Spring ORM

Spring ORM (Object-Relational Mapping) is a module in the Spring Framework that provides integration layers for popular ORM frameworks like Hibernate, JPA (Java Persistence API), and others. It simplifies the development of data access layers by managing the complexities of ORM frameworks and providing a consistent way to interact with databases.
Key features of Spring ORM include:

- Transaction Management: Simplifies transaction management by integrating with Spring's transaction management capabilities.
- Declarative Transactions: Allows the use of annotations or XML configuration to manage transactions declaratively.
- Integration with ORM Frameworks: Provides support for Hibernate, JPA, and other ORM frameworks, making it easier to switch between them if needed.
- Session Management: Manages the lifecycle of ORM sessions, reducing boilerplate code.

Spring JDBC and Spring ORM serve different purposes and have distinct features:

Spring JDBC:

- Direct Database Access: Provides a straightforward way to interact with databases using SQL queries.
- Low-Level API: Requires developers to write SQL queries and handle ResultSets, making it more hands-on and closer to the database.
- Less Abstraction: Offers less abstraction compared to ORM frameworks, giving developers more control over SQL execution.
- Manual Mapping: Developers need to manually map database rows to Java objects.

Spring ORM:

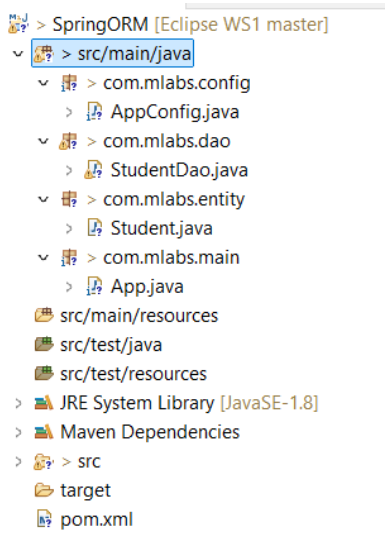
- Object-Relational Mapping: Uses ORM frameworks to map Java objects to database tables, reducing the need for manual SQL queries.
- High-Level API: Provides a higher level of abstraction, allowing developers to work with Java objects rather than SQL queries.
- Automatic Mapping: Automatically maps database rows to Java objects based on annotations or XML configuration.
- Transaction Management: Integrates seamlessly with Spring's transaction management, making it easier to manage transactions declaratively.

Summary

- Spring JDBC is ideal for applications that require direct and fine-grained control over SQL queries and database interactions.
- Spring ORM is suitable for applications that benefit from the abstraction and convenience of ORM frameworks, reducing boilerplate code and simplifying data access layers.

Step by step : Spring ORM Project

1. Create a Maven project.
2. Update pom.xml with necessary dependencies.
3. Create a Configuration class (Datasource, SessionFactory & Transaction)
4. Create an Entity Student class.
5. Create DAO class (Repository).
6. Create a main class to test the setup.



Step 1: Create a Maven Project

1. Open your IDE (like IntelliJ IDEA or Eclipse).
2. Create a new Maven project.

Step 2: Update pom.xml

Add the necessary dependencies for Spring, Hibernate, and MySQL (or any other database you prefer).

```
<dependencies>
        <!-- Spring ORM -->
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-orm</artifactId>
                <version>6.0.11</version>
        </dependency>

        <!-- Spring Context -->
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-context</artifactId>
                <version>6.0.11</version>
        </dependency>

        <!-- Hibernate -->
        <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-core</artifactId>
                <version>6.3.1.Final</version>
        </dependency>

        <!-- MySQL Connector -->
        <dependency>
                <groupId>com.mysql</groupId>
                <artifactId>mysql-connector-j</artifactId>
                <version>8.0.33</version>
        </dependency>
```

```
		<!-- JPA API -->
		<dependency>
			<groupId>jakarta.persistence</groupId>
			<artifactId>jakarta.persistence-api</artifactId>
			<version>3.1.0</version>
		</dependency>

		<!-- Commons DBCP2 (for DataSource) -->
		<dependency>
			<groupId>org.apache.commons</groupId>
			<artifactId>commons-dbcp2</artifactId>
			<version>2.9.0</version>
		</dependency>
	</dependencies>
```

Step 3: Configure a Configuration class

```
package com.mlabs.config;

import java.util.Properties;

import javax.sql.DataSource;

import org.apache.commons.dbcp2.BasicDataSource;
import org.hibernate.SessionFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.hibernate5.HibernateTransactionManager;
import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;


@Configuration
@ComponentScan(basePackages = "com.mlabs")
@EnableTransactionManagement
public class AppConfig {

    @Bean
    public DataSource getDataSource() {
        BasicDataSource ds = new BasicDataSource();
        ds.setDriverClassName("com.mysql.cj.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
        ds.setUsername("root");
        ds.setPassword("root");
        return ds;
    }

    @Bean
```

```
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sf = new LocalSessionFactoryBean();
        sf.setDataSource(getDataSource());
        sf.setPackagesToScan("com.mlabs.entity");

        Properties props = new Properties();
        props.put("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");
        props.put("hibernate.show_sql", "true");
        props.put("hibernate.hbm2ddl.auto", "update");
        sf.setHibernateProperties(props);

        return sf;
    }

    @Bean
    public HibernateTransactionManager txManager(SessionFactory sessionFactory) {
        return new HibernateTransactionManager(sessionFactory);
    }
}
```

Note:

@EnableTransactionManagement
It enables Spring's annotation-driven transaction management capability.
It tells Spring to:

1. *Look for methods annotated with @Transactional*
2. *Automatically manage the begin, commit, and rollback of transactions based on the logic inside those methods.*

Step 4: Create the Student **Class**

```
package com.mlabs.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

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

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

    private String name;
```

```
    private String department;


        public int getId() {
                return id;
        }
        public void setId(int id) {
                this.id = id;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getDepartment() {
                return department;
        }
        public void setDepartment(String department) {
                this.department = department;
        }
        @Override
        public String toString() {
                return "Student [id=" + id + ", name=" + name + ", department=" + department +
"]";
        }

}
```

Step 5: Create DAO class (Repository

```
package com.mlabs.dao;

import java.util.List;

import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import com.mlabs.entity.Student;

@Repository
public class StudentDao {

    @Autowired
    private SessionFactory sessionFactory;

    @Transactional
```

```
    public void insert(Student student) {
        sessionFactory.getCurrentSession().save(student);
    }

    @Transactional(readOnly = true)
    public Student getById(int id) {
        return sessionFactory.getCurrentSession().get(Student.class, id);
    }

    @Transactional(readOnly = true)
    public List<Student> getAll() {
        return sessionFactory.getCurrentSession().createQuery("from Student",
Student.class).list();
    }

    @Transactional
    public void update(Student student) {
        sessionFactory.getCurrentSession().update(student);
    }

    @Transactional
    public void delete(int id) {
        Student s = sessionFactory.getCurrentSession().get(Student.class, id);
        if (s != null)
            sessionFactory.getCurrentSession().delete(s);
    }
}
```

Note:
@Transactional

- *Starts a transaction when the method begins*
- *Commits the transaction if the method completes successfully*
- *Rolls back the transaction if a runtime exception is thrown*

@Transactional(readOnly = true)
This tells spring
"This method is read-only — it only retrieves data and won't modify the database"

- *Skips dirty checking and flushing*
- *Prevents unintended database modifications*

Step 6: Create a main class

```
package com.mlabs.main;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
```

```
import com.mlabs.config.AppConfig;
import com.mlabs.dao.StudentDao;
import com.mlabs.entity.Student;

public class App {
	public static void main(String[] args) {
		AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

		StudentDao dao = context.getBean(StudentDao.class);

		Student s1 = new Student();
		s1.setName("John");
		s1.setDepartment("Maths");
		dao.insert(s1);

		Student s = dao.getById(1);
		System.out.println(s);
		s.setDepartment("Computers");
		dao.update(s);
		System.out.println(s);


//		dao.delete(1);

		context.close();
	}
}
```

Step 9: Set up your database.

Step 10: Run the application.