can bring down entire system

Quantified Non-Functional Requirements

We have quantified maintainability and response time to quantify NFR's. Maintainability has been quantified by calculating maintainability rating which is obtained by using SonarQube. Response time has been calculated for both architectures using logs.

We quantified key non-functional requirements (NFRs) by measuring both maintainability and response time.

- *Maintainability* was evaluated using SonarQube metrics by using maintainability rating.
- *Response time* was analyzed using system logs across both architectural styles to provide empirical latency comparisons.

Maintainability :

Metric	**Hybrid Architecture**	**Monolithic**
Maintainability Rating	A	A

Response Time :

Operation	**Hybrid Architecture**	**Monolithic**
Recieving all facilities	avg 246.4ms	avg 61ms

Final Insight on Architectural Analysis

The chosen hybrid architecture aligns well with PitchPlease's goals of long-term evolvability, clear separation of domains, and fault isolation. While it sacrifices a small degree of response time due to service orchestration, it offers superior maintainability, resilience, and developer productivity making it the right architectural choice for a growing, feature-rich platform.

Reflections and Lessons Learned

- Learned to implement and compare multiple architectural styles including microservices, layered, and MVC architectures.
- Applied architectural tactics to address quality attributes like fault tolerance, scalability, and modifiability.
- Used design patterns such as Strategy, Factory, Adapter, Repository, and Mapper to enforce modular, reusable, and testable code structures.
- Identified and separated Functional Requirements (FR), Non-Functional Requirements (NFR), and Architecturally Significant Requirements (ASR) to prioritize design decisions.
- Understood the role of stakeholders and captured their concerns through relevant architectural viewpoints.
- Practiced writing clear and structured Architecture Decision Records (ADRs) to document key design decisions, rejected alternatives, and their justifications.
- Realized the importance of documenting rationale to support maintainability, knowledge transfer, and future architectural evolution.

- Gained confidence in making architecture trade-offs based on technical feasibility, resource constraints, and long-term impact.