PHENIKAA UNIVERSITY

PHENIKAA UNIVERSITY: SCHOOL OF COMPUTING

COURSE: SOFTWARE ARCHITECTURE
Lab 2: Layered Architecture Design (Logical View)

Instructor:	Vu Quang Dung
Group 8:	Nguyen Gia Bao (23010383)
	Pham Xuan Bach(23010118)
	Bui Minh Duc (23010513)
	Vu Duc Hieu (23010223)
Class:	**CSE703110-1-2-25(N01)**

Hanoi, 2025 December

INDEX

1. Abstract/Summary

The Movie Ticket Booking System project continued its design phase in this lab by focusing on the **Layered Architecture Pattern**. We formally defined the four horizontal layers (Presentation, Business Logic, Persistence, and Data) and the strict downward dependency rule governing their interactions.
The system's logical view was modeled using a **UML Component Diagram**, clearly defining the software components (Controller, Service, Repository) for a core feature and illustrating their contractual relationships using Provided and Required interfaces. This foundation establishes a structured static view for implementation. However, analysis confirms that this Layered Monolith is fundamentally weak against the project's critical ASRs for Scalability and Fault Isolation, suggesting the need for architectural evolution in subsequent labs.

2. Lab Specific Section:

2.1. Defining Layers and Responsibilities

2.1.1. Define the Four Layers:

Layer	Purpose/Responsibility	Output/Artifact
Presentation Layer (UI)	Handles interactions with users (Web Customer, Cinema Manager). Displays movie lists and visual seat maps. Receives booking and payment requests.	Controllers / Views
Business Logic Layer (Service/Domain)	Contains core business rules such as: processing bookings, applying discount codes, ensuring data consistency to prevent double-booking of seats (ASR 2), and integrating payment gateways.	Services / Managers
Persistence Layer (Data access)	Executes raw data queries from the database (movies, showtimes, seat status, e-tickets) and maps them to system objects (entities).	Repositories / DAOs
Data layer	The physical storage system containing movie information, booking history, and user account details.	Database Schema

2.1.2. Define Data Flow:

Client Request: The User (Web Customer) sends an HTTP POST request to the endpoint: /bookings/lock-seat?showtimeId=...&seatId=...&userId=....

Layer 1 (Presentation): The BookingController receives the request, performs user session authentication, and validates the input parameters. It then invokes Layer 2.

Layer 2 (Business Logic): The BookingService executes the lockSeat(showtimeId, seatId, userId) function. Here, the system applies critical business rules, such as **checking**

if the seat is already locked/booked (enforcing ASR 2: Data Consistency) and **starting the Reservation Window countdown**. It then invokes Layer 3.

Layer 3 (Persistence): The SeatRepository receives the command and constructs the necessary mechanism to ensure atomicity (e.g., executing a **Distributed Lock** command or an atomic SQL query: UPDATE seats SET status='LOCKED' WHERE seat_id=? AND status='AVAILABLE').

Layer 4 (Data): The Database (PostgreSQL/Redis) executes the atomic operation and returns the result (Success/Failure) of the lock attempt.

Return Path:

- The SeatRepository maps the raw lock status result into a system object (e.g., LockStatus).
- The BookingService handles the result (if lock is successful, start timer/queue message; if failed, throw exception).
- The BookingController returns the **"Seat Locked Confirmation"** (JSON) or **"Seat Already Taken"** error response to the Client.

3. Component Identification (Product Catalog)

This section breaks down the critical "Book Ticket" feature into concrete software components residing in the top three layers of the Layered Architecture.

3.1. Identify Components:

Layer 1: Presentation Layer

- **Component Name:** BookingController
- **Responsibility:** Receives the booking request (e.g., POST /book), validates the input format (seat IDs, showtime ID), and delegates the complex processing and seat locking to the Business Logic Layer.

Layer 2: Business Logic Layer

- **Component Name:** BookingService
- **Responsibility:** Coordinates the entire reservation process. It must first call the SeatLockManager to lock the selected seats (critical step, handles ASR 2), verifies discount codes, calculates the final price, and initiates the payment process via the PaymentService.
- **Component Name:** SeatLockManager
- **Responsibility:** Manages the **Reservation Window**. Uses distributed locking mechanisms (not visible here, but conceptually required) to atomically change seat status from 'Available' to 'Locked' for a fixed duration, preventing double-booking (ASR 2).
- **Component Name:** PaymentService
- **Responsibility:** Encapsulates the logic for communicating with the external **Payment Gateway** (Actor), ensuring secure communication and isolating third-party API dependencies (ASR 3).

Layer 3: Persistence Layer

- **Component Name:** TicketRepository
- **Responsibility:** Translates the finalized booking object into database commands (e.g., INSERT INTO tickets..., UPDATE seats SET

status='BOOKED') to permanently persist the reservation and update the seat status atomically.

3.2. Define Interfaces:

The following interfaces define the contracts between layers, allowing for component interchangeability and enforcing the strict downward dependency rule of the Layered Architecture:

Interface provided by Business Logic (for Presentation Layer):

- This interface is provided by the BookingService for the BookingController (Layer 1) to invoke, initiating the complex transaction.
- ITicketBookingService.createBooking(bookingRequest: Dict) -> BookingResult

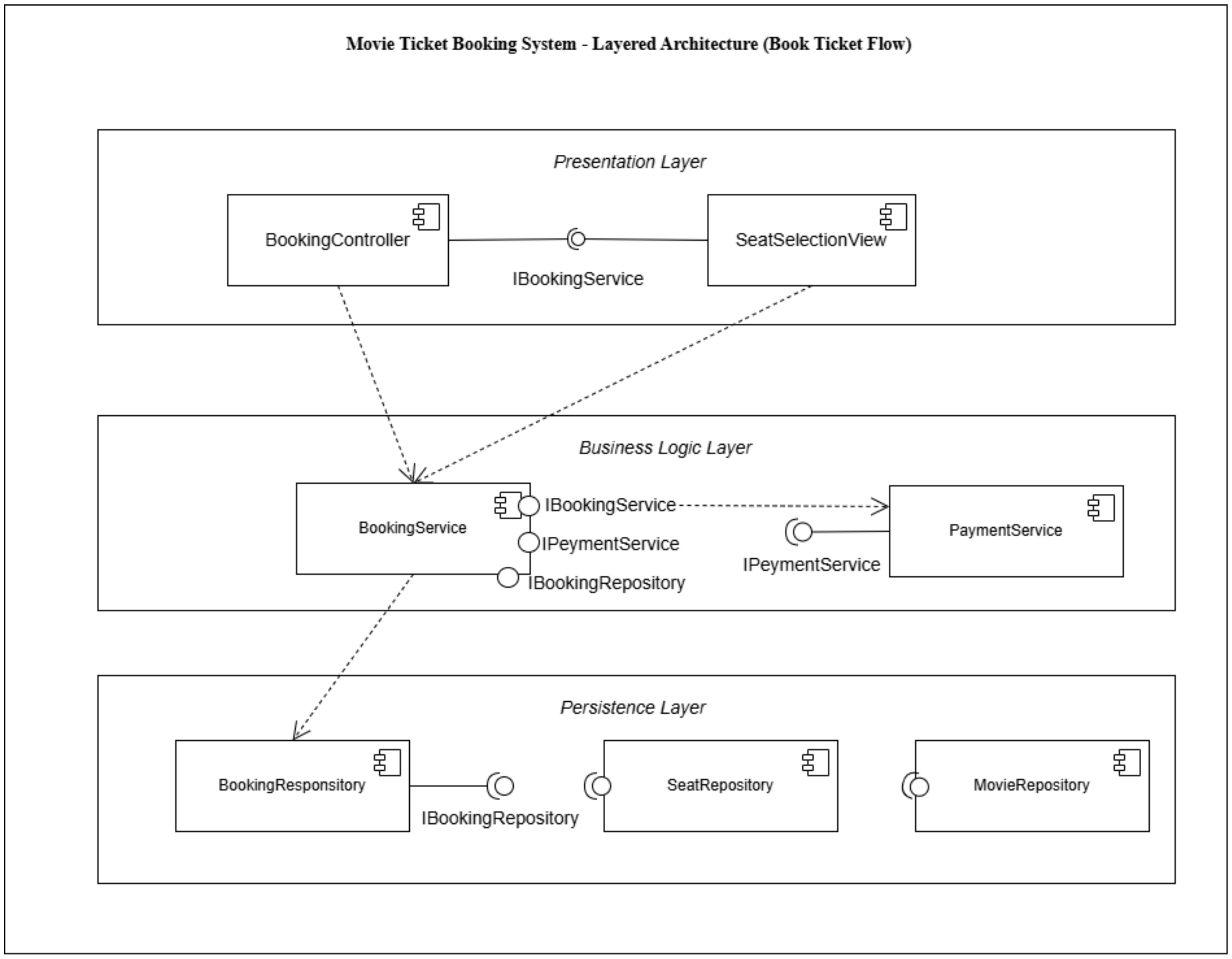
Interfaces provided by Persistence (for Business Logic Layer):

- This interface is provided by the TicketRepository for the BookingService (Layer 2) to manage final data persistence and integrity.
- ITicketRepository.save(ticket: TicketEntity) -> void
- ITicketRepository.findTicketDetails(bookingId: int) -> TicketEntity

Interface provided by specialized Business Logic (Internal Contract):

- This interface is provided by the PaymentService for the BookingService to delegate the external payment processing step (critical for ASR 3).

4. Modeling Artifact: UML Component Diagram



This UML diagram is used to model the 'Logical View' of the software architecture, helping to visualize how software modules or components are organized and connected to each other.

It describes the following design tasks:

- Defining 3 Architectural Layers:

The image divides the system into three large, stacked rectangles, representing three layers of responsibility: Presentation Layer, Business Logic Layer, and Persistence Layer.

- Components Placement:

It defines the precise location of code modules within the system:

    - ProductController: Located in the top layer (Presentation), responsible for receiving requests from users.
    - ProductService: Located in the middle layer (Business Logic), responsible for processing business logic.
    - ProductRepository: Located in the bottom layer (Persistence), responsible for data retrieval.
- Designing Connection Interfaces:

The image uses standard UML notation to describe how components communicate with each other:

    - "Lollipop" Symbol (Provided Interface): Example IProductService, representing the service that the ProductService component exposes to the outside.
    - "Socket" Symbol (Required Interface): Attached to the ProductController, indicating that the Controller needs to "plug into" the Service to function.
- Enforcing Strict Dependency:

All dashed arrows point strictly downward (from Layer 1 → Layer 2 → Layer 3). This illustrates the most critical rule of this architecture: a layer can only interact with the layer directly below it; it cannot make upward calls or skip layers.

5. Conclusion & Reflection

The Lab 2 design phase successfully established the logical structure of the Movie Ticket Booking System using the **Layered Architecture Pattern**. We formally defined four distinct layers with strict downward dependencies and modeled the core **"Book Ticket"** feature using a **UML Component Diagram**, identifying key components like BookingController and TicketRepository. While this monolithic structure offers simplicity for initial development, analysis confirms it is insufficient for the project's critical **Scalability (ASR 1)** and **Fault Isolation (ASR 2)** requirements. Consequently, while this design supports the immediate implementation in Lab 3, an evolution toward a **Distributed Architecture** will be necessary to effectively handle high-concurrency demands.