Class-23: JPA Entities & Repositories

by Pial Kanti Samadder

Entity

- Entities in JPA are nothing but POJOs representing data that can be persisted in the database.
- An entity represents a table stored in a database.
- Every instance of an entity represents a row in the table.

The *Entity* Annotation

Let's say we have a POJO called *Student,* which represents the data of a student, and we would like to store it in the database.

So let's define it by making use of the *@Entity* annotation. We must specify this annotation at the class level. **We must also ensure that the entity has a no-arg constructor and a primary key:**

```
@Entity
public class Student {
   // fields, getters and setters
}
```

Entity classes must not be declared *final*.

The *Id* Annotation

Each JPA entity must have a primary key that uniquely identifies it. The *@Id* annotation defines the primary key. We can generate the identifiers in different ways, which are specified by the *@GeneratedValue* annotation.

We can choose from four id generation strategies with the *strategy* element. **The value can be *AUTO, TABLE, SEQUENCE,* or *IDENTITY:***

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    private String name;

    // getters and setters
}
```

If we specify `GenerationType.AUTO`, the JPA provider will use any strategy it wants to generate the identifiers.

Primary Key Generation Strategies in JPA

JPA provides four strategies to generate primary keys for entities using the `@GeneratedValue` annotation.

Syntax

```
@Id
@GeneratedValue(strategy = GenerationType.XXX)
private Long id;
```

GenerationType.AUTO

- This is the **default** strategy.
- JPA will **choose the strategy based on the underlying database** and the JPA provider (like Hibernate).
- Example: Hibernate may choose `SEQUENCE` for PostgreSQL, or `IDENTITY` for MySQL.

Example

```
@Entity
public class Product {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
}
```

Real-Time Use Case:

- Good for **portable applications** where the DBMS is not fixed.
- For example, in a **multi-tenant SaaS product** where customers may use different databases

GenerationType.IDENTITY

- Uses the database's **auto-increment** feature.
- Primary key is generated by the DB at **insert time**.
- You can't batch insert with Hibernate using this strategy (limitation).

Example (MySQL/PostgreSQL)

```
@Entity
public class Customer {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
}
```

Real-Time Use Case:

- Used when your database supports auto-increment (e.g., **MySQL**, **PostgreSQL**).
- Common for **simple apps**, **CRUD apps**, or when inserting records one at a time.

Note:

- Since ID is generated after insert, Hibernate can't predict it before saving, making batch inserts tricky.

GenerationType.SEQUENCE

- Uses a **database sequence object**.
- Hibernate first fetches the next sequence value, then inserts the record.
- Allows **batch insert**.

Example (PostgreSQL, Oracle):

```
@Entity
public class Order {
   @Id
   @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "order_seq")
   @SequenceGenerator(name = "order_seq", sequenceName = "order_sequence", allocationSize = 1)
   private Long id;
}
```

Real-Time Use Case:

- Best for **enterprise systems** using **PostgreSQL or Oracle**, where performance and batch inserts are important.
- Example: A **banking system** or **e-commerce** app managing millions of transactions.

Tip:

- You can increase `allocationSize` for fewer DB calls and better performance in batch operations

GenerationType.TABLE

- Uses a **separate table** to simulate sequence behavior.
- Useful in **non-standard or older databases** that don't support `SEQUENCE` or `IDENTITY`.

Example:

```
@Entity
public class Invoice {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "invoice_gen")
    @TableGenerator(name = "invoice_gen", table = "id_gen", pkColumnName = "gen_name", valueColumnName = "gen_val",
pkColumnValue = "invoice_id", allocationSize = 1)
    private Long id;
}
```

Real-Time Use Case:

You're using an old or **non-standard database** that doesn't support sequences or auto-increment. You create a table called `id_generator` that stores the current ID value for each entity.

The *Table* Annotation

In most cases, **the name of the table in the database and the name of the entity won't be the same.**

In these cases, we can specify the table name using the *@Table* annotation:

```
@Entity
@Table(name="students")
public class Student {
    // fields, getters and setters
}
```

We can also mention the schema using the *schema* element:

```
@Entity
@Table(name="students", schema="school")
public class Student {
    // fields, getters and setters
}
```

Schema name helps to distinguish one set of tables from another.

If we don't use the *@Table* annotation, the name of the table will be the name of the entity.

The *Column* Annotation

Just like the *@Table* annotation, we can use the *@Column* annotation to mention the details of a column in the table.

The *@Column* annotation has many elements such as `name`, `length`, `nullable`, *and* `unique`:

```
@Entity
@Table(name="students")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name="student_name", length=50, nullable=false, unique=false)
    private String name;

    // other fields, getters and setters
}
```

The *name* element specifies the name of the column in the table. The `length` element specifies its length. The *nullable* element specifies whether the column is nullable or not, and the *unique* element specifies whether the column is unique.

If we don't specify this annotation, the name of the column in the table will be the name of the field.

The *Transient* Annotation

Sometimes, we may want to **make a field non-persistent.** We can use the *@Transient* annotation to do so. It specifies that the field won't be persisted.

For instance, we can calculate the age of a student from the date of birth.

So let's annotate the field *age* with the *@Transient* annotation:

```
@Entity
@Table(name="students")
public class Student {
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private Long id;

   @Column(name="name", length=50, nullable=false)
   private String name;

   @Transient
   private Integer age;

   // other fields, getters and setters
}
```

As a result, the field *age* won't be persisted in the table.

What is a Transaction?

A **transaction** is a set of operations that must either **all succeed** or **all fail** as one unit.

🏦 Example: Bank Transfer

- **Step 1:** Deduct money from Account A
- **Step 2:** Add money to Account B
  If one step fails, the **entire operation rolls back** — ensuring data consistency.

⚙️ In Spring Framework

A transaction groups database actions (e.g., `INSERT`, `UPDATE`, `DELETE`) so that:

- ✅ All succeed → **Changes are committed**
- ❌ Any fails → **All changes are rolled back**

🔒 Goal

Ensure **data integrity**, **consistency**, and **reliability** in your application.

ACID Properties of Transactions

Transactions usually use these four rules, known as **ACID**:

A — Atomicity

- All operations in a transaction are **executed as one unit**.
- Either **everything succeeds** or **nothing happens**.
- On failure → all changes are **rolled back**.

C — Consistency

- The database moves from **one valid state to another**.
- All **rules, constraints, and relationships** remain intact.
- Ensures **data integrity** after each transaction.

I — Isolation

- Each transaction runs as if it's **the only one** in the system.
- Prevents **conflicts and interference** from concurrent transactions.
- Changes are **invisible to others** until committed.

D — Durability

- Once committed, changes are **permanent**.
- Data **survives crashes or power failures**.
- The system **recovers to a consistent state** on restart.

Why Do We Need Transactions?

Without transactions, multi-step or dependent operations can lead to **data inconsistency**.

Example: Bank Transfer

If money is deducted from the sender's account but **not credited** to the receiver's — the system ends up with **corrupted data**.

Transactions Ensure:

- **All-or-nothing execution** (maintains data integrity)
- **Automatic rollback** on failure
- **Consistent database state**
- **Controlled concurrency** through isolation levels
- **Flexible behavior** using propagation settings

Spring provides built-in transaction management features like **rollback rules**, **isolation levels**, and **propagation behaviors**, making it easier to maintain reliable and consistent data operations.

Stages of a Transaction

1. **Start**
   - Begins when a **transactional method** starts executing.
   - All database operations within the method become part of **one transaction**.
   - Changes are **temporary** until committed.
2. **Commit**
   - Happens when the method **completes successfully** (no exceptions).
   - All changes are **saved permanently** to the database.
   - The transaction is now **complete and closed**.
3. **Rollback**
   - Triggered when an exception occurs during execution.
   - Undoes all changes made within the transaction.
   - Returns the database to its original state.
   - By default, Spring rolls back on unchecked (runtime) exceptions. You can customize this with the `rollbackFor` attribute.

Transactions in Spring

The `@Transactional` annotation in Spring Framework simplifies working with transactions.

Instead of manually writing code to start, commit or rollback transactions, `@Transactional` handles all of that, and more, for us.

How to Use @Transactional in Java Spring Methods

Using @Transactional is very simple in the Spring Framework, all you have to do is add the annotation to a method, most of the time it should be a service layer method. That essentially says Spring to make this method transactional.

Below is an example of using @Transactional in a simple mock example of using transactions for money transferring in the BankService:

```
public class BankService {
    @Transactional
    public void transferMoney(User sender, User recipient, double amount) {
        removeFrom(sender, amount);
        addTo(recipient, amount);
    }

    private void removeFrom(User user, double amount) {
      // ... some logic for handling it would go here
    }

    private void addTo(User user, double amount) {
      // ... some logic for handling it would go here
    }
}
```

In this example, we have two methods which will be encapsulated within an transaction.

- When the transferMoney method is called, @Transactional does all the work and transforms the operation into a transaction. The method is now a transaction in the start phase.
- If both the removeFrom and the addTo methods go through successfuly without error, the transaction is committed. That is also known as commit stage.
- If there is an error in either of the methods, the transaction is rolled back to prevent partial updates. In case of an error, the transaction goes into the rollback stage.

Considerations for `@Transactional`

Best Practices

- **Apply on service layer methods** containing business logic, not repositories.
- Use `@Transactional(readOnly = true)` for **read-only operations**.
- Configure **propagation**, **isolation**, and **rollback rules** for advanced scenarios.

Important Considerations

- **Private methods:** Transactions **do not apply** (pre-Spring 6) due to proxy limitations.
- **Internal method calls:** Calling a transactional method **within the same class bypasses the proxy**, so `@Transactional` is ignored.
- **Repositories:** Spring Data JPA auto-manages CRUD transactions, but **custom behavior still requires explicit annotation**.

Key Takeaway

- **Transactional methods work only when called from outside the same class.**
- **Why:** Spring wraps your service in a “transaction manager” behind the scenes. Calls made **inside the same class** bypass this wrapper, so transactions are **not applied**.
- **Tip:** Always call transactional methods from another bean or from the client code, not from a private/internal method in the same class.

What is Spring Data JPA Repository

Goal: Simplify database access by reducing boilerplate code.

Key Idea:
Spring Data JPA automatically provides CRUD (Create, Read, Update, Delete) operations without writing SQL or implementation code.

Core Interface Hierarchy:

```
Repository
  └── CrudRepository
      └── PagingAndSortingRepository
          └── JpaRepository
```

JpaRepository adds:

- Pagination and sorting support
- Batch operations
- `Flush` and `saveAndFlush` methods

Example:

```
public interface UserRepository extends JpaRepository<User, Long> {
}
```

What Happens Under the Hood

Spring Magic

- When your app starts, Spring creates a **proxy implementation** of the repository interface.
- This proxy handles:
  - Database connections (via `EntityManager`)
  - Transactions (auto-managed for CRUD)
  - Query generation based on method names

Example:

```
userRepository.save(user); // Proxy executes SQL INSERT
```

No manual JDBC or SQL needed!

In summary

- `JpaRepository` is a high-level abstraction over JPA's `EntityManager`.
- Spring auto-generates implementations for your interfaces.
- Transactions are handled automatically for CRUD operations.

Running Sample Project

- ## Clone the Repository

If you haven't cloned the repository yet, run the following command (ensure `git` is installed):

```
git clone https://github.com/PialKanti/Ostad-SpringBoot-Course.git
```

Then switch to the correct branch for today's class (replace with the actual branch name, e.g., `class-23-jpa-entity`):

```
git fetch
git switch class-23-jpa-entity
```

Or,

If You Already Have the Repository Cloned, simply open your existing project folder and switch (or checkout) to the appropriate branch:

```
git fetch
git switch class-23-jpa-entity
```

- ## Set Up and Run PostgreSQL Database

You can run PostgreSQL **either via Docker** or a **desktop installation**.

Option 1: Run via Docker

A `compose.yml` file is available in the root of the repository.
Run the following command from the project root:

```
docker compose up -d
```

This will start a PostgreSQL container automatically.

Or,

Option 2: Run via PostgreSQL Desktop (Manual Setup)

If you already have PostgreSQL installed locally:

1. Start your PostgreSQL server.
2. Create a new database named `crud_db` if not exists.

- ## Open the Project in IntelliJ IDEA

1. Open IntelliJ IDEA.
2. Click **File → Open** and select the `crud-sample` folder inside the repository.
3. Let IntelliJ import Maven/Gradle dependencies automatically.