In-Class Exercise – Implementing a Many-to-Many Relationship with a Rental Entity

Background

In a video rental store, customers can rent movies. For this exercise, each movie has only **one copy** available. A movie can be rented **only if it's not already rented**. We want to track which customers have rented which movies, and when.

---

1. Create a Rental Entity

Design a `Rental` entity that models the rental of a movie by a customer. It should include:

- A unique ID.
- A reference to the customer who rented the movie.
- A reference to the movie being rented.
- A rental date (when the movie was rented).
- An optional return date (when the movie was returned).
- The number of days the movie is rented.
- A rental status using an enum (e.g., `ACTIVE`, `RETURNED`, `PENALTIES`, `OVERDUE`).

---

2. Model the Relationships

- Set up a many-to-many relationship **through the Rental entity**.
- Each customer can have multiple rentals over time.
- Each movie can be rented by different customers over time, but only one at a time.
- Use `@ManyToOne` for both the customer and movie within the `Rental` entity.

---

3. Update Movie Availability Logic

- A movie should only be rented if it is not currently rented out (`isRented = false`).
- When rented:
  - Mark the movie as unavailable (`isRented = true`).
- When returned:
  - Mark the movie as available again (`isRented = false`).

---

4. Create Repository, Service, and Controller

`RentalRepository`

- Add query methods to:
  - Find rentals by customer, movie, date range, and rental status.
  - Count rentals based on different filters (e.g., by status or customer).
  - Perform aggregate queries like average rental duration or grouped counts.

---

5. Repository and Service Methods

`RentalService`

Implement business logic for handling rental actions. Include methods that:

- **Rent a movie if available, and update its rented status.**
  Implement **two overloaded methods** for this:
  1. One that takes `Movie`, `Customer`, and `days` as arguments.
  2. One that takes `movieId`, `customerId`, and `days` as arguments.
     In both methods:
     - Check if the movie is already rented (`movie.isRented()`), and throw an exception if so.
     - Create a new `Rental` object, mark the movie as rented, and save both movie and rental.
- Return a movie and update the return date and rental status.
  - If returned late, set the status to `PENALTIES`; otherwise, `RETURNED`.
  - Also update the movie's rented status to `false`.
- Retrieve rentals filtered by:
  - Customer ID
  - Movie ID
  - Date range
  - Rental status
- Count and analyze rentals using:
  - Total count
  - Count by status or customer
  - Grouped counts (e.g., per movie or status)
  - Average rental duration
- Use `MovieService` and `CustomerService` to fetch related entities by ID when needed (e.g., in the ID-based rental method).

---

6. Controller Endpoints

`RentalsController`

Expose REST endpoints under `/api/rentals` for rental operations:

Method	Endpoint	Description
`GET`	`/api/rentals`	Get all rentals
`POST`	`/api/rentals/rent`	Rent a movie
		Request body (JSON): `{ "customerId": 1, "movieId": 2, "daysRented": 5 }`
`POST`	`/api/rentals/return/{rentalId}`	Return a movie
`GET`	`/api/rentals/customer/{customerId}`	Get rentals for a customer
`GET`	`/api/rentals/movie/{movieId}`	Get rentals for a movie

GET	/api/rentals/daterange?start=YYYY-MM-DD&end=YYYY-MM-DD	Rentals in a date range
GET	/api/rentals/active	Currently active rentals
GET	/api/rentals/due-today	Rentals due today
GET	/api/rentals/count	Total rental count
GET	/api/rentals/count/status/{status}	Count by rental status
GET	/api/rentals/count/customer/{customerId}	Count by customer
GET	/api/rentals/average-duration	Average number of days rented
GET	/api/rentals/count/grouped-by-status	Count grouped by status
GET	/api/rentals/count/per-movie	Count grouped by movie
