Spring JDBC

What is Spring JDBC?

Spring JDBC is a part of the Spring Framework that helps you interact with databases using JDBC (Java Database Connectivity) in a simpler and cleaner way. It reduces the amount of code you need to write while handling database operations.

Feature	**JDBC**	**Spring JDBC**
Boilerplate Code	We have to write a lot of code (connection, statement, result set handling).	Reduces code by handling common tasks automatically.
Exception Handling	Throws checked `SQLExceptions`. we must manually handle them.	Converts `SQLExceptions` to runtime exceptions (`DataAccessException`), making it easier to manage errors.
Transaction Management	We must write complex code to manage transactions.	Provides **declarative transactions**, reducing code complexity.
Resource Management	We have to manually close connections and resources.	Automatically manages and closes resources.
Code Readability	Messy and hard to read due to repetitive code.	Clean and easy to read. Focuses only on business logic.

JdbcTemplate:

Spring JdbcTemplate is a helper class in the Spring Framework that simplifies JDBC (Java Database Connectivity) operations. It reduces boilerplate code by handling tasks like resource management, SQL execution, and exception handling, making database interactions more efficient and less error-prone.

To use JdbcTemplate, we need to provide a DataSource object, which contains the database configuration details such as:
✔ Database URL
✔ Driver class name
✔ Username and Password

Spring internally manages the database connection, executes queries, and ensures proper closing of resources, making it a preferred approach for interacting with relational databases in Spring applications.

DataSource:

In Spring JDBC, DataSource is an interface that manages database connections. Instead of manually using DriverManager like in plain JDBC, Spring uses DataSource to simplify and optimize database access.

Since DataSource is an interface, we use its implementation classes like DriverManagerDataSource to create an object. It is configured with:
✔ Database URL
✔ Username & Password
✔ Driver Class Name

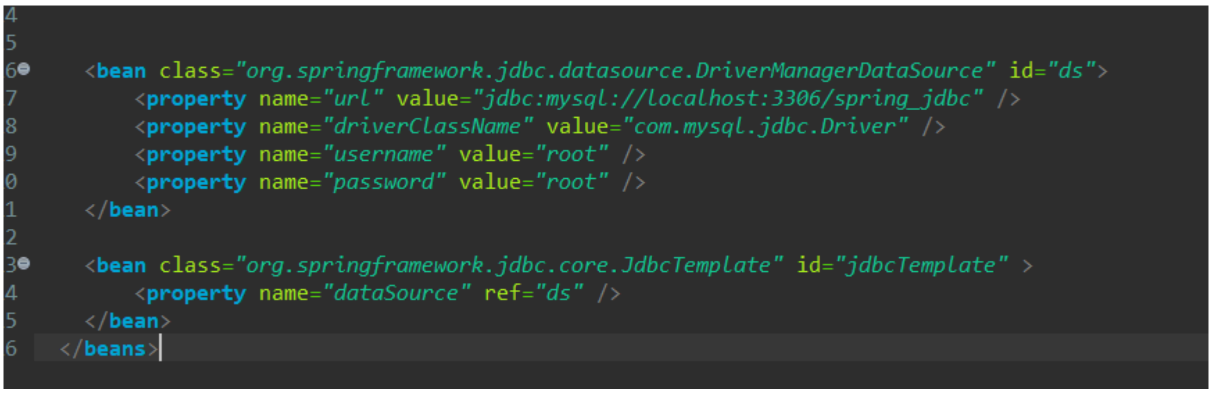
Additionally, we can use advanced DataSource implementations like HikariCP, Apache DBCP, or C3P0 to enable connection pooling, which significantly improves performance.

Following are the required maven dependencies for spring jdbc project :

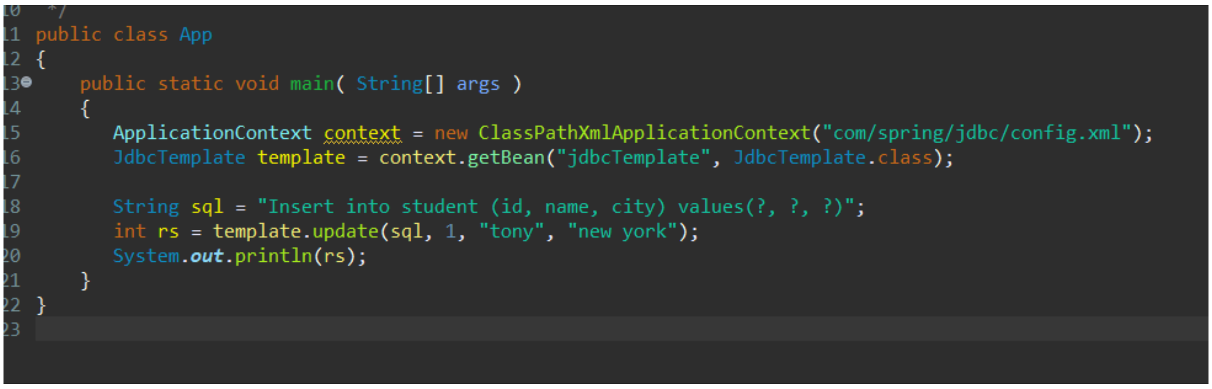
- Spring-core
- Spring-context
- Spring-jdbc
- Mysql-connector

XML Base Configuration

To interact with the database using Spring JdbcTemplate, we need an instance of JdbcTemplate. Since JdbcTemplate requires a DataSource object (which contains database configuration details), we first create a DriverManagerDataSource bean and then inject its reference into the JdbcTemplate bean.



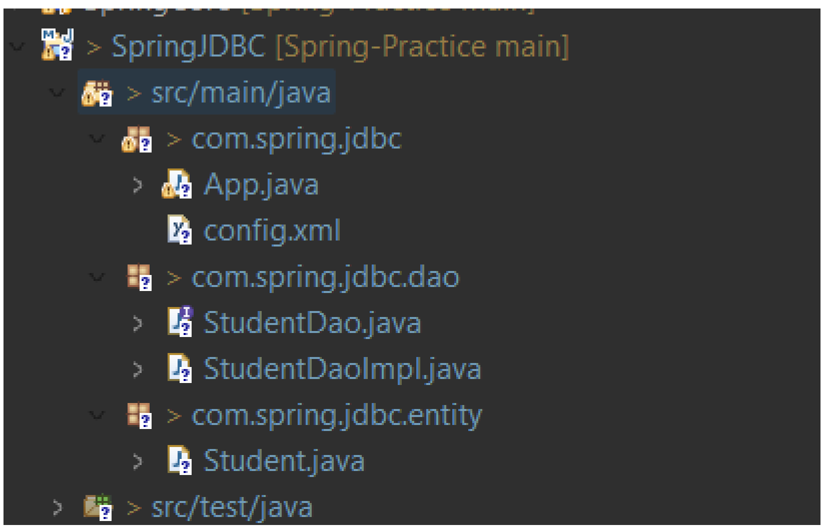
```


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

    <bean class="org.springframework.jdbc.core.JdbcTemplate" id="jdbcTemplate" >
        <property name="dataSource" ref="ds" />
    </bean>
  </beans>
```

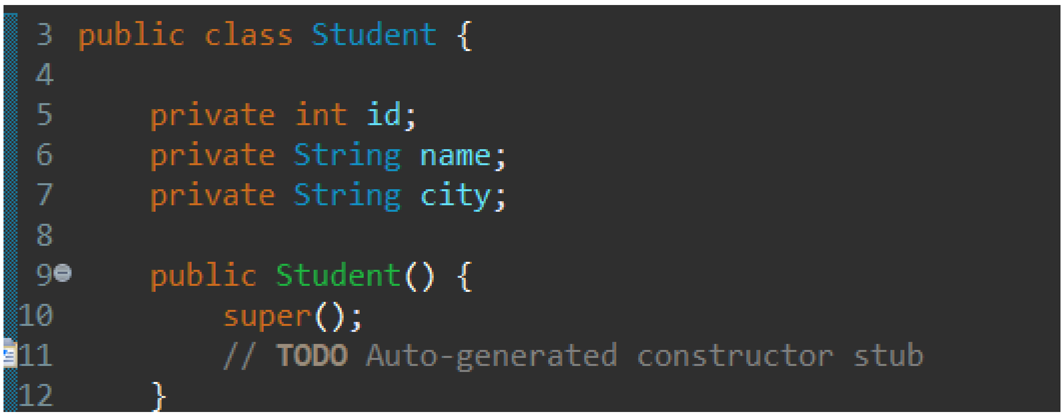


```java
 */
public class App
{
    public static void main( String[] args )
    {
       ApplicationContext context = new ClassPathXmlApplicationContext("com/spring/jdbc/config.xml");
       JdbcTemplate template = context.getBean("jdbcTemplate", JdbcTemplate.class);

       String sql = "Insert into student (id, name, city) values(?, ?, ?)";
       int rs = template.update(sql, 1, "tony", "new york");
       System.out.println(rs);
    }
}
```

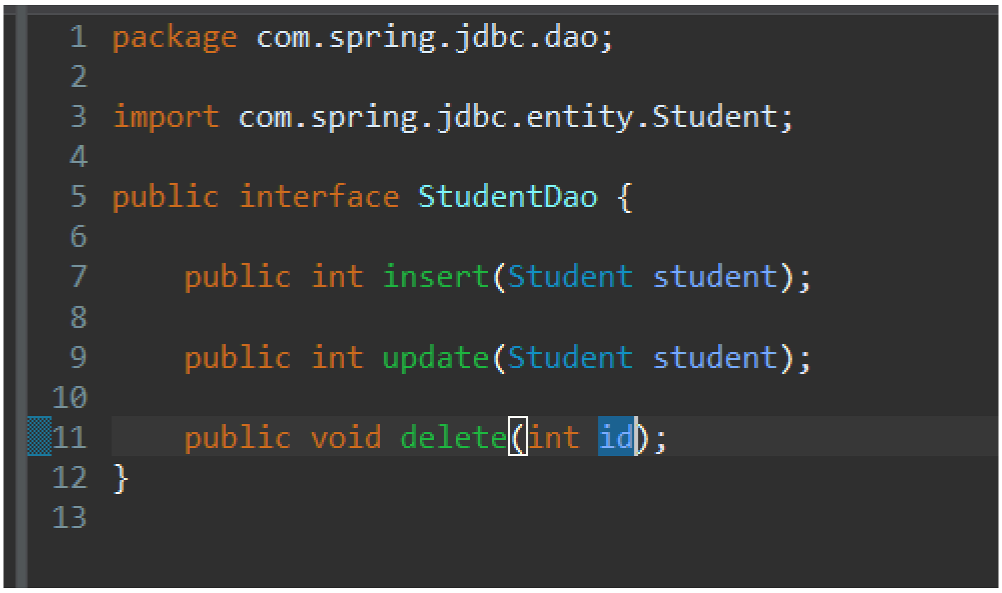
Structured Approach for All Database Operations

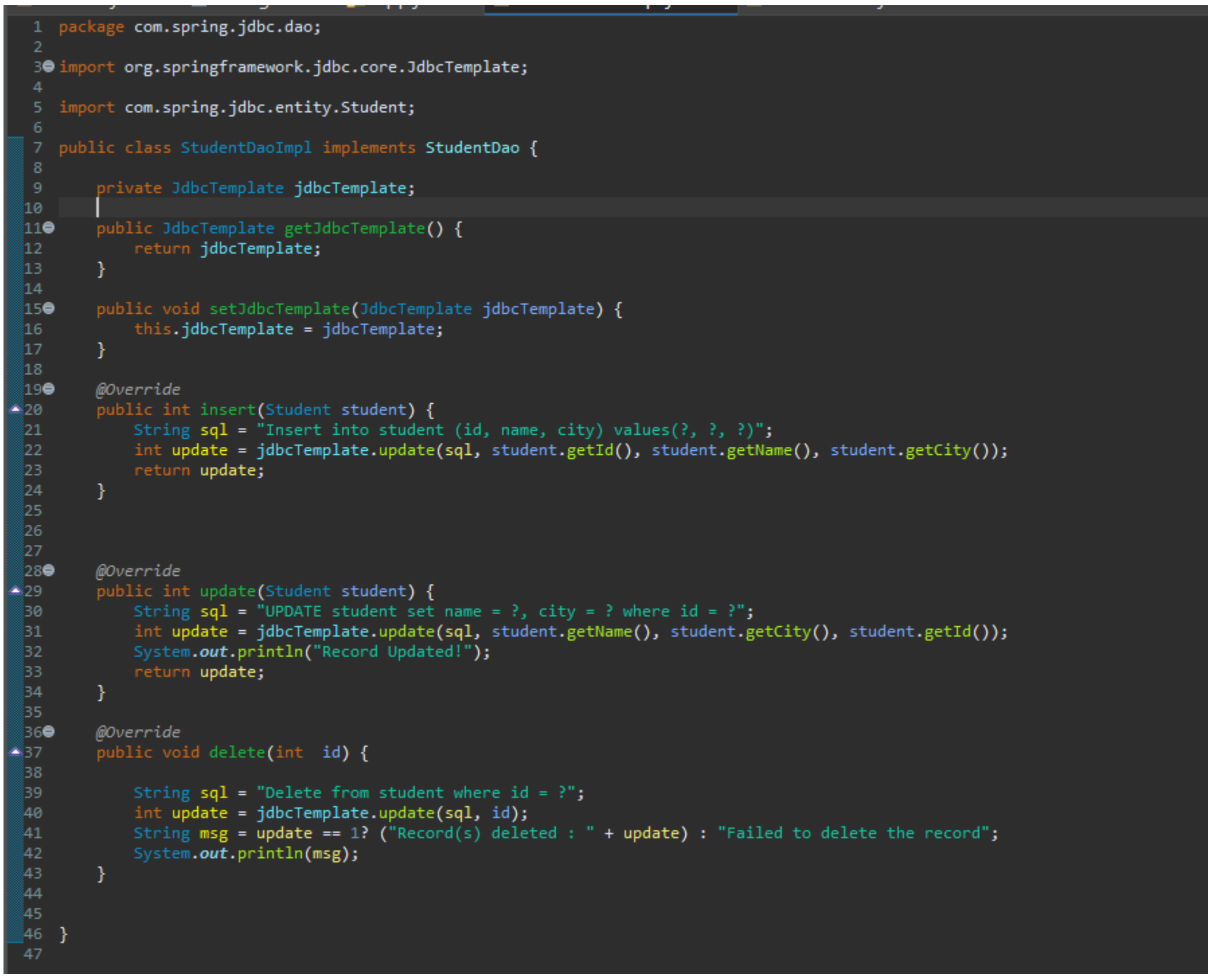
1. **Create an `entity` package:**
    - Add your POJO classes here.
    - These classes will match your database tables (columns as fields).
2. **Create a `dao` package:**
    - Inside it, create an **interface** with methods like `insert()`, `update()`, `delete()`, `getById()`, and `getAll()`.
    - Then create an **implementation class** for this interface, where you will write the actual database code.
3. **In the DAO implementation class:**
    - Create a **`JdbcTemplate` variable**.
    - Inject the `JdbcTemplate` into this class using setter or constructor based injection.
    - Each method will take an respective **entity object** as a parameter to extract data from it (if required).
4. **In the Spring configuration file:**
    - Define the DAO implementation class as a **bean**.
    - Pass the **`JdbcTemplate` bean** to it.
5. **In the main class:**
    - Get the **DAO bean** from the Spring container.
    - Call the required method and pass the populated **entity object** to it.


```
SpringJDBC [Spring-Practice main]
  src/main/java
    com.spring.jdbc
      App.java
      config.xml
    com.spring.jdbc.dao
      StudentDao.java
      StudentDaoImpl.java
    com.spring.jdbc.entity
      Student.java
  src/test/java
```

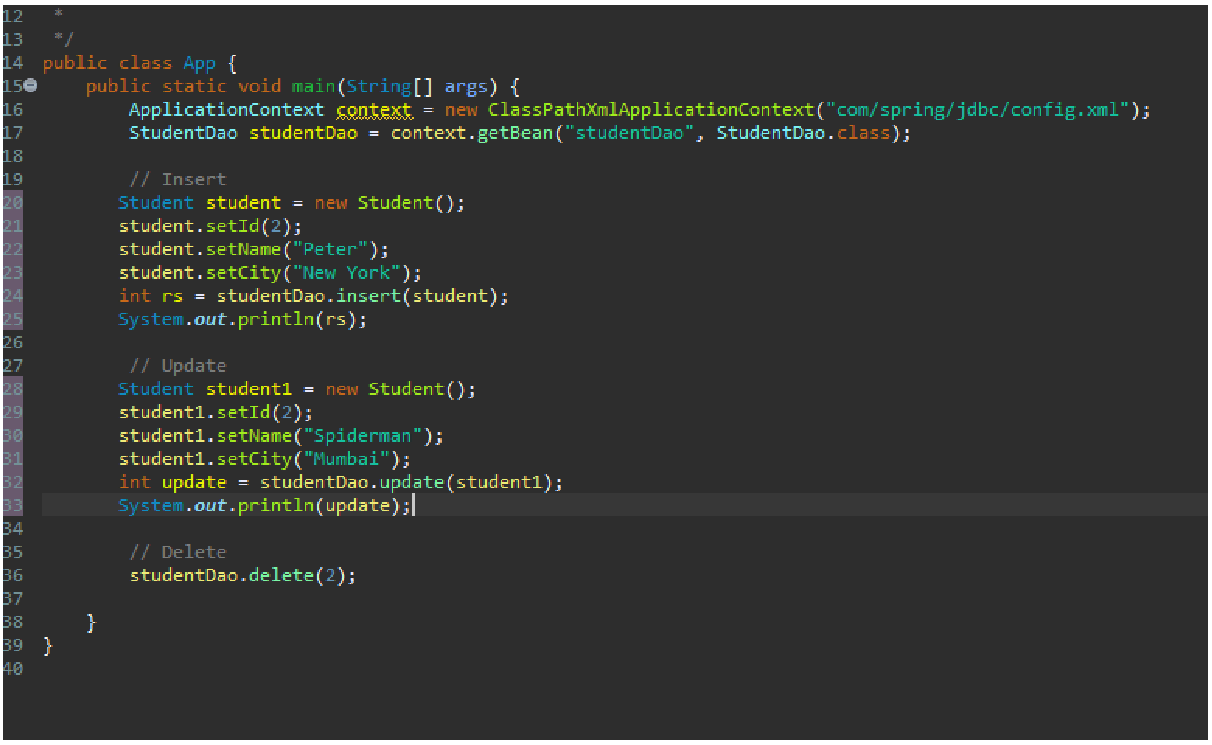

```java
public class Student {

    private int id;
    private String name;
    private String city;

    public Student() {
        super();
        // TODO Auto-generated constructor stub
    }
```

Insert, Update, Delete

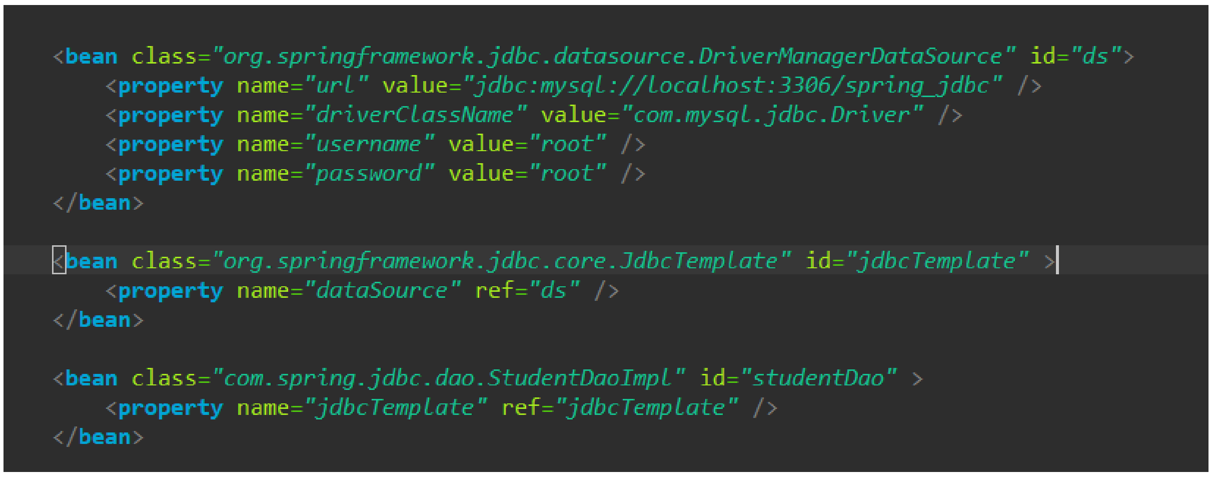

```java
package com.spring.jdbc.dao;

import com.spring.jdbc.entity.Student;

public interface StudentDao {

    public int insert(Student student);

    public int update(Student student);

    public void delete(int id);
}

```



```java
package com.spring.jdbc.dao;

import org.springframework.jdbc.core.JdbcTemplate;

import com.spring.jdbc.entity.Student;

public class StudentDaoImpl implements StudentDao {

    private JdbcTemplate jdbcTemplate;

    public JdbcTemplate getJdbcTemplate() {
        return jdbcTemplate;
    }

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override
    public int insert(Student student) {
        String sql = "Insert into student (id, name, city) values(?, ?, ?)";
        int update = jdbcTemplate.update(sql, student.getId(), student.getName(), student.getCity());
        return update;
    }



    @Override
    public int update(Student student) {
        String sql = "UPDATE student set name = ?, city = ? where id = ?";
        int update = jdbcTemplate.update(sql, student.getName(), student.getCity(), student.getId());
        System.out.println("Record Updated!");
        return update;
    }

    @Override
    public void delete(int  id) {

        String sql = "Delete from student where id = ?";
        int update = jdbcTemplate.update(sql, id);
        String msg = update == 1? ("Record(s) deleted : " + update) : "Failed to delete the record";
        System.out.println(msg);
    }


}
```



```java
 *
 */
public class App {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("com/spring/jdbc/config.xml");
        StudentDao studentDao = context.getBean("studentDao", StudentDao.class);

        // Insert
       Student student = new Student();
       student.setId(2);
       student.setName("Peter");
       student.setCity("New York");
       int rs = studentDao.insert(student);
       System.out.println(rs);

        // Update
       Student student1 = new Student();
       student1.setId(2);
       student1.setName("Spiderman");
       student1.setCity("Mumbai");
       int update = studentDao.update(student1);
       System.out.println(update);

        // Delete
        studentDao.delete(2);

    }
}
```



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

<bean class="org.springframework.jdbc.core.JdbcTemplate" id="jdbcTemplate" >
    <property name="dataSource" ref="ds" />
</bean>

<bean class="com.spring.jdbc.dao.StudentDaoImpl" id="studentDao" >
    <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

Read

What is RowMapper in Spring JDBC?

- RowMapper is an interface.
- It is used to map each row of the result from the database to a Java object.
- Whenever you run a query like `SELECT * FROM table`, Spring gives you each row — RowMapper tells Spring how to convert that row into your POJO (Entity).

How does it works?

- You implement the `RowMapper` interface.
- You override its method `mapRow(ResultSet rs, int rowNum)`.
- Inside `mapRow()`, you pick data from ResultSet and set it into your object.

Selecting a Single Object Using Spring JDBC

- If you want to select a single row (single object) from the database using Spring JDBC,
  you can use the method:
  👉 `jdbcTemplate.queryForObject()`

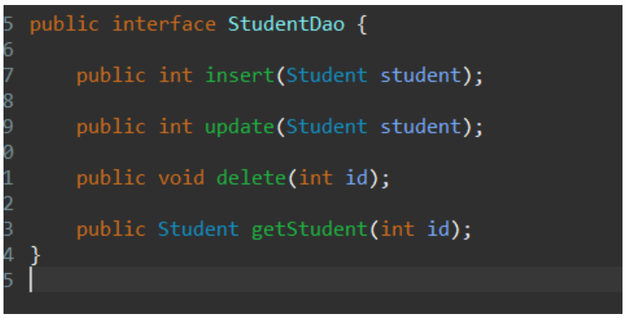
- Syntax:

```
YourObject obj = jdbcTemplate.queryForObject(sqlQuery,
new RowMapperClass(), parameters);
```

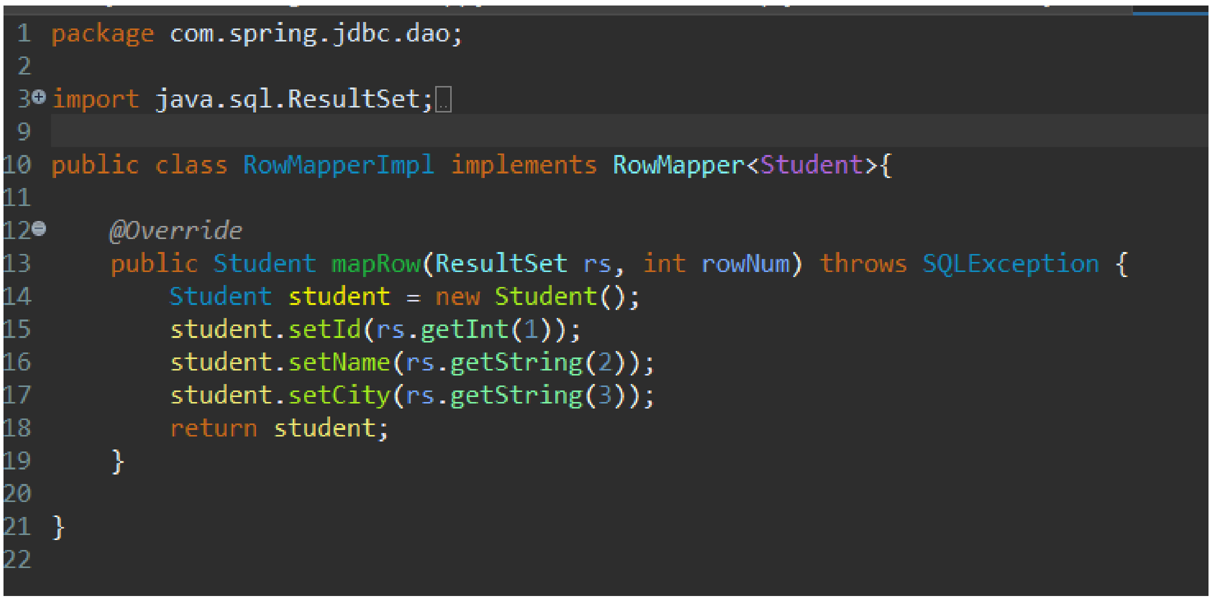
Here:

- `sqlQuery` → Your SQL string (e.g., `"SELECT * FROM student WHERE id=?"`)`RowMapperClass()` → A class that implements `RowMapper` to map the row into your Java object.
- `parameters` → Values to be passed to the query (`?` placeholders).

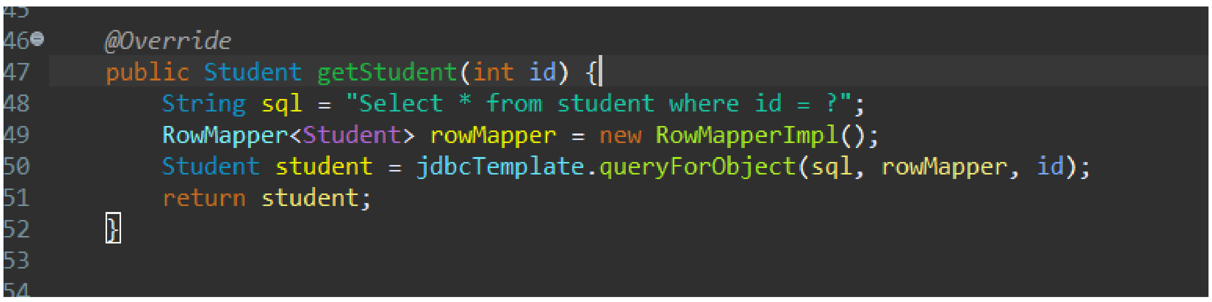
Example :



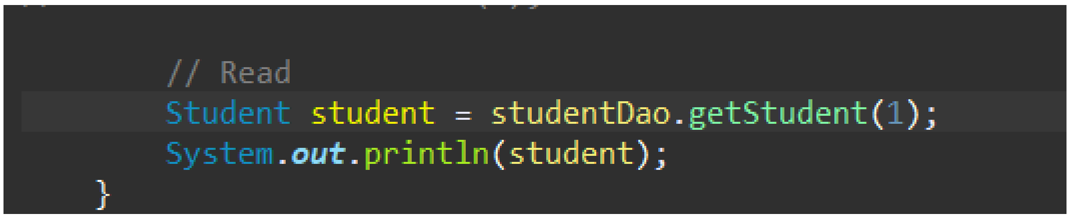
```java
public interface StudentDao {

    public int insert(Student student);

    public int update(Student student);

    public void delete(int id);

    public Student getStudent(int id);
}
```



```java
package com.spring.jdbc.dao;

import java.sql.ResultSet;

public class RowMapperImpl implements RowMapper<Student>{

    @Override
    public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
        Student student = new Student();
        student.setId(rs.getInt(1));
        student.setName(rs.getString(2));
        student.setCity(rs.getString(3));
        return student;
    }

}
```

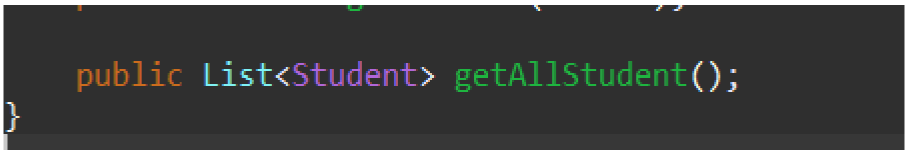


```java
    @Override
    public Student getStudent(int id) {
        String sql = "Select * from student where id = ?";
        RowMapper<Student> rowMapper = new RowMapperImpl();
        Student student = jdbcTemplate.queryForObject(sql, rowMapper, id);
        return student;
    }
```

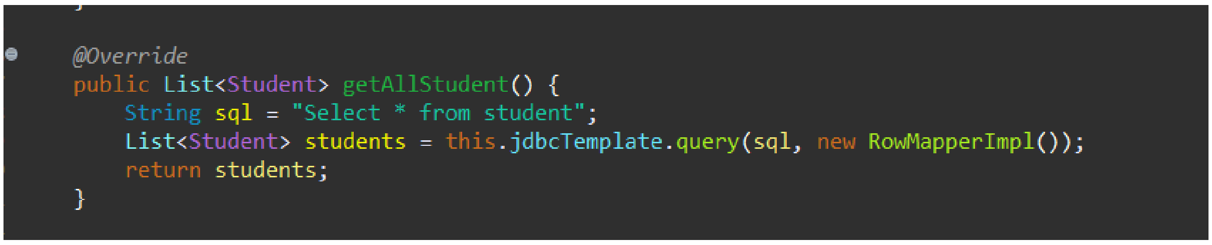


```
        // Read
        Student student = studentDao.getStudent(1);
        System.out.println(student);
    }
```

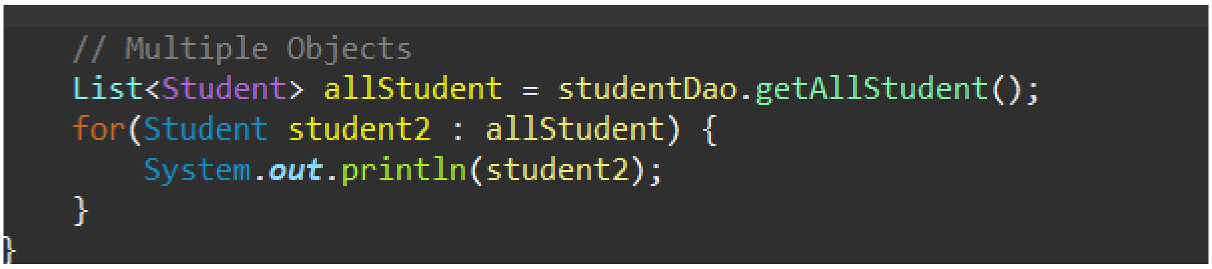
Selecting a Multiple Object Using Spring JDBC



```
    public List<Student> getAllStudent();
}
```



```
    @Override
    public List<Student> getAllStudent() {
        String sql = "Select * from student";
        List<Student> students = this.jdbcTemplate.query(sql, new RowMapperImpl());
        return students;
    }
```

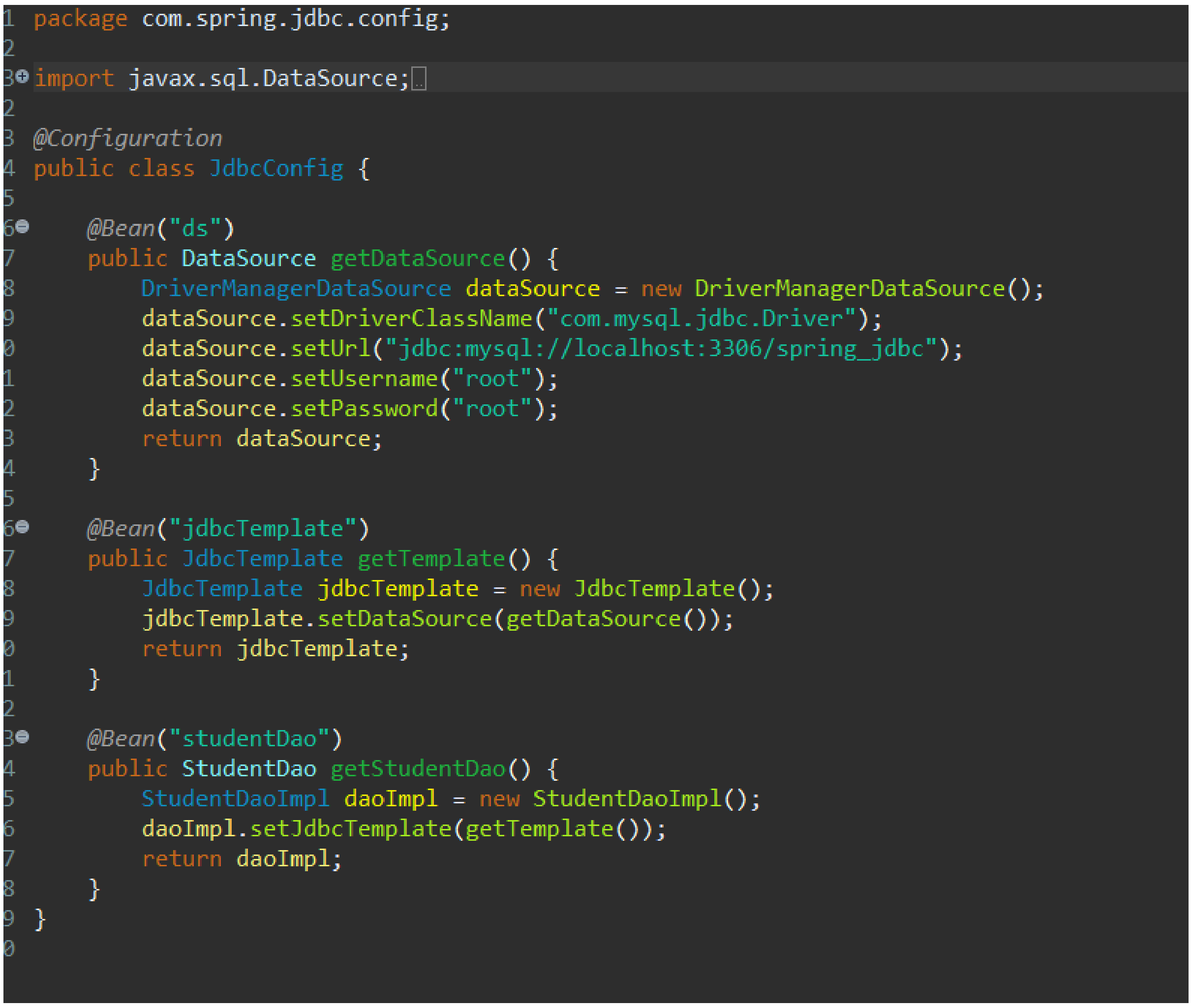


```
    // Multiple Objects
    List<Student> allStudent = studentDao.getAllStudent();
    for(Student student2 : allStudent) {
        System.out.println(student2);
    }
}
```

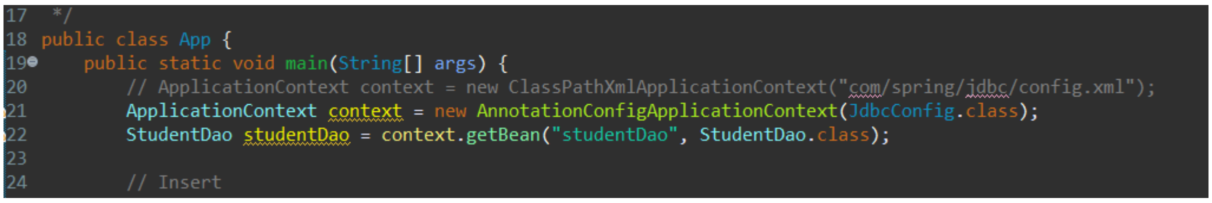
Java Base Configuration

Now we will use java base configuration and remove the xml file completely from our application.

Example :



```java
package com.spring.jdbc.config;

import javax.sql.DataSource;

@Configuration
public class JdbcConfig {

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

    @Bean("jdbcTemplate")
    public JdbcTemplate getTemplate() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(getDataSource());
        return jdbcTemplate;
    }

    @Bean("studentDao")
    public StudentDao getStudentDao() {
        StudentDaoImpl daoImpl = new StudentDaoImpl();
        daoImpl.setJdbcTemplate(getTemplate());
        return daoImpl;
    }
}
```



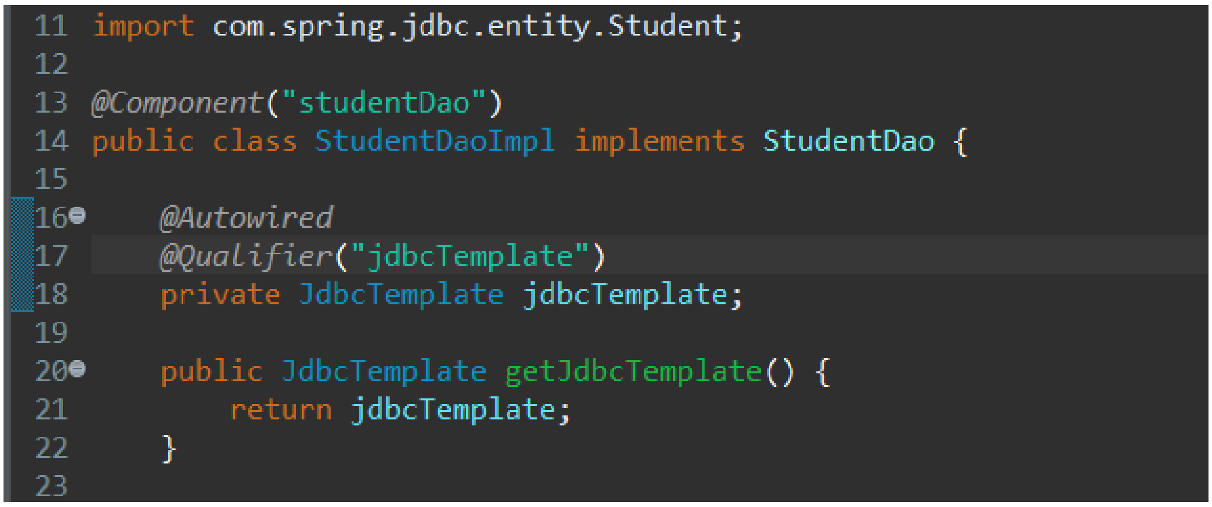
```java
 */
public class App {
    public static void main(String[] args) {
        // ApplicationContext context = new ClassPathXmlApplicationContext("com/spring/jdbc/config.xml");
        ApplicationContext context = new AnnotationConfigApplicationContext(JdbcConfig.class);
        StudentDao studentDao = context.getBean("studentDao", StudentDao.class);

        // Insert
```

Rest all the code will be the same…

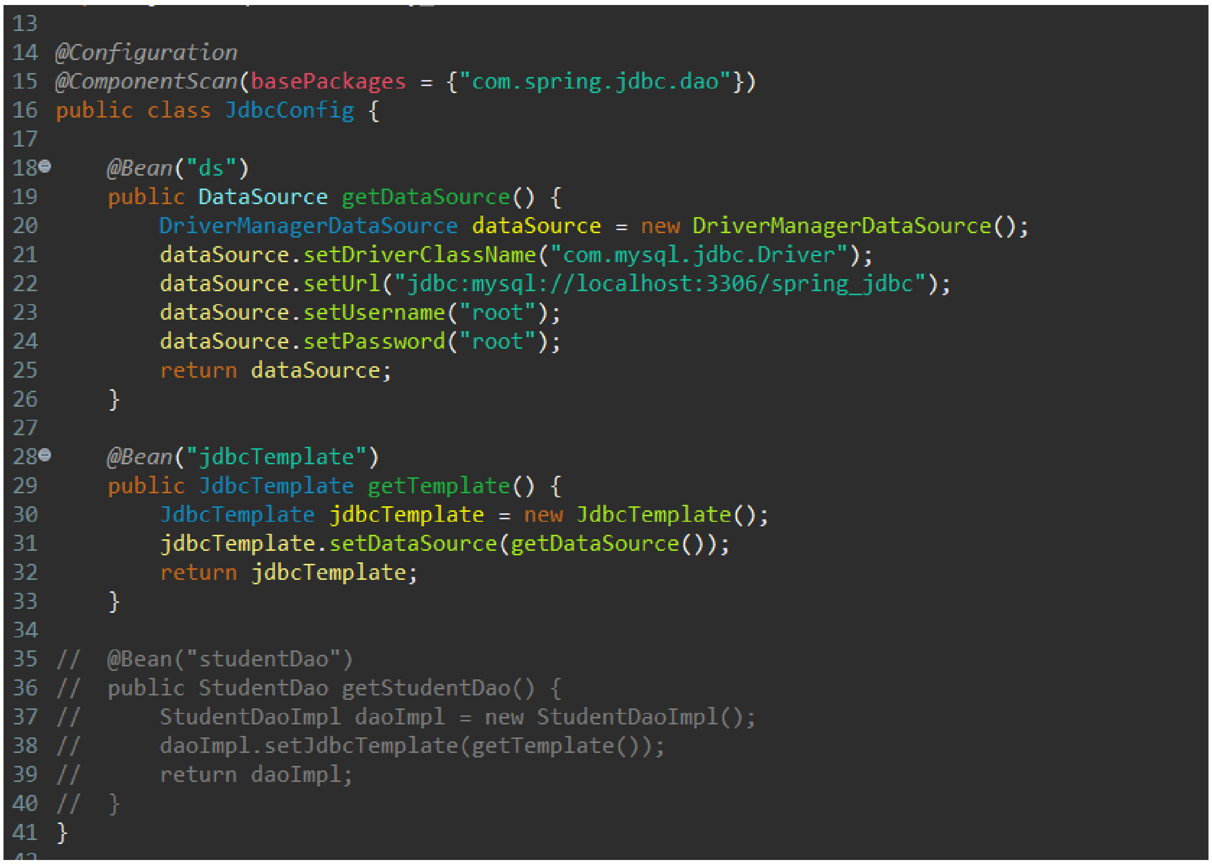
Autowiring in Spring JDBC

Instead of declaring a bean in java config class, declare the bean using @Comonent anno on bean itself and autowire the dependencies

Example



```java
import com.spring.jdbc.entity.Student;

@Component("studentDao")
public class StudentDaoImpl implements StudentDao {

    @Autowired
    @Qualifier("jdbcTemplate")
    private JdbcTemplate jdbcTemplate;

    public JdbcTemplate getJdbcTemplate() {
        return jdbcTemplate;
    }

```



```java

@Configuration
@ComponentScan(basePackages = {"com.spring.jdbc.dao"})
public class JdbcConfig {

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

    @Bean("jdbcTemplate")
    public JdbcTemplate getTemplate() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(getDataSource());
        return jdbcTemplate;
    }

//  @Bean("studentDao")
//  public StudentDao getStudentDao() {
//      StudentDaoImpl daoImpl = new StudentDaoImpl();
//      daoImpl.setJdbcTemplate(getTemplate());
//      return daoImpl;
//  }
}
```