Spring ORM Frameworks

* ORM, Spring Data JPA and Hibernate
* Perform CRUD operations against a database with two simple steps
* Configure auto generated IDs for the Primary Key fields
* Realize the power of Spring Data Finder methods
* Load data from database without implementing any code or SQL
* Learn and use JPQL - Java Persistence Query Language
* Execute native Sql queries from your Spring Data Application
* Use Paging and Sorting
* Hibernate Associations and implement all the four types of associations

**1. ORM Basics:**

* **ORM (Object Relational Mapping)**
* **What is JPA?**
* **What and why Spring Data?**

**2. Simple CRUD Operations:**

Project - **Productdata**

Use case:

Product: Product – Table Name

DB

Product - Entity Class

Product Repository – Repository Interface

Configure the Data Source

Step 1: Create the database and tables.

Product.sql

Step 2: Create the Entity class.

Step 3: Create the Repository.

Step 4: Configure the Data Source.

Step 5: Test the Application.

Create/Read/Update/Delete/Exists

3. ID Generators - Project – Id Generator

Product product = new Product();

product.setId(123);

product.setName("Iphone");

repository.save(product);

GenerationType.AUTO

GenerationType.IDENTITY

GenerationType.SEQUENCE

GenerationType.TABLE

@Id

@GeneratedValue(Strategy = GenerationType.AUTO)

int id;

**Persistence provider** refers to the specific JPA implementation used in our application to persist objects to the database

Database

**GenerationType.AUTO**

**Persistence provider**

GenerationType.IDENTITY

GenerationType.SEQUENCE

|  |  |  |
| --- | --- | --- |
| Id |  |  |
|  |  |  |
|  |  |  |

GenerationType.TABLE

**GenerationType.IDENTITY**

Database

**Persistence provider**

|  |  |  |
| --- | --- | --- |
| Id |  |  |
|  |  |  |
|  |  |  |

PRIMARY KEY – Auto Increment

**GenerationType.TABLE**

Database

**Persistence provider**

|  |  |  |
| --- | --- | --- |
| Id |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| MYID |  |  |
|  |  |  |
|  |  |  |

Special Table ----🡪

Create the Database Schema:

ID Gen Employee.sql

Example:

“Table" and "identity" refer to strategies for generating primary key values for entity instances.

**GenerationType.TABLE**

The **GenerationType.TABLE** strategy uses a separate table in the database to generate primary key values. This table, often called a sequence table or key allocation table, keeps track of the next available primary key.

**Advantages**:

* Works consistently across different databases.
* Provides flexibility in key generation.

**Disadvantages**:

* Can introduce additional overhead due to the separate table lookups.
* May have performance implications for high-concurrency applications.

### GenerationType.IDENTITY

The **GenerationType.IDENTITY** strategy relies on the database to automatically generate the primary key values, typically using auto-increment columns. This strategy delegates the responsibility of primary key generation to the database.

**Advantages**:

* Simple and efficient.
* No need for additional tables or sequences.

**Disadvantages**:

* Limited flexibility; only works with databases that support auto-increment columns.

4. Spring Data Finder Methods:

Finder Methods

product

id

name

desc

price

ProductRepository

List<product> findByName(String name);

List<product> findByNameAndDesc(String name, String desc);

List<product> findByDescLike(String desc);

List<product> findByPriceGreaterThan(Double price);

Spring data jpa documentation :

**https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repository-query-keywords**

Paging and Sorting:

JPQL

**Java Persistence Query Language**

Employee

Int id;

String firstName;

String lastName;

Object based Queries:

**employeetab**

|  |  |  |
| --- | --- | --- |
| Empid | Fname | lname |
|  |  |  |
|  |  |  |

Select \* from Employee;

Select firstName,lastName from Employee;

**Relationships In Hibernate:**

**Association Mapping: (Unidirectional & bidirectional)**

**order**

|  |  |  |  |
| --- | --- | --- | --- |
| Id | Productname | Description | shippingaddress |
|  |  |  |  |
|  |  |  |  |

**Order**

|  |  |  |
| --- | --- | --- |
|  | Address\_Id | Product\_id |
|  |  |  |
|  |  |  |

**product**

|  |  |  |
| --- | --- | --- |
| id |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| Id | Street | city |
|  |  |  |
|  |  |  |

**Address**

One to One:

**Person**

Long id

License license

**License**

Long id

Person person

Many to Many:

Order

Long id

Set<Product> product

Product

Long id

String name

Set<Order> orders

One to Many & Many to One:

Customer

Long Id

Set<PhoneNumber> numbers

PhoneNumber

Long id

String number

@OneToOne

@OneTo Many

@ManyToOne

@ManyToMany

### Example Usage

#### @OneToMany and @ManyToOne

Let's consider an example of a bidirectional one-to-many relationship between Author and Book.

Author Entity

@Entity

public class Author {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "author")

private List<Book> books = new ArrayList<>();

// getters and setters

}

Book Entity:

@Entity

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

@ManyToOne

@JoinColumn(name = "author\_id")

private Author author;

// getters and setters

}

‘In this example:

* The Author entity has a @OneToMany relationship with the Book entity. The mappedBy = "author" attribute indicates that the author field in the Book entity is the owner of the relationship.
* The Book entity has a @ManyToOne relationship with the Author entity, and it includes a @JoinColumn annotation to specify the foreign key column (author\_id) in the Book table.

**Why Use mappedBy?**

* **Ownership**: It clarifies which entity is responsible for maintaining the relationship.
* **Bidirectional Relationships**: It helps in defining bidirectional relationships, ensuring both sides are aware of the relationship and can be navigated from either side.

In summary, mappedBy is a crucial attribute for defining and managing bidirectional relationships in JPA, ensuring clear ownership and proper mapping between entities.