FULL STACK DEV

ORM - JPA vs Hibernate

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

WHAT IS ORM -(1) ?

- Object-Relational Mapping maps objects (model or entity) to data (table) in a relational database.

- Essentially, it acts as a bridge between the object-oriented world of Java and the relational world of SQL databases.

- Reduces boilerplate code: You don't have to write repetitive SQL queries for database operations.

  -

WHAT IS ORM - (2)?

- Object-oriented approach: You work with Java objects directly, rather than raw SQL tables and rows.
- Database independence: With ORM, you can often switch between different relational databases with minimal code changes.
- Improved maintainability: Changes to the database schema can often be handled by updating entity mappings rather than with SQL.

JPA (CAKE BOOK)

- JPA (Java Persistence API):

- JPA is a specification (a set of interfaces and annotations) for managing relational data in Java applications. It defines how an ORM tool should behave.

- It's part of the Jakarta EE platform (formerly Java EE).

- JPA itself does not provide an implementation; it's just a standard.

```java
@Entity
public class Student {
    @Id
    private Long id;
    private String name;
}
```

HIBERNATE (MASTER CHEF)

- Hibernate is a popular, open-source implementation of the JPA specification.

- It's one of the most widely used ORM frameworks in the Java ecosystem.

- When you use Spring Data JPA, you are typically using Hibernate under the hood as the default JPA provider.

JPA defines the rules and Hibernate plays by those rules !!

```
Session session = sessionFactory.openSession();
session.beginTransaction();
session.save(student);
session.getTransaction().commit();
```

FULL STACK DEV

Setting up Spring Data JPA

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

JPA DEPEDENCIES

- To integrate Spring Data JPA and MySQL into your Spring Boot project, you need to add specific dependencies to your pom.xml file.
- Open pom.xml: Locate your pom.xml file in your project's root directory.
- Add Dependencies: Add the following dependencies within the <dependencies> section:

```
<!-- Spring Data JPA for database interaction -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

  -

DATABASE SPECIFIC CONFIG.

- <!-- MySQL JDBC Driver for connecting to MySQL database -->

```
<dependency>
    <groupId>com.mysql</groupId>
   <artifactId>mysql-connector-j</artifactId>
   <scope>runtime</scope>
</dependency>
```

- <!-- postgreSQL JDBC Driver -->

```
<dependency>
    <groupId>org.postgresql</groupId>
   <artifactId>postgresql</artifactId>
   <scope>runtime</scope>
</dependency>
```

STARTER & DRIVER

- **spring-boot-starter-data-jpa**
  - It brings in all the necessary dependencies for using Spring Data JPA, including Hibernate.
  - It provides auto-configuration for JPA and a convenient way to work with repositories.

- **mysql-connector-j**
  - This is the official JDBC (Java Database Connectivity) driver for MySQL.
  - It allows your Java application to connect to and communicate with a MySQL database.
  - The <scope>runtime</scope> means it's only needed at runtime, not during compilation.

CONFIGURATION IN APPLICATION.PROPERTIES

- spring.datasource.url=jdbc:mysql://localhost:3306/your_database_name?createDatabaseIfNotExist=true&useSSL=false&serverTimezone=UTC
- spring.datasource.username=root
- spring.datasource.password=your_mysql_root_password
- spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

- # JPA/Hibernate settings
- spring.jpa.hibernate.ddl-auto=update
- spring.jpa.show-sql=true
- spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
    -

FULL STACK DEV

Create a JPA Entity Class Student for Entity Mapping with mySQL

Presented by:

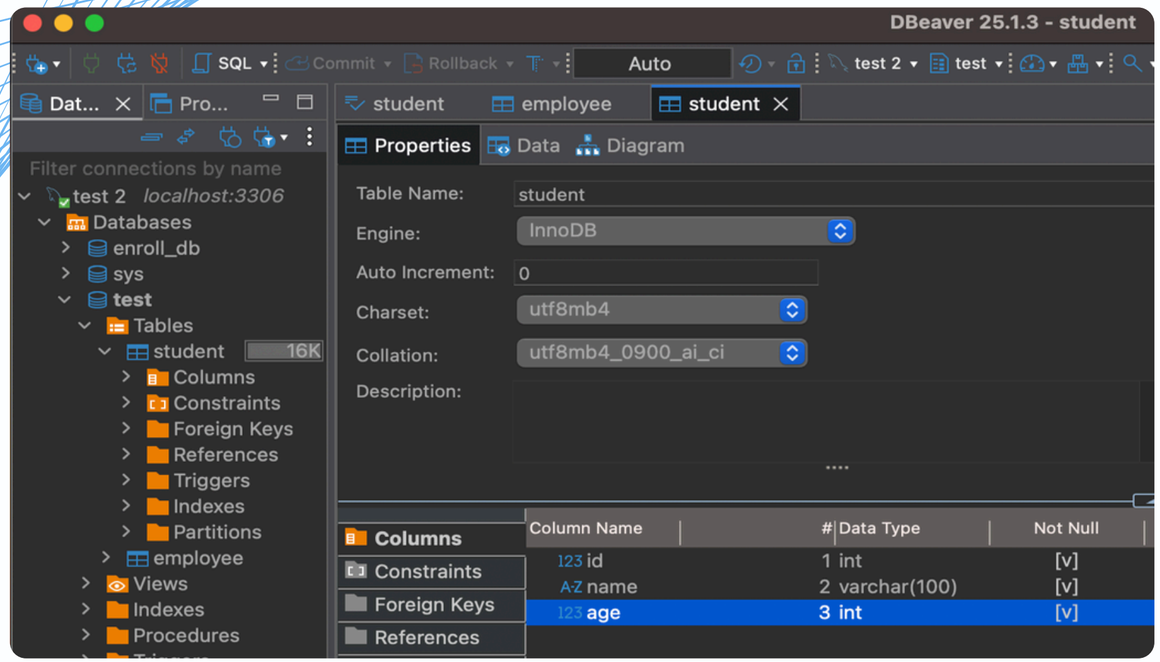
Rajeev Khoodeeram

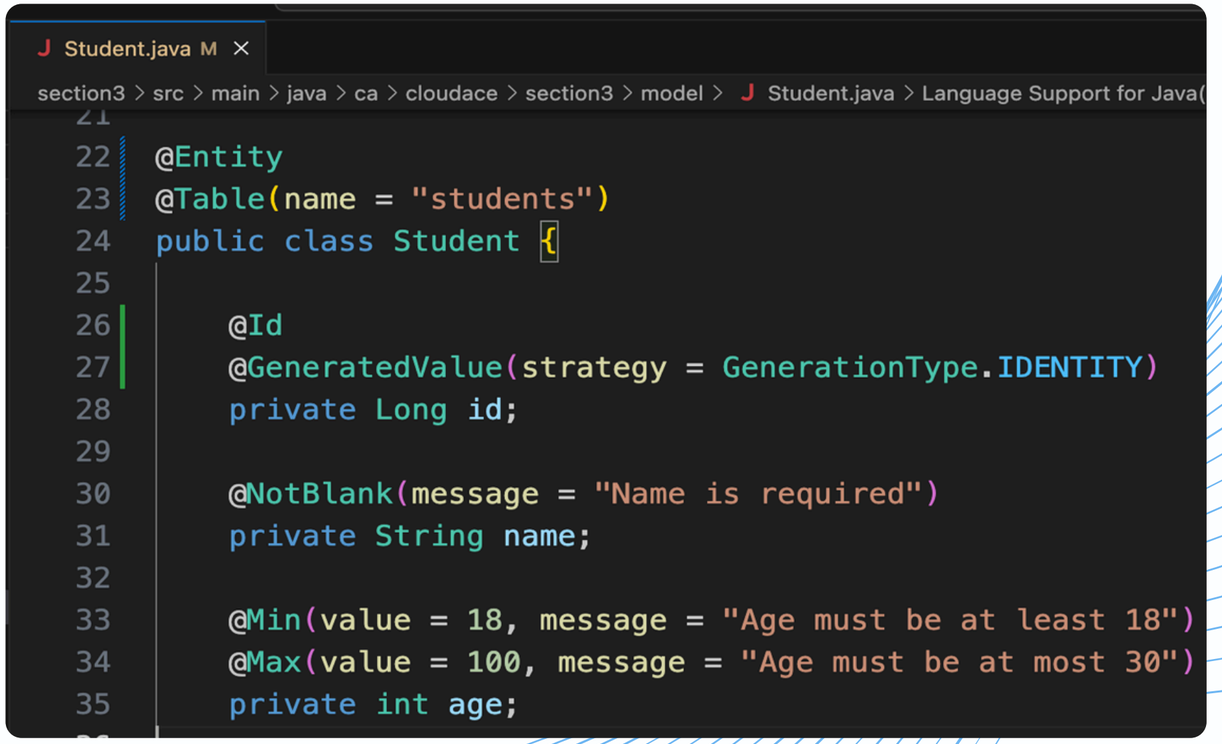
OCTOBER 2025

JPA ENTITY

- In JPA, an entity is a lightweight persistent domain object.
- It represents a table in your database, and each instance of the entity represents a row in that table.
- Entity mapping involves defining how your Java classes map to database tables and their fields map to column
  -

JAVA - MYSQL


DBeaver 25.1.3 - student
SQL
Commit
Rollback
Auto
test 2
test
Dat...
Pro...
Filter connections by name
test 2 localhost:3306
Databases
enroll_db
sys
test
Tables
student 16K
Columns
Constraints
Foreign Keys
References
Triggers
Indexes
Partitions
employee
Views
Indexes
Procedures
student
employee
student
Properties
Data
Diagram
Table Name: student
Engine: InnoDB
Auto Increment: 0
Charset: utf8mb4
Collation: utf8mb4_0900_ai_ci
Description:
Columns
Constraints
Foreign Keys
References
Column Name
Data Type
Not Null
123 id 1 int [v]
A-Z name 2 varchar(100) [v]
123 age 3 int [v]


Student.java M
section3 > src > main > java > ca > cloudace > section3 > model > Student.java > Language Support for Java(
@Entity
@Table(name = "students")
public class Student {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
@NotBlank(message = "Name is required")
private String name;
@Min(value = 18, message = "Age must be at least 18")
@Max(value = 100, message = "Age must be at most 30")
private int age;

ANNOTATIONS (1)

- @Entity: Declares the class as an entity bean.
  - This is the primary annotation for marking a class as a JPA entity.

- @Table(name = "students")
  - Specifies the primary table for the annotated entity.
  - If omitted, the table name defaults to the entity class name (Student).
  - It's good practice to explicitly define table names, especially if they differ from your class names or you want to use a specific naming convention (e.g., plural names).

ANNOTATIONS (2)

- **@Id**: Specifies the primary key of the entity.
  - Every entity must have a primary key.
- **@GeneratedValue(strategy = GenerationType.IDENTITY)**
  - Configures the strategy for primary key generation.
- **GenerationType.IDENTITY**
  - Relies on an auto-incremented column in the database (e.g., AUTO_INCREMENT in MySQL). This is often the most straightforward strategy for MySQL.
- **@Column**
  - Specifies the mapped column for a persistent property or field.
  - name: The name of the column in the database (defaults to field name if omitted).
  - nullable: true (default) if the column can contain NULL values, false if not.

FULL STACK DEV

Configure Spring Data JPA Repository Interface

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

THE SCENARIO

- StudentController → *controller package*
  - Handles HTTP requests (CRUD)
- StudentService → *service package*
  - Interface between controller and database (via repository)
- Student Object or table → *model package*
- StudentRepository → *repository package*
  -

JPAREPOSITORY

- Create a new package: **repository** (in main)

- For this, we will create a new folder or package inside the project called repository.
  - >>sudo mkdir repository
  - >>sudo chmod -R 777 repository

- Create **StudentRepository** interface which extends JpaRepository

  -

STUDENTREPOSITORY

- import com.example.demo.model.Product;
- import org.springframework.data.jpa.repository.JpaRepository;
- import org.springframework.stereotype.Repository;

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long>
{
// JpaRepository provides methods like:
// save(), findById(), findAll(), deleteById(), count(), etc.
// No implementation needed! Spring Data JPA provides it at runtime.
}
```

So it is blank but you can implement you own method as well

ANNOTATIONS

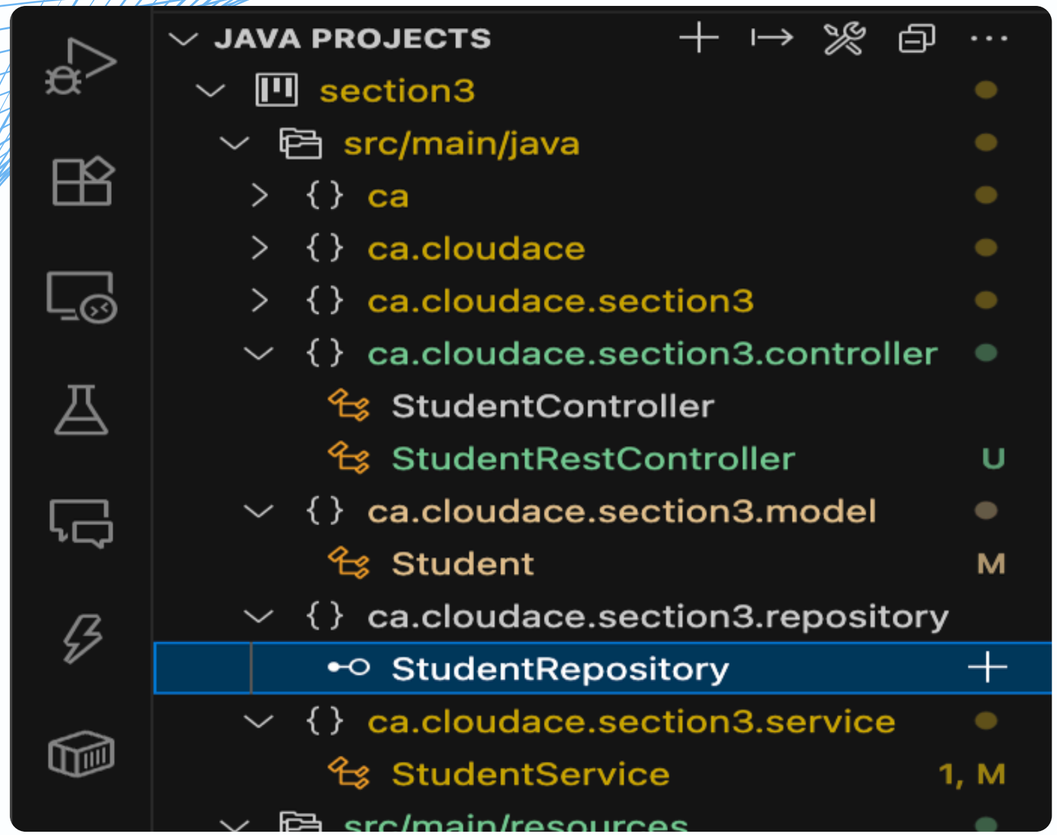
- **@Repository**
  - A stereotype annotation that indicates the class (or interface, in this case) is a repository and ensures that it's picked up by component scanning.
  - It also enables Spring's exception translation for persistence-related exceptions.

- **extends JpaRepository**<Student, Long>: This is the key.
  - **Student**: The entity type that this repository will manage.
  - **Long**: The data type of the entity's primary key (id in our Product entity).

PREDEFINED METHODS

- **By simply extending JpaRepository, you automatically get methods like:**
  - save(entity): Saves a given entity.
  - findById(id): Retrieves an entity by its ID. Returns Optional<T>.
  - findAll(): Returns all instances of the type.
  - deleteById(id): Deletes the entity with the given ID.
  - count(): Returns the number of entities available.
  - existsById(id): Returns whether an entity with the given ID exists.

FULL EXAMPLE



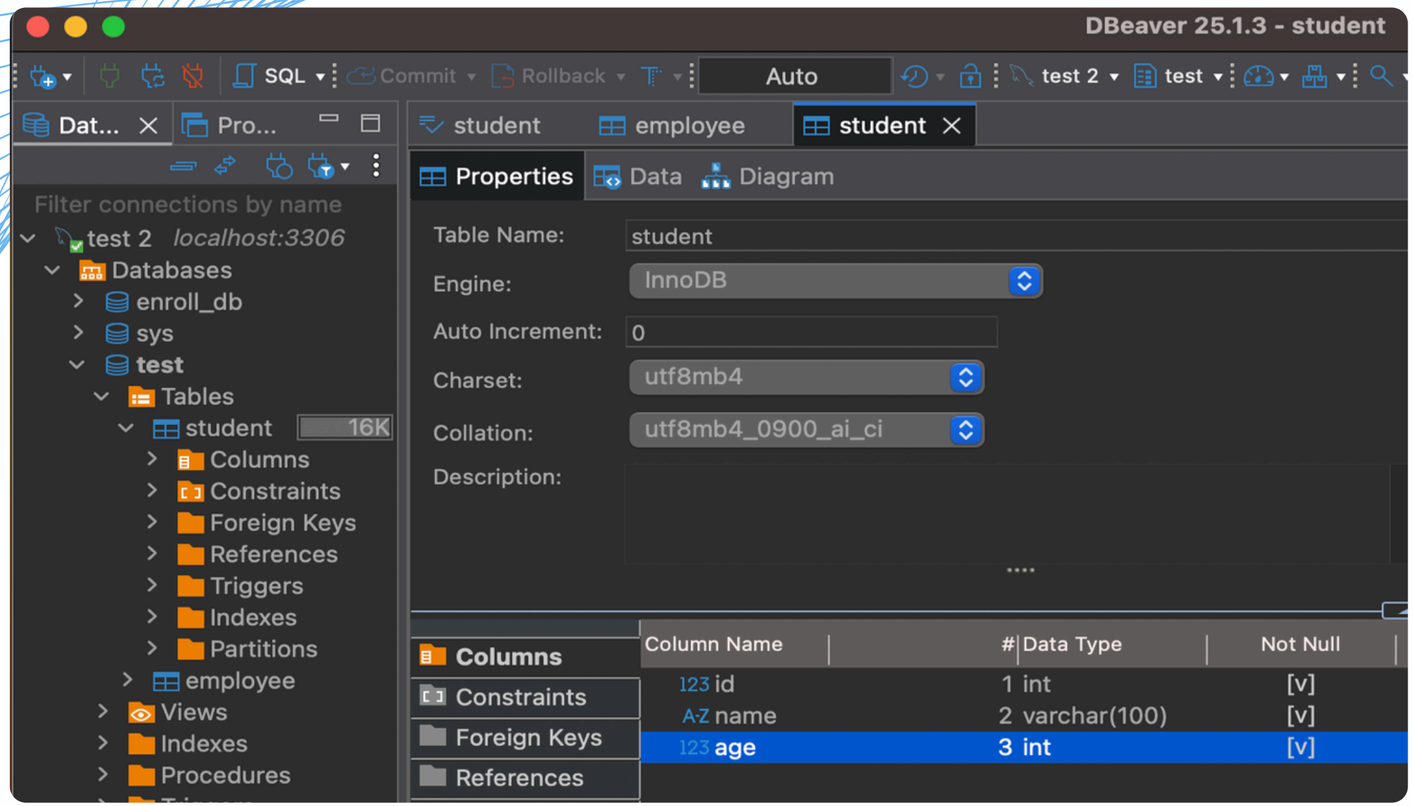

```java
public class StudentRestController {

    // Injecting the StudentService to handle business logic
    // This is a good practice to separate concerns and keep the cor
    // If you are using a service layer, you can inject it here to h
    @Autowired
    private final StudentService studentService;


    public StudentRestController(StudentService studentService) {
        this.studentService = studentService;

    }
```

PREDEFINED METHODS

```java
@Service
public class StudentService {

    @Autowired
    private StudentRepository studentRepository;

    public StudentService(StudentRepository studentRepository) {
        this.studentRepository = studentRepository;
    }

    /**
     * Retrieves all students from the database.
     *
     * @return a list of all students
     */
    public List<Student> getAllStudents() {
        return studentRepository.findAll();
    }
```

```java
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {
    // Additional query methods can be defined here if needed

}
```

MYSQL CONNECTION




```
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=rajeev
spring.datasource.password=Rk2025.;
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA/Hibernate settings
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```