Spring ORM Example using Hibernate

Last Updated : 23 Jul, 2025

Spring ORM is a module of the Java Spring framework used to implement the ORM(Object Relational Mapping) technique. It can be integrated with various mapping and persistence frameworks like **Hibernate, Oracle Toplink, iBatis**, etc. for database access and manipulation. This article covers an example of the integration of the Spring ORM module with Hibernate framework.

Prerequisites:

- Java Programming
- Spring Framework (Core, Context, and JDBC modules)
- Hibernate or any other ORM tool
- Maven for dependency management
- A relational database like MYSQL, PostgreSQL or H2

Key Components of Spring ORM

Spring ORM provides various classes and interfaces for integrating Spring applications with the Hibernate framework. Some useful classes in Spring ORM are:

- **LocalSessionFactoryBean:** Configures Hibernate's SessionFactory.
- **HibernateTransactionManager:** Manages transactions for Hibernate.
- **SessionFactory:** Used to create Hibernate Session instances for database operations.

Note: HibernateTemplate, which was widely used in older versions, is now deprecated. Modern applications use **SessionFactory** directly with

@Transactional annotations.

Step-by-Step Implementation

Step 1: Add Dependencies

We will use Maven for dependency management. We need to add the following dependencies to the pom.xml file.

Key Dependencies with Versions:

Dependency	Version
Spring ORM (spring-orm)	6.1.0
Spring Context (spring-context)	6.1.0
Spring JDBC (spring-jdbc)	6.1.0
Hibernate Core (hibernate-core)	6.4.0
MySQL Connector/J (mysql-connector-j)	8.1.0
Jakarta Persistence API (jakarta.persistence-api)	3.1.0
Spring Transaction Management (spring-tx)	6.1.0
JUnit 5 (junit-jupiter-api, junit-jupiter-engine)	5.10.0

pom.xml:

```xml
<project xmlns="https://maven.apache.org/POM/4.0.0"
         xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>spring-orm-hibernate-example</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <java.version>17</java.version>
    </properties>

    <dependencies>
        <!-- Spring ORM -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>6.1.0</version>
        </dependency>

        <!-- Spring Context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>6.1.0</version>
        </dependency>

        <!-- Spring JDBC -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>6.1.0</version>
        </dependency>

        <!-- Hibernate Core -->
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>6.4.0</version>
        </dependency>

        <!-- MySQL Connector/J -->
        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <version>8.1.0</version>
        </dependency>

        <!-- Jakarta Persistence API (JPA) -->
        <dependency>
            <groupId>jakarta.persistence</groupId>
```

```
            <artifactId>jakarta.persistence-api</artifactId>
            <version>3.1.0</version>
        </dependency>

        <!-- Spring Transaction Management -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>6.1.0</version>
        </dependency>

        <!-- Testing Dependencies -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.10.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.10.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- Compiler Plugin for Java 17 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.11.0</version>
                <configuration>
                    <source>${java.version}</source>
                    <target>${java.version}</target>
                </configuration>
            </plugin>

            <!-- Maven Surefire Plugin for Testing -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.2.0</version>
                <configuration>
                    <forkCount>1</forkCount>
                    <reuseForks>false</reuseForks>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Note:

- *The project is configured for **Java 17 version**. if you are using a different version, update the **<java.version> property.***
- *If you are using **Jakarta EE 9 or latest,** make sure all javax.persistence imports are replaced with **jakarta.persistence.***
- *The **maven-surefire-plugin** is configured to run Junit 5 tests.*

Step 2: Create Entity Class

Now, we are going to define an entity class that maps to a database table. For example let's create a Student entity.

Student.java:

```
package com.example.model;

import jakarta.persistence.*;

@Entity
@Table(name = "students")
public class Student {

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

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

    @Column(name = "email", unique = true)
    private String email;

    // Getters and Setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

Note: If you are using Jakarta EE 9 or later. Use **jakarta.persistence** instead of **javax.persistence.**

Step 3: Configure Hibernate and Spring

Use Java-based configuration to set up Hibernate and Spring. Create a configuration class.

```
package com.example.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
import org.springframework.orm.hibernate5.HibernateTransactionManager;
import
org.springframework.transaction.annotation.EnableTransactionManagement;
import javax.sql.DataSource;
import java.util.Properties;

@Configuration
@EnableTransactionManagement
public class HibernateConfig {

    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactory = new
LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource());
        sessionFactory.setPackagesToScan("com.example.model");
        sessionFactory.setHibernateProperties(hibernateProperties());
        return sessionFactory;
    }

    @Bean
    public DataSource dataSource() {
        org.springframework.jdbc.datasource.DriverManagerDataSource
dataSource = new
```

```
org.springframework.jdbc.datasource.DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");
        dataSource.setUsername("root");
        dataSource.setPassword("password");
        return dataSource;
    }

    private Properties hibernateProperties() {
        Properties properties = new Properties();
        properties.put("hibernate.dialect",
"org.hibernate.dialect.MySQL8Dialect");
        properties.put("hibernate.show_sql", "true");
        properties.put("hibernate.hbm2ddl.auto", "update");
        return properties;
    }

    @Bean
    public HibernateTransactionManager transactionManager() {
        HibernateTransactionManager txManager = new
HibernateTransactionManager();
        txManager.setSessionFactory(sessionFactory().getObject());
        return txManager;
    }
}
```

Step 4: Create DAO Interface and Implementation

Define a DAO (Data Access Object) interface and its implementation

DAO Interface:

```
package com.example.dao;

import com.example.model.Student;
import java.util.List;

public interface StudentDao {
    void save(Student student);
    Student findById(Long id);
    List<Student> findAll();
    void update(Student student);
    void delete(Long id);
}
```

DAO Implementation:

```java
package com.example.dao;

import com.example.model.Student;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Repository
@Transactional
public class StudentDaoImpl implements StudentDao {

    @Autowired
    private SessionFactory sessionFactory;

    @Override
    public void save(Student student) {
        Session session = sessionFactory.getCurrentSession();
        session.save(student);
    }

    @Override
    public Student findById(Long id) {
        Session session = sessionFactory.getCurrentSession();
        return session.get(Student.class, id);
    }

    @Override
    public List<Student> findAll() {
        Session session = sessionFactory.getCurrentSession();
        return session.createQuery("FROM Student",
Student.class).getResultList();
    }

    @Override
    public void update(Student student) {
        Session session = sessionFactory.getCurrentSession();
        session.update(student);
    }

    @Override
    public void delete(Long id) {
        Session session = sessionFactory.getCurrentSession();
        Student student = session.get(Student.class, id);
        if (student != null) {
            session.delete(student);
        }
    }
}
```

Step 5: Test the Application

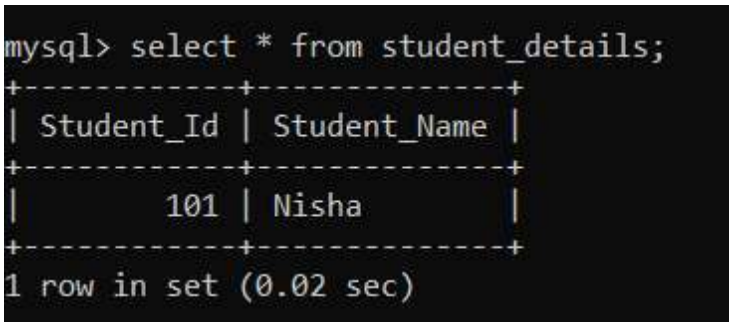
Create a main class to test the CRUD operations

```
package com.example;

import com.example.config.HibernateConfig;
import com.example.dao.StudentDao;
import com.example.model.Student;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

import java.util.List;

public class MainApp {
    public static void main(String[] args) {
        // Initialize Spring context
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(HibernateConfig.class);
        StudentDao studentDao = context.getBean(StudentDao.class);

        // Insert a new student with ID 101 and name "Nisha"
        Student student1 = new Student();
         // Manually set ID for demonstration
        student1.setId(101L);
        student1.setName("Nisha");
        studentDao.save(student1);
        System.out.println("Student inserted: " + student1);

        // Update the student's name to "Priya"
        student1.setName("Priya");
        studentDao.update(student1);
        System.out.println("Student updated: " + student1);

        // Retrieve the student with ID 101
        Student retrievedStudent = studentDao.findById(101L);
        System.out.println("Retrieved Student: " + retrievedStudent);

        // Insert additional students for demonstration
        Student student2 = new Student();
         // Manually set ID for demonstration
        student2.setId(102L);
        student2.setName("Danish");
        studentDao.save(student2);

        Student student3 = new Student();
        // Manually set ID for demonstration
        student3.setId(103L);
        student3.setName("Sneha");
        studentDao.save(student3);

        // Retrieve all students before deletion
        System.out.println("All Students Before Deletion:");
        List<Student> studentsBeforeDeletion = studentDao.findAll();
```

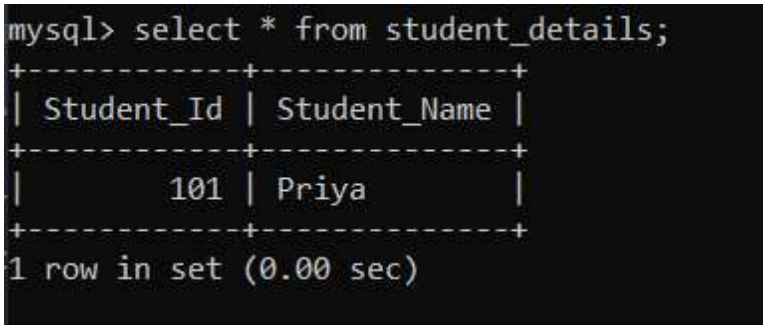
```
        studentsBeforeDeletion.forEach(System.out::println);

        // Delete the student with ID 102
        studentDao.delete(102L);
        System.out.println("Student with ID 102 deleted.");

        // Retrieve all students after deletion
        System.out.println("All Students After Deletion:");
        List<Student> studentsAfterDeletion = studentDao.findAll();
        studentsAfterDeletion.forEach(System.out::println);

        // Close the Spring context
        context.close();
    }
}
```

Output:

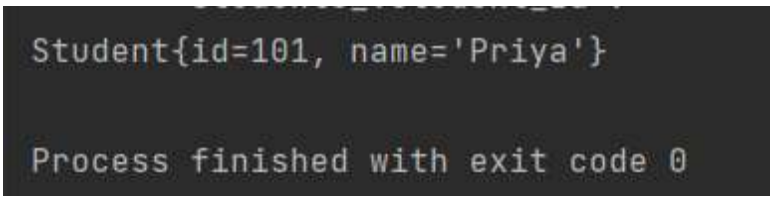
Insertion:



Updation:



Retrieval (Get):



Retrieval (Get All) After Deletion:

```
Student{id=101, name='Priya'}
Student{id=102, name='Danish'}
Student{id=103, name='Sneha'}

Process finished with exit code 0
```

Deletion:

```
mysql> select * from student_details;
+------------+--------------+
| Student_Id | Student_Name |
+------------+--------------+
|        101 | Priya        |
|        103 | Sneha        |
+------------+--------------+
2 rows in set (0.00 sec)
```

Comment More info Advertise with us

GeeksforGeeks
Sanchhaya Education Private Limited

Corporate & Communications Address:

A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305)

Registered Address:

K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305