Database Integration with Spring Data JPA

In this section, we explored how to integrate a relational database into a Spring Boot application using Spring Data JPA. We covered everything from designing and managing databases to interacting with them efficiently using repositories and dynamic queries.

1. Data Access Technologies

There are different ways to interact with a database in Java:

- **JDBC:** A low-level API that requires writing raw SQL and manually handling database connections.
- **JPA:** An abstraction over JDBC that allows us to work with databases using Java objects.
- **Hibernate:** The most popular JPA implementation, handling object-relational mapping (ORM) and query generation.
- **Spring Data JPA:** A Spring project built on top of JPA/Hibernate that further simplifies data access by providing repository interfaces and additional functionality like sorting and pagination.

2. Designing the Database

There are two workflows for designing a database:

- **Database-first:** We define tables using SQL DDL statements or visual tools, then create Java entities to map to them.
- **Model-first:** We define Java entities first, and Hibernate generates the database schema. For real-world applications, database-first gives us more control and better versioning.

3. Managing the Database Schema

- We can create database tables visually using IntelliJ’s database tool or write SQL DDL statements.

- Flyway helps us version our database by applying migrations, ensuring consistency across environments.
- Migrations are automatically applied when the application starts, but we can also run them manually using the Maven plugin.
- When modifying the schema, we never edit existing migrations. Instead, we create new migration files to track changes properly.

4. Defining Entities

- In JPA, we map Java classes to database tables using annotations like `@Entity`, `@Table`, and `@Column`.
- We can simplify entity classes using Lombok, eliminating repetitive getters, setters, and constructors.
- JPA Buddy allows us to quickly generate entities, repositories, and queries.
- Hibernate can auto-generate database tables from entities, but this should only be used for prototyping, not production.

5. Working with Repositories

- A repository is an abstraction that allows us to interact with the database using object-oriented methods.
- With Spring Data JPA, we only define repository interfaces to perform CRUD operations, and Spring automatically generates their implementations at runtime.
- There are different types of repository interfaces, but the most common ones are:
  - `CrudRepository`: Provides basic CRUD functionality.
  - `JpaRepository`: Extends CrudRepository and adds additional features like pagination, sorting, and batch operations.
- Under the hood, these repositories use `EntityManager`, which is a JPA interface responsible for managing database operations and tracking entity states.
- Internally, `EntityManager` maintains a persistence context, which is a container that keeps track of entities managed by Hibernate.

6. Understanding Entity States and Transactions

- Entities can be in different states:
    - **Transient:** Not yet saved to the database.
    - **Persistent:** Managed by Hibernate and tracked in the persistence context.
    - **Detached:** No longer tracked by Hibernate but still exists in the database.
    - **Removed:** Scheduled for deletion.
- The persistence context is tied to a transaction, meaning any changes made to an entity within a transaction are automatically saved when the transaction commits.
- When the transaction is complete, the persistence context is cleared, and entities that were managed within that transaction become detached.
- Repository methods are transactional by default, meaning each method runs within a separate transaction unless we override this behavior using `@Transactional`.

7. Fetching Data Efficiently

- There are two fetch strategies when it comes to loading related objects:
    - **Eager loading:** The related entity is loaded immediately along with the parent entity.
    - **Lazy loading:** The related entity is not loaded until explicitly accessed.
- The default fetch type depends on the relationship:
    - `@OneToMany` and `@ManyToMany` → Lazy by default
    - `@ManyToOne` and `@OneToOne` → Eager by default
- We can override fetch strategies using `FetchType.LAZY` or `FetchType.EAGER`.
- Lazy loading can improve performance by preventing unnecessary data from being loaded, but it can also lead to the N+1 problem.
- The N+1 problem happens when an initial query fetches a list of entities (1 query), and then additional queries (N queries) are executed to load each related entity individually.
- We can fix this using eager loading in the relationship or `@EntityGraph` to load related entities in a single optimized query.

8. Writing Custom Queries

- There are two ways to write custom queries in Spring Data JPA:
  - **Derived queries:** We follow a set of naming conventions, and Spring automatically generates the query (eg `findByEmailOrderByName`).
  - **Custom query methods:** We use the `@Query` annotation to define our own queries.
- The `@Query` annotation allows us to write custom queries using JPQL or native SQL, giving us more control over how data is retrieved.
- **JPQL (Jakarta Persistence Query Language)** is an object-oriented query language designed for querying entities rather than database tables. Since JPQL is database-agnostic, it works across different relational databases, but it has limitations compared to native SQL when it comes to database-specific features.
- Projections let us fetch only specific fields instead of full entities, improving performance.
- We can call stored procedures using Spring Data JPA for database-side logic execution.

9. Dynamic Queries

- Derived and custom query methods are useful for common database operations, but sometimes we need to fetch data dynamically based on user input.
- Instead of hardcoding multiple query methods in our repositories, we can use dynamic queries to construct queries at runtime, making our code more flexible and maintainable.
- **Query by Example (QBE)** allows us to build queries dynamically using an example object instead of writing SQL.
- **Criteria API** lets us construct queries programmatically for complex filtering.
- **Specifications API** builds on top of Criteria API and allows us to compose reusable query conditions.
- We can sort and paginate results using `Sort` and `Pageable` objects, making it easier to handle large datasets.