Bus Ticket Reservation System – Design Document

Overview

This system simulates a simplified **Bus Ticket Reservation System** via REST APIs using Java. The application allows clients to:

- **Check seat availability** and pricing between two locations.
- **Book tickets** for a specified number of passengers if enough seats are available.
- Simulate multiple users booking simultaneously using a multithreaded client.

The system is implemented using the com.sun.net.httpserver.HttpServer package (no third-party frameworks), and uses **Google Gson** for JSON processing.

Assumptions & Constraints

- The bus operates **between A to D**, with intermediate stops at B and C.
- **One round-trip per day**: A → D and D → A.
- Ticket prices are fixed and predefined.
- The bus contains **40 seats** named from 1A to 10D (4 seats per row, 10 rows).
- Passengers can board from any station (A, B, C) to any later stop and vice versa.

Architecture Overview

```
client.MultiUserSim   ---> Simulates concurrent users
       |
       v

client.BookingClient ---> Calls HTTP APIs
       |
       v
HTTP Server
     (Port 8080)

 |
 |        +--> /availability (AvailabilityHandler)
 |        +--> /reserve      (ReservationHandler)
 |
 v

ReservationService
Validates requests
Calculates availability
Books seats
```

Package Structure

server.model

- **RequestData** – Represents incoming requests for availability/reservation.
- **ResponseData** – Output of availability check.
- **TicketResponse** – Final reservation response with ticket details.

server.service

- **ReservationService** – Core logic for handling pricing, availability, seat tracking, and ticket generation.

server

- **AvailabilityHandler** – HTTP handler for availability check.
- **ReservationHandler** – HTTP handler for ticket reservation.
- **BusHttpServer** – Launches the HTTP server and configures endpoints.

client

- **BookingClient** – Makes sequential API calls to check and book tickets.
- **MultiUserSimulation** – Simulates concurrent users by running BookingClient instances in parallel threads.

API Contracts

1. /availability (POST)

Request JSON:

```
{
  "origin": "A",
  "destination": "D",
  "passengerCount": 3
}
```

Response JSON:

```
{
  "available": true,
  "totalPrice": 450,
  "availableSeats": ["1A", "1B", "1C"]
}
```

2. /reserve (POST)

Request JSON:

```
{
  "origin": "A",
  "destination": "D",
  "passengerCount": 3,
  "paymentAmount": 450
}
```

Response JSON:

```
{
  "ticketNumber": "TKT1000",
  "bookedSeats": ["1A", "1B", "1C"],
  "origin": "A",
  "destination": "D",
  "departureTime": "09:00 AM",
  "arrivalTime": "12:00 PM",
  "totalPrice": 450
}
```

Unit Test

Implemented using **JUnit 5**. Covers:

- Successful availability check
- Successful ticket reservation
- Insufficient payment failure
- Booking when no seats are available
- Invalid origin/destination validation
- Overpayment handling

Example Test Snippet

```
@Test
public void testReserveTicketSuccess() {
    ResponseData availability =    ReservationService.checkAvailability(request);
    request.setPaymentAmount(availability.getTotalPrice());
    TicketResponse ticket = ReservationService.reserveTicket(request);
    assertNotEquals("FAILED", ticket.getTicketNumber());
    assertEquals(2, ticket.getBookedSeats().size());
}
```

Simulating Concurrency

- The MultiUserSimulation class spawns multiple threads to mimic real-world simultaneous booking attempts.

- This validates that your in-memory booking system can handle race conditions (to a basic extent, but would need locks or synchronization in real-world scenarios).

How to Run This Project in IntelliJ IDEA

Project Requirements

Java
Maven (if configured for dependencies like Gson and JUnit 5)

Running the Server

- Open project in IntelliJ IDEA.
- Run BusHttpServer.java.
- Server starts on http://localhost:8080/

Running the Client

- Open project in IntelliJ IDEA.
- Run BookingClient.java.

Possible Improvements

1. **Thread Safety:**

    - Use synchronized blocks or ConcurrentHashMap to prevent race conditions on seat booking in multi-user scenarios.

2. **Persistence Layer:**

    - Replace in-memory storage with a database (e.g., PostgreSQL or SQLite) for durability.

3. **Use Frameworks:**
    - Use Spring Boot for a more scalable and configurable server.
4. **Validation:**
    - Validate input fields more rigorously (e.g., null checks, valid station names, positive integers).
5. **Return Journey Logic:**
    - Add return trip booking with time slot differentiation.
6. **Seat Layout Handling:**
    - Implement a visual seat map or sectioning logic (e.g., window/aisle preferences).
7. **REST Standards:**
    - Use HTTP status codes properly (e.g., 400 for bad input, 409 for seat conflict).
8. **Logging & Monitoring:**
    - Add structured logging and metrics (via SLF4J + Prometheus).
9. **Swagger Documentation:**
    - Provide OpenAPI/Swagger UI to describe API contracts.

Conclusion

This assignment showcases a complete and functional ticket booking REST API using core Java concepts. It implements request validation, pricing, availability logic, and booking logic with support for concurrent simulation. With a few enhancements in concurrency control, persistence, and validations, this system can be evolved into a real-world booking platform.