Lab 2: Layered Architecture Design (Logical View)

Bus Ticket Management System

Introduction

This report presents the logical view and layered architecture design of the Bus Ticket Management System. Building upon the functional and non-functional requirements identified in Lab 1, this stage of the design emphasizes the system's static structure and internal organization rather than runtime behavior. The primary objective of this architecture is to clearly separate responsibilities across different parts of the system, thereby improving maintainability, scalability, and long-term extensibility. To achieve these goals, the system adopts a strict layered architecture pattern, where each layer has a well-defined role and communicates only with its adjacent lower layer. This approach minimizes coupling between components and allows changes in one layer to have minimal impact on others.

Activity Practice 1: Defining Layers and Responsibilities

1.1 The Four Main Architectural Layers

The backend of the Bus Ticket Management System is structured into four logical layers, each responsible for a specific set of concerns. At the top of the architecture is the Presentation Layer, which serves as the entry point for all client interactions. This layer handles incoming HTTP requests from client applications such as a React-based frontend. It is responsible for routing requests to the appropriate controllers, validating request data using Zod schemas, and returning standardized JSON responses with suitable HTTP status codes. By limiting its responsibility to request handling and response formatting, the Presentation Layer remains lightweight and free of business-specific logic.

Beneath the Presentation Layer lies the Business Logic Layer, often referred to as the Service Layer. This layer encapsulates the core business rules of the system, including operations such as verifying seat availability, validating trip status, and calculating ticket pricing. It acts as the central coordinator of application behavior by orchestrating calls to the persistence layer and managing transaction boundaries when necessary. By isolating business rules within services, the system ensures that domain logic remains consistent and reusable across different entry points, such as APIs or potential future interfaces.

The third layer in the architecture is the Persistence Layer, which abstracts direct access to the database. This layer is implemented using repositories that leverage Knex.js to execute SQL queries. Its primary responsibility is to translate high-level business requests into database operations and map raw database records into domain models. By isolating database access logic within repositories, the system avoids scattering SQL queries throughout the codebase, thereby improving readability and easing future database migrations or optimizations.

At the lowest level of the architecture is the Data Layer, which represents the physical storage mechanism. In this system, the data layer consists of a MySQL database that stores all persistent information related to bus schedules, tickets, vehicles, and related entities. This layer is not concerned with application logic; instead, it focuses solely on reliable and efficient data storage through well-defined schemas and relational constraints.

Layer	Purpose/Responsibility	Output/Artifact
1. Presentation Layer (UI/API)	Handles HTTP requests, manages routing, validates input using Zod schemas, and returns standardized JSON responses. It serves as the entry point for the client applications.	Controllers (e.g., VehicleScheduleController), Routers (e.g., VehicleScheduleRouter)
2. Business Logic Layer (Service)	Implements core business rules (e.g., seat availability check, pricing calculations), orchestrates data access, and handles transaction boundaries.	Services (e.g., VehicleScheduleService)
3. Persistence Layer (Data Access)	Abstracts database operations using Knex.js. Responsible for mapping business objects to database entities and executing SQL queries.	Repositories (e.g., VehicleScheduleRepository)
4. Data Layer (Physical storage)	The physical database storage system where the bus booking data resides.	Database Schema (MySQL Tables: schedules, tickets, cars, etc.)

1.2 Data Flow Logic

To illustrate how these layers collaborate, consider the scenario in which a customer views the details of a specific bus trip. The interaction begins when the React frontend sends an HTTP GET request to the endpoint /vehicle-schedules/{id}. This request is received by the Presentation Layer, where the corresponding controller extracts the schedule identifier from the request parameters and forwards it to the Business Logic Layer.

Upon receiving the request, the service component in the Business Logic Layer performs any necessary validations, such as confirming that the schedule exists and that its status is active. If additional information is required, such as calculating the number of available seats, the service coordinates with other domain services or repositories. Once the business logic has been executed, the service invokes the Persistence Layer to retrieve the required data.

The Persistence Layer then executes a parameterized SQL query through Knex.js, such as selecting a record from the schedules table based on the provided identifier. The MySQL database processes this query and returns the corresponding row data. This data is mapped into a domain model and passed back through the layers in reverse order. The service layer wraps the result in a standardized ServiceResponse object, and the controller ultimately returns a JSON response to the client with an appropriate HTTP status code. This clear and linear flow ensures that each layer remains focused on its specific responsibility.

The following trace represents the request flow for a customer viewing a **Bus Trip Detail** page:

1. **Client Request**: The React frontend sends a GET /vehicle-schedules/{id} request to the backend.
2. **Layer 1 (Presentation)**: VehicleScheduleController receives the request, extracts the id, and calls the Service layer.
3. **Layer 2 (Business Logic)**: VehicleScheduleService receives the id, performs any necessary business validation, and calls the Repository layer.
4. **Layer 3 (Persistence)**: VehicleScheduleRepository executes a Knex query like select * from schedules where id = ? to the database.
5. **Layer 4 (Data Layer)**: The MySQL database executes the query and returns the raw row data.
6. **Return Path (L3 -> L2 -> L1)**:
    - **Repository** returns a Model instance to the Service.
    - **Service** processes the data (completing the ServiceResponse object) and returns it to the Controller.
    - **Controller** returns the JSON response with the appropriate HTTP status code to the client.

Activity Practice 2: Component Identification (Trip Catalog/Schedules)

Focusing on the “View Trip Details” functionality, which corresponds to the trip catalog requirement, several key components are identified across the upper layers of the architecture. In the Presentation Layer, the VehicleScheduleController is responsible for handling incoming requests related to vehicle schedules. It processes HTTP GET requests, extracts route parameters, and delegates the core processing to the service layer. Its role is limited to request handling and response formatting, ensuring that no business logic leaks into the controller.

Within the Business Logic Layer, the VehicleScheduleService implements the rules associated with retrieving trip details. This includes verifying that a requested trip is valid and active, and optionally aggregating related information such as seat availability or pricing details. By centralizing this logic within the service, the system ensures consistency and simplifies future enhancements to trip-related functionality.

The Persistence Layer contains the VehicleScheduleRepository, which directly interacts with the database. This repository executes SQL queries against the schedules table and returns domain objects representing vehicle schedules. If a schedule cannot be found, the repository returns a null value, allowing the service layer to handle this scenario according to business rules.

2.1 Component Identification

For the feature **"View Trip Details"**, the following components are identified within the top three layers:

- **Layer 1 (Presentation):**
    - **Component Name:** VehicleScheduleController

    - **Responsibility:** Receives the HTTP request GET /vehicle-schedules/{id}. It parses the ID parameter and calls the VehicleScheduleService. It formats the ServiceResponse into a standard JSON output.

- **Layer 2 (Business Logic):**
    - **Component Name:** VehicleScheduleService
    - **Responsibility:** Implements the business logic for retrieving a trip. This includes checking if the trip's status is active and potentially aggregating additional data like available seat count from the SeatRepository before returning the trip details.
- **Layer 3 (Persistence):**
    - **Component Name:** VehicleScheduleRepository
    - **Responsibility:** Uses Knex.js to query the schedules table in MySQL. It handles the raw database connection and returns the trip entity or null if not found.

2.2 Interface Definitions

Clear interfaces define how layers communicate with one another. The VehicleScheduleService exposes a method that allows the Presentation Layer to retrieve trip details by identifier, returning a ServiceResponse object that encapsulates both the data and the operation status. This abstraction allows controllers to remain agnostic of how the data is retrieved or processed internally.

Similarly, the VehicleScheduleRepository defines an interface for accessing schedule data from the database. It accepts a primary key identifier and returns either a corresponding domain model or null. This simple and well-defined contract ensures that the service layer does not depend on database-specific details, reinforcing the principle of separation of concerns.

The primary interfaces between layers for the "View Trip Details" operation are defined below:

VehicleScheduleService Interface (for Layer 1):

public async findById(id: number): Promise<ServiceResponse<VehicleSchedule | null>>

- **Input:** id (The unique identifier for the schedule).
- **Output:** A ServiceResponse object containing the VehicleSchedule data and the operation status.

VehicleScheduleRepository Interface (for Layer 2):

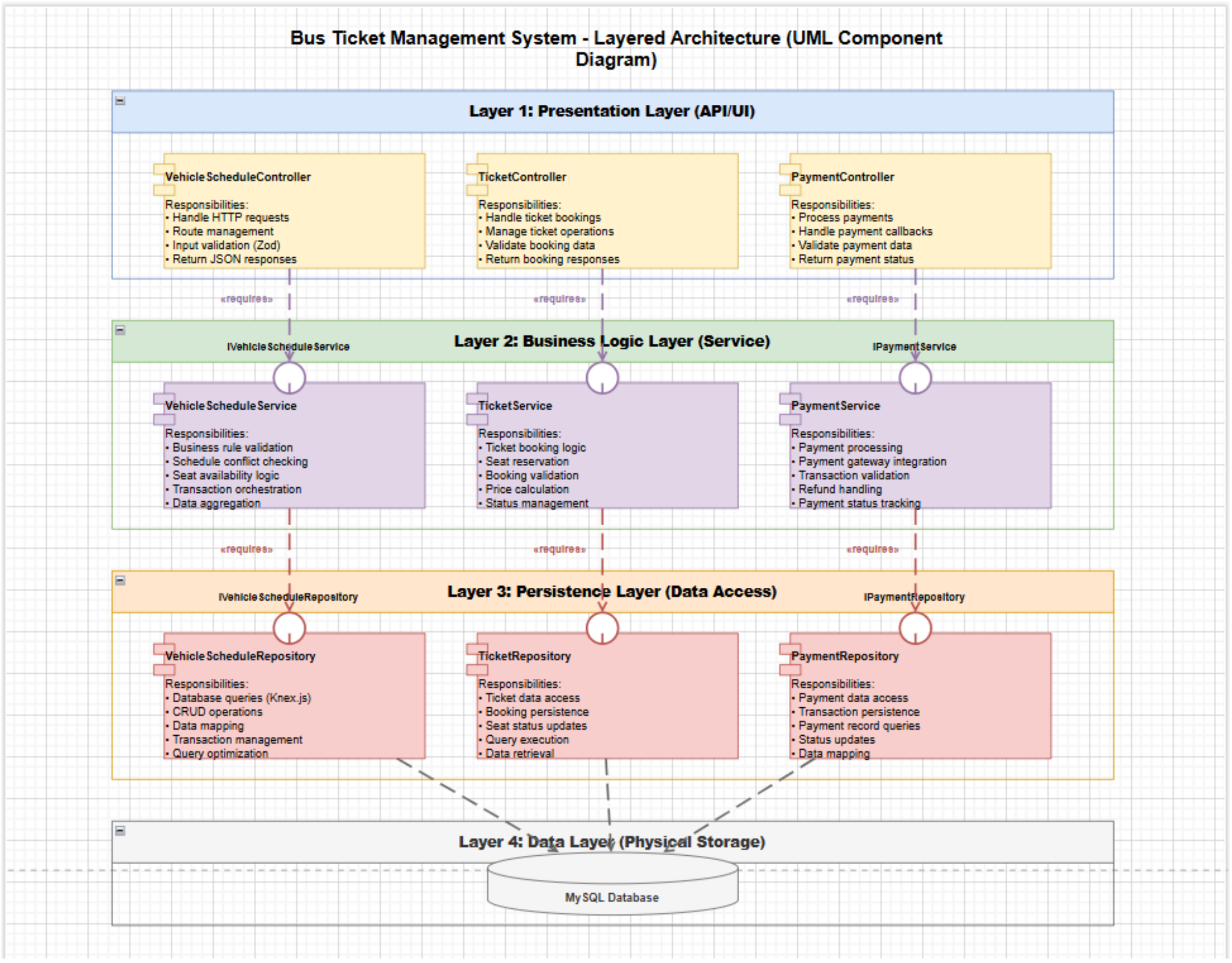
public async findById(id: number): Promise<VehicleSchedule | null>

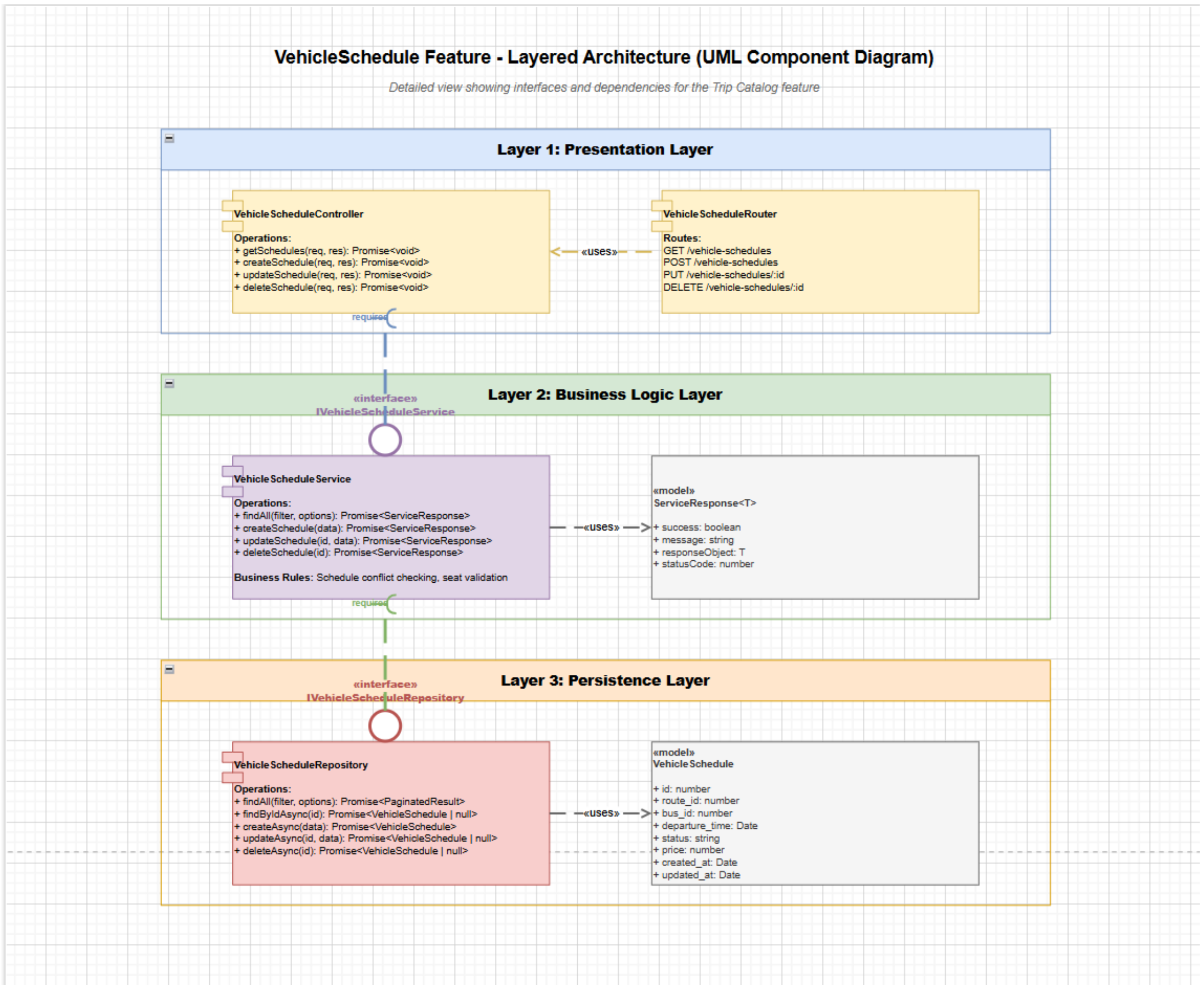
- **Input:** id (The database primary key).
- **Output:** The raw VehicleSchedule model object or null.

Conclusion

The Bus Ticket Management System demonstrates a classic four-tier layered architecture that effectively separates presentation, business logic, persistence, and data storage concerns. This architectural approach enhances maintainability by isolating changes within specific layers and promotes scalability by allowing individual layers to evolve independently. The detailed examination of the trip schedule feature illustrates how requests traverse the architecture in a controlled and predictable manner, supported by clear interfaces and well-defined responsibilities. As a result, the system provides a solid foundation for future development, extension, and optimization while adhering to established software design principles.

Activity Practice 3: Component Diagram Modeling




VehicleSchedule Feature - Layered Architecture (UML Component Diagram)
Detailed view showing interfaces and dependencies for the Trip Catalog feature
Layer 1: Presentation Layer
VehicleScheduleController
Operations:
+ getSchedules(req, res): Promise<void>
+ createSchedule(req, res): Promise<void>
+ updateSchedule(req, res): Promise<void>
+ deleteSchedule(req, res): Promise<void>
«uses»
VehicleScheduleRouter
Routes:
GET /vehicle-schedules
POST /vehicle-schedules
PUT /vehicle-schedules/:id
DELETE /vehicle-schedules/:id
required
Layer 2: Business Logic Layer
«interface»
IVehicleScheduleService
VehicleScheduleService
Operations:
+ findAll(filter, options): Promise<ServiceResponse>
+ createSchedule(data): Promise<ServiceResponse>
+ updateSchedule(id, data): Promise<ServiceResponse>
+ deleteSchedule(id): Promise<ServiceResponse>
Business Rules: Schedule conflict checking, seat validation
«uses»
«model»
ServiceResponse<T>
+ success: boolean
+ message: string
+ responseObject: T
+ statusCode: number
required
Layer 3: Persistence Layer
«interface»
IVehicleScheduleRepository
VehicleScheduleRepository
Operations:
+ findAll(filter, options): Promise<PaginatedResult>
+ findByIdAsync(id): Promise<VehicleSchedule | null>
+ createAsync(data): Promise<VehicleSchedule>
+ updateAsync(id, data): Promise<VehicleSchedule | null>
+ deleteAsync(id): Promise<VehicleSchedule | null>
«uses»
«model»
VehicleSchedule
+ id: number
+ route_id: number
+ bus_id: number
+ departure_time: Date
+ status: string
+ price: number
+ created_at: Date
+ updated_at: Date