Lecture 8

Hibernate and Object-Relational Mapping (ORM)

- **Hibernate** is a popular Java framework that automatically maps Java objects to database tables (ORM).
- Hibernate is the most popular implementation of the JPA specification.
- **Key Benefit:** Hibernate eliminates the need for writing most manual SQL queries, allowing developers to work primarily with Java classes.
- **What Hibernate Handles:**
  - Table creation and schema updates.
  - Query generation (like SELECT, INSERT, UPDATE, DELETE).
  - Managing relationships (e.g., `@OneToMany`).
  - Transaction management and caching.
- In Spring Boot, Hibernate is included via the `spring-boot-starter-data-jpa` dependency.

Setting up JPA Entities

- **JPA Entities** are Java classes that map directly to database tables.
  - The properties (fields) of the Java class map to the columns of the table.
- **Key Annotations:** Annotations define how Java objects map to the database schema.
  - `@Entity`: Marks a Java class as an object that should be managed by JPA/Hibernate (a database table).
  - `@Id`: Designates the primary key of the entity.
- These annotations tell JPA to automatically manage table creation and primary key mapping.

Database Setup and Configuration

- **Required Dependencies:** You need Spring Data JPA and a database driver (like MySQL Driver).
- **Configuration in application.properties:** You must configure the datasource and Hibernate.
- **Schema Generation (ddl-auto):** This property controls how Hibernate manages the database schema.
    - `spring.jpa.hibernate.ddl-auto=create-drop`: **Used for learning/labs.** Spring Boot creates tables automatically when the app starts, based on the `@Entity` classes, and then deletes (drops) all those tables when the application stops. This guarantees a fresh database every run.
    - `spring.jpa.hibernate.ddl-auto=update`: **Used in real-world applications** to update the existing schema.

Spring Data Repositories (The Data Access Layer)

- A **Spring Data Repository** is the component that handles data access in Spring Data JPA. It is a very important concept for Backend development.
- It provides **CRUD** (Create, Read, Update, Delete) operations.
- **Why use Repositories?** They simplify database operations, reduce boilerplate code, and provide query abilities using method names.

  Key Repository Interfaces

  1. **CrudRepository**: The most basic interface, providing fundamental CRUD methods.
      - **save(S entity):** Saves a new object or updates an existing one.
      - **findById(ID id):** Retrieves an entity using its primary key.
      - **findAll():** Retrieves all entities.
      - **deleteById(ID id):** Deletes an entity using its primary key.
  2. **JpaRepository**: Extends `CrudRepository` and adds methods for pagination and sorting.
      - **findAll(Sort sort):** Retrieves all entities sorted by a specified field.

- **findAll(Pageable pageable):** Retrieves entities with pagination capabilities.

How to Create and Use a Repository

- You only need to **define an interface** that extends `CrudRepository` or `JpaRepository`, specifying the Entity type and the type of its primary key (e.g., `public interface HotelRepository extends CrudRepository<Hotel, Integer>`).
- **Spring Boot handles the implementation** automatically at runtime.

Query Methods

- **Query Methods** allow developers to retrieve data by defining methods in the repository interface using specific naming conventions, eliminating the need for raw SQL.
- **Naming Convention:** Query methods follow the pattern `findBy` + `[Property Name]` + `[Operation]`.
  - *Example:* `findByName(String name)`.
- **Keywords Supported:**
  - **Logical:** `And`, `Or` (e.g., `findByNameAndCategory`).
  - **Comparison:** `GreaterThan`, `LessThan`, `Between`.
  - **String Operations:** `Containing`, `Like`, `StartsWith`, `EndsWith` (e.g., `findByTitleContaining`).
  - **Sorting:** `OrderBy` (e.g., `findByCategoryOrderByNameAsc`).

Integrating Repositories into Applications

- Integrating repositories means using them within your application's service or controller layers to switch from static, in-memory data storage to persistent database storage.
- **Steps to Integrate:**

  1. Define the repository interface (as mentioned above).

  2. **Inject the Repository:** Use the `@Autowired` annotation to inject the repository into your service or controller (e.g., a `@RestController`).

3. **Use Methods:** Call repository methods (`save`, `findAll`, etc.) within your application logic to interact with the database.

    - This integration ensures scalability and allows business logic to focus on core tasks rather than low-level database details.

Data Relationships and Advanced Configuration

- Relationships define how entities connect to each other.
    - **One-to-One:** One entity is associated with one other entity. (`@OneToOne`).
    - **One-to-Many / Many-to-One:** One entity is associated with many others.
        - `@OneToMany`: Used on the "one" side (e.g., a Hotel has many Rooms).
        - `@ManyToOne`: Used on the "many" side (e.g., a Room belongs to one Hotel). The `@ManyToOne` side usually uses `@JoinColumn` to define the foreign key.
- **Many-to-Many:** Many entities are associated with many others. (`@ManyToMany`).
    - This requires an intermediate **Join Table** in the database, specified using the `@JoinTable` annotation, which links the two entities.

Bidirectional Relationships

- In a **bidirectional relationship** (where both entities know about each other), you must use the `mappedBy` attribute on the non-owning side.
- `mappedBy` indicates the field in the related entity that owns the relationship.
- Using `mappedBy` is crucial to avoid inefficiency and complexity issues caused by both sides trying to take ownership.

Cascading Operations

- Cascading allows an operation performed on a parent entity to automatically apply to its related child entities.
- **Common Values:**
    - `CascadeType.ALL`: Cascades all operations (PERSIST, MERGE, REMOVE, REFRESH).

- `CascadeType.PERSIST`: Saves the child entity when saving the parent.
- `CascadeType.REMOVE` / `DELETE`: Deletes the child entity when the parent is deleted (e.g., deleting a Post deletes all its Comments).

Orphan Removal

- If set to `true`, **orphan removal** automatically deletes child entities that are detached (no longer associated) from their parent.

Fetching Associations

- Fetching controls whether related entities are loaded immediately with the parent or only when needed.
  - **FetchType.LAZY**: Loads associated objects only when they are explic itly requested.
  - **FetchType.EAGER**: Loads associated objects immediately along with the parent object.

Common Mistakes & Debugging Tips

- **Missing Annotations:** Errors occur if `@Entity` or `@Id` are missing from the corresponding Java class.
- **Wrong Imports:** Avoid importing `java.lang.Module` (part of the JDK module system) if you are trying to reference your own entity class named `Module`. JPA will fail because your entity is not correctly annotated.
- **Configuration Scanning:** By default, Spring Boot only scans for annotated `@Entity` classes within the package hierarchy located below the main `@SpringBootApplication` class.
- **Debugging SQL:** Use `'logging.level.org.hibernate.SQL=debug'` in configuration to inspect the actual SQL queries being generated by Hibernate.

Analogy for ORM (JPA/Hibernate):

JPA and Hibernate act like an **international shipping courier** for your application data. When you want to send a Java object (like a package) across the border to the Database (a different country), the courier (ORM) automatically handles all the necessary steps: translating your plain Java object into the proper SQL format (customs paperwork), moving it across the network, and ensuring it lands safely in the correct database table (storage locker). You, the developer, just deal with the friendly courier interface (Java objects and repository methods), not the complex foreign language (SQL) or logistics underneath.