# SPRING ORM

**Spring ORM (Object Relational Mapping)** is a module in the Spring Framework that helps you integrate Spring applications with ORM tools like **Hibernate**, **JPA**, or **JDO**. It makes database operations easier by managing things like transactions, session handling, and exception translation, so you can focus more on writing business logic instead of boilerplate database code.

## Getting started

1. Create a maven project
2. And add following dependencies in pom.xml
    a. spring-core
    b. spring-context
    c. mysql-connector-java
    d. spring-orm

## Spring ORM project

1. Create a spring configuration file like following

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:p="http://www.springframework.org/schema/p"
        xmlns:c="http://www.springframework.org/schema/c"
        xmlns:util="http://www.springframework.org/schema/util"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context.xsd
 http://www.springframework.org/schema/util
 http://www.springframework.org/schema/util/spring-util.xsd">




 </beans>
```

2. Create a package for entity and add Student entity like following

```
package com.spring.orm.entity;

@Entity
@Table(name = "student_info")
public class Student {
	@Id
	@Column(name = "student_id")
	private int id;
	@Column(name = "student_id")
	private String name;
	@Column(name = "student_laguage")
	private String laguage;
```

Create getter, setter, parameterized and non-parameterized constructor.

3. Create a dao package and add Student dao like following

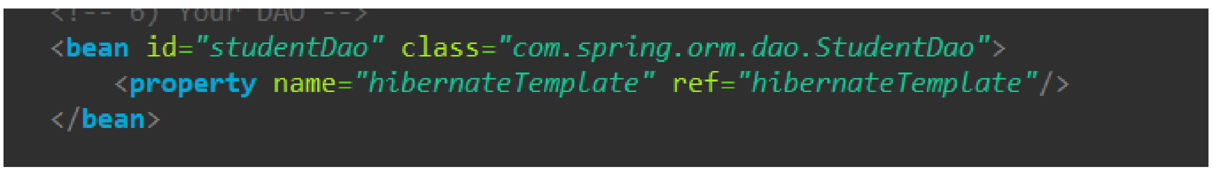
```
package com.spring.orm.dao;

public class StudentDao {
	private HibernateTemplate hibernateTemplate;

	public int insert(Student student) {
		Integer rs = (Integer) hibernateTemplate.save(student);
		return rs;
	}
}
```

## Configuring HibernateTemplate in Spring

Now you must be Wondering how we will get the object of HibernateTemplate in StudentDao, for that we have to do following configuration

**1. Expose HibernateTemplate to your DAO**

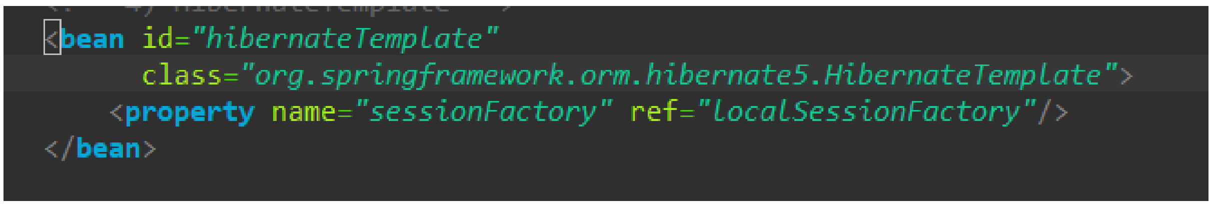
- In your `StudentDao` class, you'll have a property of type `HibernateTemplate`.
- To wire it up, define a `StudentDao` bean in `spring-config.xml` and set its `hibernateTemplate` property by reference.



```xml
<bean id="studentDao" class="com.spring.orm.dao.StudentDao">
    <property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>
```

**2. Define the HibernateTemplate bean**

- `HibernateTemplate` itself needs a `SessionFactory`.
- Declare a `<bean>` of type `org.springframework.orm.hibernate5.HibernateTemplate` and pass in the `SessionFactory` bean:



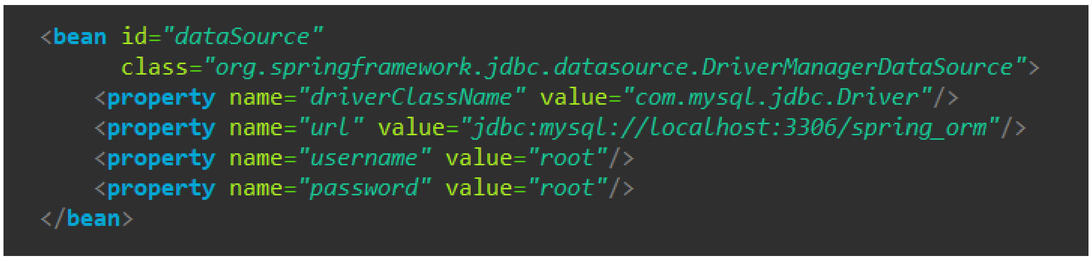
```xml
<bean id="hibernateTemplate"
      class="org.springframework.orm.hibernate5.HibernateTemplate">
    <property name="sessionFactory" ref="localSessionFactory"/>
</bean>
```

**3. Create the SessionFactory**

- Since `SessionFactory` is an interface, use Spring's `LocalSessionFactoryBean` implementation.
- This bean has three key properties:
    - **`dataSource`** – the JDBC source for connections
    - **`hibernateProperties`** – a `java.util.Properties` set containing things like dialect, DDL mode, show-SQL flag, etc
    - **`annotatedClasses`** – a list of your `@Entity` classes

A. DataSource

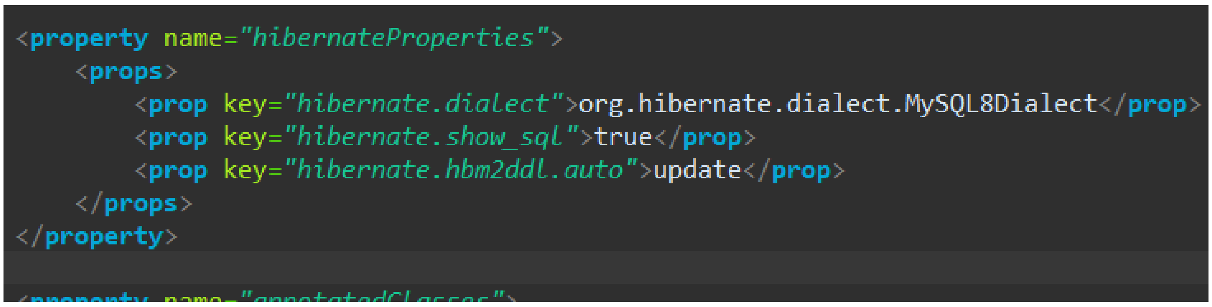
- You'll use Spring's `DriverManagerDataSource` (or Hikari, etc.)(implementation of **DataSource Interface**) to supply JDBC URL, driver class, username/password etc...



```xml
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/spring_orm"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>
```

B. Hibernate Properties

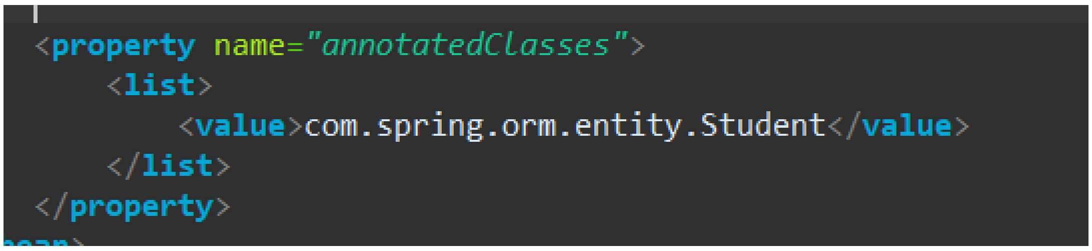
- In the `LocalSessionFactoryBean` definition, nest a `<props>` block with keys such as:



```xml
<property name="hibernateProperties">
    <props>
        <prop key="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</prop>
        <prop key="hibernate.show_sql">true</prop>
        <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
</property>

<property name="annotatedClasses">
```

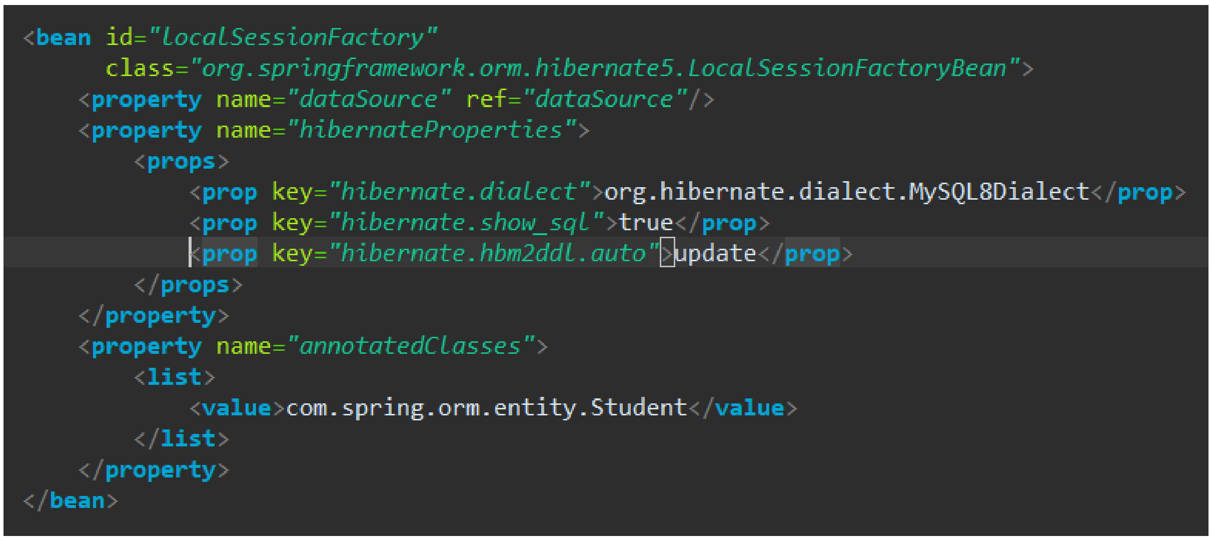
C. Annotated Classes

- Provide a `<list>` of fully qualified `@Entity` class names so Hibernate knows what to map:



```xml
<property name="annotatedClasses">
    <list>
        <value>com.spring.orm.entity.Student</value>
    </list>
</property>
bean>
```

Now, in conclusion the entire LocalSessionFactory  bean will be like following



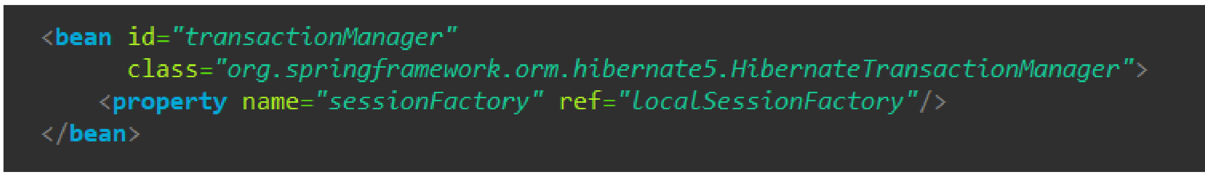
```xml
<bean id="localSessionFactory"
      class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
        </props>
    </property>
    <property name="annotatedClasses">
        <list>
            <value>com.spring.orm.entity.Student</value>
        </list>
    </property>
</bean>
```

**4.  By now, our configuration is enough only for reading data from the database.**

But if we want to perform **DML operations** (like insert, update, delete) or handle **transactions**, we need to do a bit more setup:

1. **Create a Transaction Manager**

    - Define a bean of `HibernateTransactionManager` in your `spring-config.xml`.
    - Set its `sessionFactory` property to the same one used by `HibernateTemplate`.

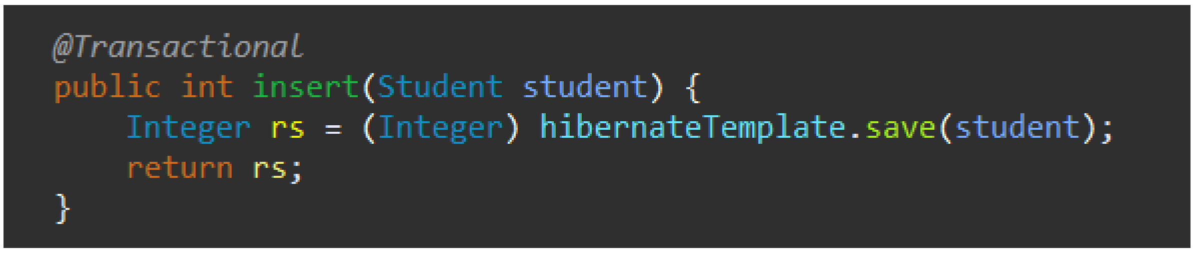


```xml
<bean id="transactionManager"
      class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="localSessionFactory"/>
</bean>
```

2. **Enable Transaction Support in Spring**
    - Add this namespace to your XML root:
      `xmlns:tx="http://www.springframework.org/schema/tx"`
    - And add this schema location:
      `http://www.springframework.org/schema/tx`
      `https://www.springframework.org/schema/tx/spring-tx.xsd`
    - Inside the `<beans>` tag, add:
      `<tx:annotation-driven />`

3. **Use `@Transactional` Annotation**

   - Now, just annotate your DAO or service methods (the ones doing DML) with `@Transactional`.
   - Spring will automatically manage transactions — it will commit or roll back as needed.



```java
@Transactional
public int insert(Student student) {
    Integer rs = (Integer) hibernateTemplate.save(student);
    return rs;
}
```

**Following is the complete configuration file**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:util="http://www.springframework.org/schema/util"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="
     http://www.springframework.org/schema/beans
      https://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/context
      https://www.springframework.org/schema/context/spring-context.xsd
     http://www.springframework.org/schema/util
      https://www.springframework.org/schema/util/spring-util.xsd
     http://www.springframework.org/schema/tx
      https://www.springframework.org/schema/tx/spring-tx.xsd">
 <tx:annotation-driven/>
 <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
   <property name="url" value="jdbc:mysql://localhost:3306/spring_orm"/>
   <property name="username" value="root"/>
   <property name="password" value="root"/>
 </bean>
 <bean id="localSessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
   <property name="dataSource" ref="dataSource"/>
   <property name="hibernateProperties">
     <props>
       <prop key="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</prop>
       <prop key="hibernate.show_sql">true</prop>
       <prop key="hibernate.hbm2ddl.auto">update</prop>
     </props>
   </property>
   <property name="annotatedClasses">
     <list>
       <value>com.spring.orm.entity.Student</value>
     </list>
   </property>
 </bean>
 <bean id="hibernateTemplate"
    class="org.springframework.orm.hibernate5.HibernateTemplate">
   <property name="sessionFactory" ref="localSessionFactory"/>
 </bean>
 <bean id="transactionManager"
    class="org.springframework.orm.hibernate5.HibernateTransactionManager">
   <property name="sessionFactory" ref="localSessionFactory"/>
 </bean>
 <bean id="studentDao" class="com.spring.orm.dao.StudentDao">
   <property name="hibernateTemplate" ref="hibernateTemplate"/>
 </bean>      </beans>
```

## Crud Operation Using Spring ORM

```java
public class StudentDao {
	private HibernateTemplate hibernateTemplate;

	public HibernateTemplate getHibernateTemplate() {
		return hibernateTemplate;
	}
	public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
		this.hibernateTemplate = hibernateTemplate;
	}
	// Create
	@Transactional
	public int insert(Student student) {
		Integer rs = (Integer) this.hibernateTemplate.save(student);
		return rs;
	}

	// Read
	//Read single row
	public Student getStudent(int studentId) {
		Student student = this.hibernateTemplate.get(Student.class, studentId);
		return student;
	}

	//Read All rows
	public List<Student> getAllStudents() {
		List<Student> all = this.hibernateTemplate.loadAll(Student.class);
		return all;
	}

	// Delete
	@Transactional
	public void delete(int studentId) {
		Student student = this.hibernateTemplate.get(Student.class, studentId);
		this.hibernateTemplate.delete(student);
	}

	//Update
	@Transactional
	public void update(Student student) {
		this.hibernateTemplate.update(student);
	}
}
```

Create

```
// Create

    Student student1 = new Student(1, "Tony", "Java");
    Student student2 = new Student(2, "Peter", "Python");
    Student student3 = new Student(3, "Durgesh", "React");
    studentDao.insert(student1);
    studentDao.insert(student2);
    studentDao.insert(student3);
```

Read

```
// Read

    Student student = studentDao.getStudent(1);
    System.out.println(student);
    System.out.println("-------------------------------------------------------------");
    List<Student> allStudents = studentDao.getAllStudents();
    for(Student stnt : allStudents) {
        System.out.println(stnt);
    }
```

Update

```
// update

    Student uStudent1 = new Student(1, "Iron Man", "All");
    studentDao.update(uStudent1);
```

Delete

```
// Delete

    studentDao.delete(3);
```

## Java-Based Configuration

```
@Configuration
@EnableTransactionManagement
public class OrmConfiguration {
	@Bean(name = "studentDao")
	public StudentDao getStudentDao() {
		StudentDao studentDao = new StudentDao();
		studentDao.setHibernateTemplate(getHibernateTemplate());
		return studentDao;
	}
	@Bean(name = "hibernateTemplate")
	public HibernateTemplate getHibernateTemplate() {
		HibernateTemplate hibernateTemplate = new HibernateTemplate();
		hibernateTemplate.setSessionFactory(getSessionFactory().getObject());

		return hibernateTemplate;
	}
	@Bean(name = "localSessionFactory")
	public LocalSessionFactoryBean getSessionFactory() {
		LocalSessionFactoryBean factoryBean = new LocalSessionFactoryBean();
		factoryBean.setDataSource(getDataSource());

		Properties hibernateroperties = new Properties();
		hibernateroperties.setProperty("hibernate.dialect",
	"org.hibernate.dialect.MySQL8Dialect");
		hibernateroperties.setProperty("hibernate.show_sql", "true");
		hibernateroperties.setProperty("hibernate.hbm2ddl.auto", "update");
		factoryBean.setHibernateProperties(hibernateroperties);

		factoryBean.setAnnotatedClasses(Student.class);

		return factoryBean;
	}
	@Bean(name = "dataSource")
	public DataSource getDataSource() {
		DriverManagerDataSource ds = new DriverManagerDataSource();
		ds.setDriverClassName("com.mysql.jdbc.Driver");
		ds.setUrl("jdbc:mysql://localhost:3306/spring_orm");
		ds.setUsername("root");
		ds.setPassword("root");
		return ds;
	}
	@Bean
	public HibernateTransactionManager getTransactionManager() {
		HibernateTransactionManager transactionManager = new
HibernateTransactionManager();
		transactionManager.setSessionFactory(getSessionFactory().getObject());
		return transactionManager;
	}
}
```

# Java-Based Configuration for Spring ORM with Hibernate

In this configuration, we use **Java-based configuration** (`@Configuration`) instead of `spring-config.xml` to set up Spring + Hibernate integration.

We also use **`@EnableTransactionManagement`** to enable Spring's annotation-driven transaction management using `@Transactional`.

---

### `@Configuration` and `@EnableTransactionManagement`

- `@Configuration`: Marks the class as a Spring configuration class.
- `@EnableTransactionManagement`: Enables annotation-driven transaction support (@Transactional).

---

### Defining the `DataSource` Bean

- We use `DriverManagerDataSource`, which is a simple implementation of `DataSource` for testing or small apps.
- Replace the URL, username, and password based on your DB setup.

---

### Configuring `LocalSessionFactoryBean`

- This bean creates and configures the Hibernate `SessionFactory`.
- We set 3 key things:
    - `DataSource` – for DB connection
    - `hibernateProperties` – to configure Hibernate behavior
    - `annotatedClasses` – all entity classes with `@Entity` annotation

If you have more than one entity, pass them like:

```
factoryBean.setAnnotatedClasses(Student.class, Course.class,
Teacher.class);
```

---

### Defining the `HibernateTemplate` Bean

- `HibernateTemplate` simplifies common Hibernate operations like save, update, delete, and load.
- It requires a `SessionFactory`, which we get using `getSessionFactory().getObject()` because:
    - `LocalSessionFactoryBean` is a **FactoryBean**, so `getObject()` gives the actual `SessionFactory`.

---

### Defining the DAO Bean (`StudentDao`)

- We create and return the DAO object and inject `HibernateTemplate` into it.

---

### Enabling Transactions with `HibernateTransactionManager`

- `HibernateTransactionManager` manages transaction boundaries.
- You must annotate transactional methods with `@Transactional` (works because of `@EnableTransactionManagement`).