1. What is the purpose of the SeatBookingController class?

Answer: The SeatBookingController is a REST controller in a Spring Boot application. Its primary responsibility is to handle incoming HTTP requests related to seat booking. It provides endpoints for fetching available seats (/seats/available) and booking seats (/seats/book). The controller interacts with the SeatBookingService to process these requests.

2. How do you handle exceptions in the bookSeats method in SeatBookingController?

Answer: The bookSeats method catches specific exceptions (SeatAlreadyBookedException, SeatLimitExceededException, CustomerNotFoundException) and returns a 400 Bad Request response with the error message. If any other unhandled exceptions occur, it returns a 500 Internal Server Error with a generic error message.

3. Why is the @Transactional annotation used in the bookSeats method in SeatBookingService?

Answer: The @Transactional annotation ensures that the entire booking process is treated as a single atomic transaction. If any exception occurs during the booking process (e.g., a seat is already booked), the changes made to the database (such as marking seats as booked) will be rolled back, maintaining data consistency.

4. What is the purpose of the getAvailableSeats method in SeatBookingService?

Answer: The getAvailableSeats method retrieves a list of seats that are not booked (isBooked = false) from the database using the SeatRepository. This method is used to show users the seats available for booking.

5. What is the role of the MAX_SEATS_PER_USER constant in the service class?

Answer: The MAX_SEATS_PER_USER constant defines the maximum number of seats a user is allowed to book. It is used in the bookSeats method to ensure that a customer doesn't exceed the seat booking limit, and an exception (SeatLimitExceededException) is thrown if the limit is exceeded.

6. Explain the use of the @RequiredArgsConstructor annotation in the SeatBookingController and SeatBookingService classes.

Answer: The @RequiredArgsConstructor annotation is part of Project Lombok and automatically generates a constructor for any final fields in the class. This removes the need to manually write constructors and helps with Dependency Injection in Spring, allowing Spring to inject the required services automatically.

7. How does the bookSeats method ensure that a seat is not double-booked?

Answer: The bookSeats method checks each seat in the seatIds list to ensure that it is not already booked by calling the isBooked() method. If any seat is already booked, a SeatAlreadyBookedException is thrown, preventing double booking.

8. How does the service handle booking limits for users?

Answer: The service uses the getTotalBookedSeatsByCustomer helper method to calculate how many seats a customer has already booked. It then checks if booking the new seats would exceed the MAX_SEATS_PER_USER limit. If the limit is exceeded, a SeatLimitExceededException is thrown.

9. What happens if the customer ID provided in the bookSeats method is invalid?

Answer: If the customer ID does not exist in the CustomerRepository, the bookSeats method throws a CustomerNotFoundException, which is caught in the controller, and a 400 Bad Request response is returned with the appropriate error message.

10. Why is the seatRepository.saveAll(seats) method used after marking the seats as booked?

Answer: After marking the seats as booked, the seatRepository.saveAll(seats) method is used to persist the updated seat information to the database. This ensures that the seat status changes (from available to booked) are stored correctly.

These questions cover a range of topics, including Spring Boot, exception handling, transactions, and the logic behind booking limits, helping to assess understanding of both the code and key concepts in Java and Spring Boot.

1. **@RestController**:
   - Combines @Controller and @ResponseBody in Spring. It is used to define a controller that handles HTTP requests and returns data (typically JSON or XML) directly as the response body.
2. **@RequestMapping("/seats")**:
   - Specifies the base URL path for the controller. In this case, all endpoints inside the SeatBookingController will be prefixed with /seats.
3. **@GetMapping("/available")**:
   - Maps HTTP GET requests to the getAvailableSeats() method. The URL for this method will be /seats/available.
4. **@PostMapping("/book")**:
   - Maps HTTP POST requests to the bookSeats() method. The URL for this method will be /seats/book.
5. **@RequestParam**:
   - Used to extract query parameters from the request URL. In this case, it is used to retrieve the customerId from the URL in the bookSeats() method.
6. **@RequestBody**:
   - Binds the HTTP request body to a method parameter. In this code, it maps the list of seat IDs from the request body to the seatIds parameter in the bookSeats() method.
7. **@RequiredArgsConstructor** (from Lombok):

    - Automatically generates a constructor with parameters for all final fields. This simplifies dependency injection by eliminating the need for manually writing constructors in both SeatBookingController and SeatBookingService.

8. **@Transactional**:
    - Ensures that the method's operations are performed in a single database transaction. If an exception occurs, any changes made within the transaction are rolled back to maintain consistency.

These annotations streamline the development process by reducing boilerplate code and managing HTTP requests, transactions, and dependency injection in a Spring Boot application

Working of the Booking Model Class

The Booking class represents an entity in the database, mapped to a table that stores information about seat bookings. Here's how it works:

- **@Entity**: This annotation marks the class as a JPA entity, meaning it will be mapped to a database table (in this case, likely named booking).
- **@Id**: The bookingId field is marked as the primary key for the Booking table.
- **@GeneratedValue(strategy = GenerationType.IDENTITY)**: The primary key is auto-generated by the database.
- **@ManyToOne**: The customer field represents a many-to-one relationship with the Customer entity. This means that many bookings can belong to one customer.
- **@OneToMany(cascade = CascadeType.ALL)**: The seats field represents a one-to-many relationship with the Seat entity. This means that one booking can contain many seats. The cascade = CascadeType.ALL ensures that any changes to a Booking will also affect its associated seats (e.g., if a booking is deleted, its seats are also deleted).
- **bookingTime**: Stores the date and time when the booking was made, represented using LocalDateTime.
- **isPaymentConfirmed**: A boolean flag that indicates whether the payment for the booking has been confirmed or not, initially set to false.

How Lombok Reduces Code

Lombok simplifies the creation of typical boilerplate code, such as getters, setters, constructors, and toString methods. Here's how Lombok annotations are used in this class:

- **@Data**: This is a composite annotation that automatically generates several things for the class:
    - Getters for all fields.
    - Setters for all non-final fields.
    - toString(), equals(), and hashCode() methods.
    - A RequiredArgsConstructor (constructor with required arguments, though in this case, AllArgsConstructor and NoArgsConstructor override this).

Without Lombok, you would have to write all of these methods manually, which would increase the size of your code significantly.

- **@NoArgsConstructor**: Generates a no-argument constructor, which is often required by JPA for entity classes.
- **@AllArgsConstructor**: Generates a constructor that takes one argument for each field in the class. This is useful when you want to quickly create an object with all of its fields initialized.

How Lombok Works in This Code

With Lombok, you don't need to explicitly write the following methods or constructors:

1. **Getters and Setters**:
    - Without Lombok, you'd have to manually write getter and setter methods for each field (bookingId, customer, seats, bookingTime, and isPaymentConfirmed).

java

Copy code

```
public Long getBookingId() {
    return bookingId;
}


public void setBookingId(Long bookingId) {
    this.bookingId = bookingId;
}
// Similar for other fields...
```

2. **Constructors**:
    - You would have to manually write a no-args constructor and a constructor that takes all fields as arguments.

java

Copy code

```
public Booking() {
    // No-args constructor
}


public Booking(Long bookingId, Customer customer, List<Seat> seats, LocalDateTime bookingTime, boolean isPaymentConfirmed) {
    this.bookingId = bookingId;
    this.customer = customer;
    this.seats = seats;
    this.bookingTime = bookingTime;
    this.isPaymentConfirmed = isPaymentConfirmed;
}
```

3. **Other Methods**:
    - Lombok automatically generates equals(), hashCode(), and toString(), reducing code clutter. Without Lombok, you'd have to write these methods yourself if you need them, especially in cases where you need to compare objects or log them.

Lombok Behind the Scenes

Lombok uses **annotation processing** at compile time. When you build your project, Lombok generates the required methods (like getters, setters, constructors) in the bytecode, even though you don't see them in the source code. This approach keeps your codebase clean while still providing all the required functionality under the hood.

In short, Lombok helps reduce boilerplate code significantly, making the code cleaner and easier to maintain without compromising functionality.

4o