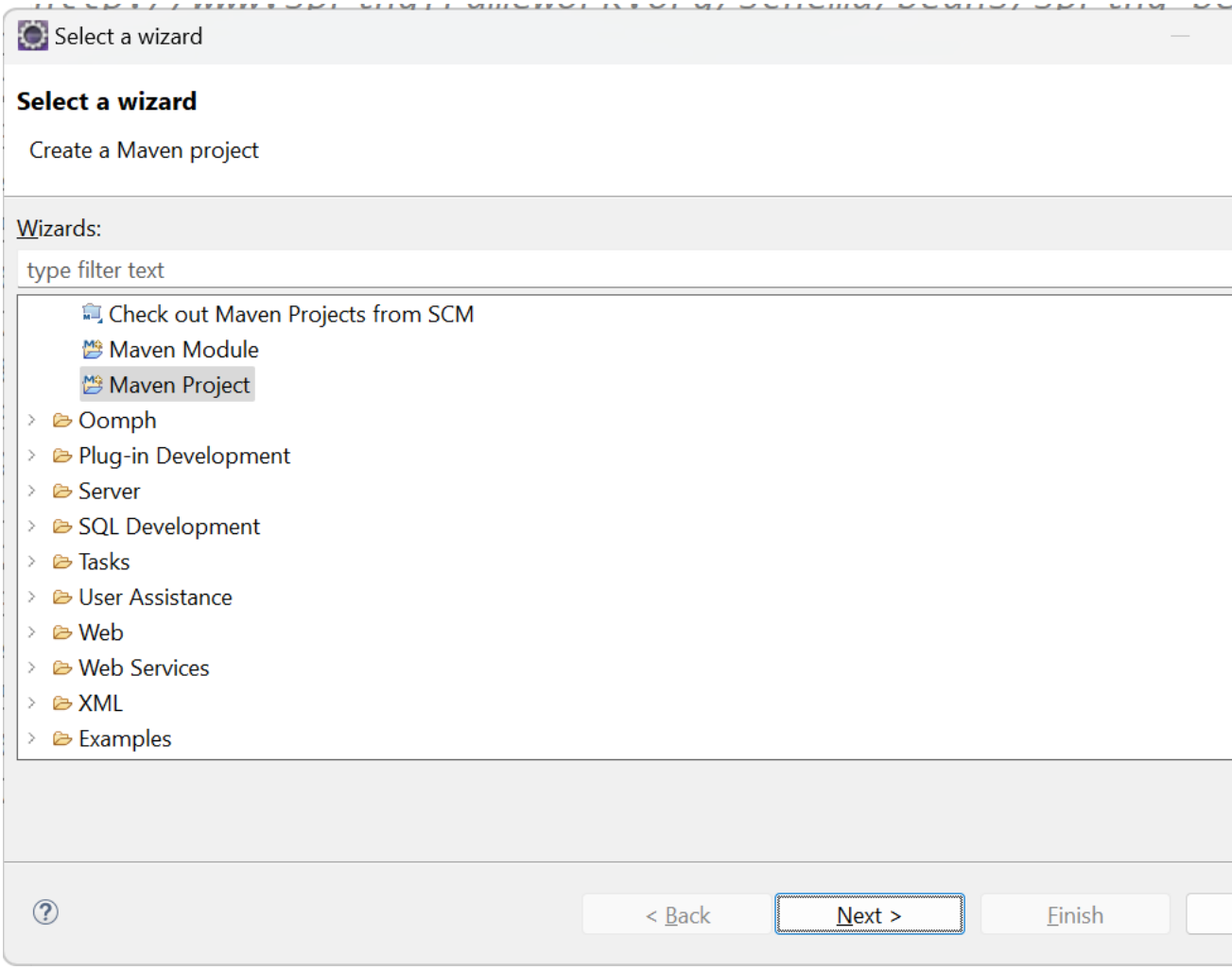
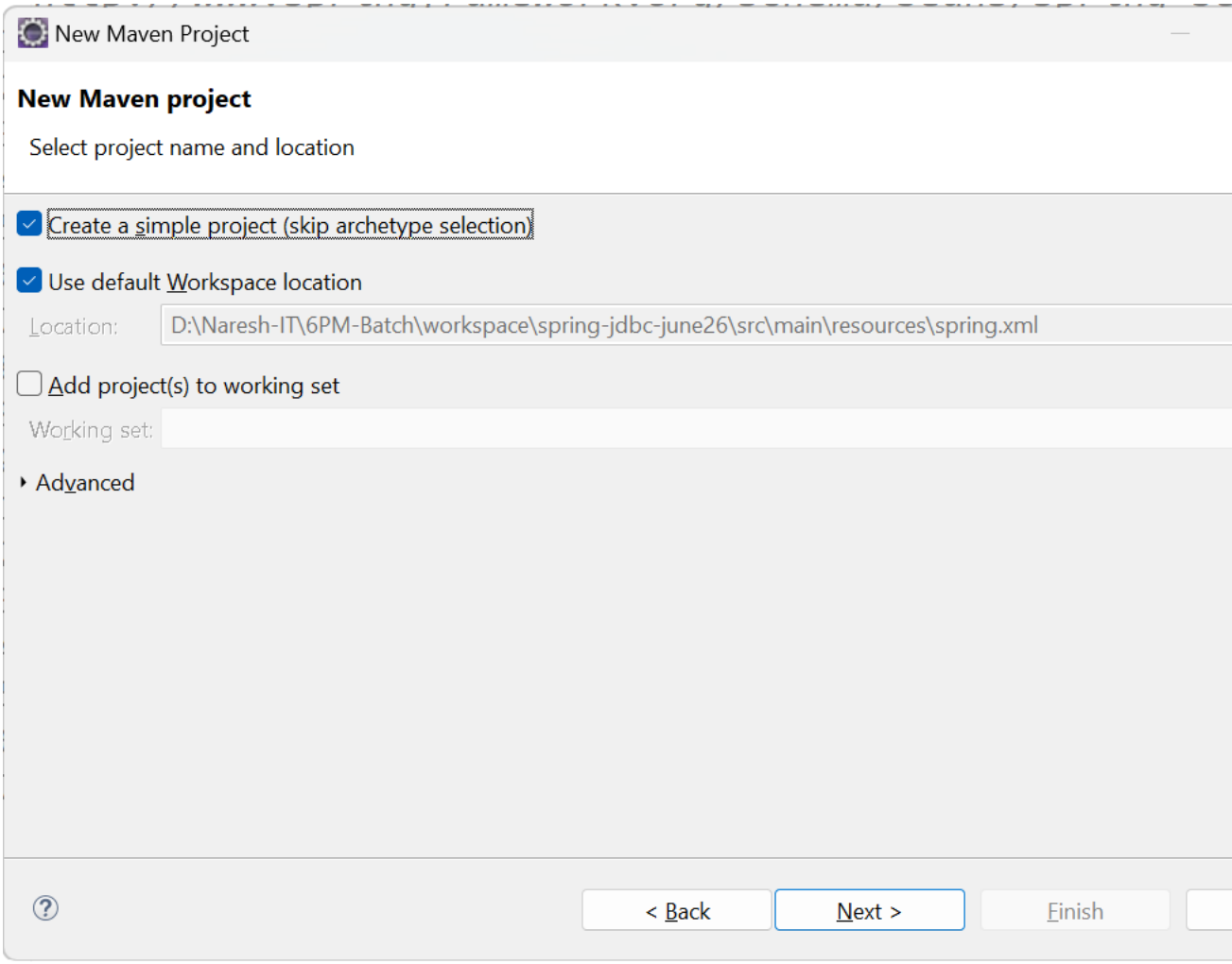
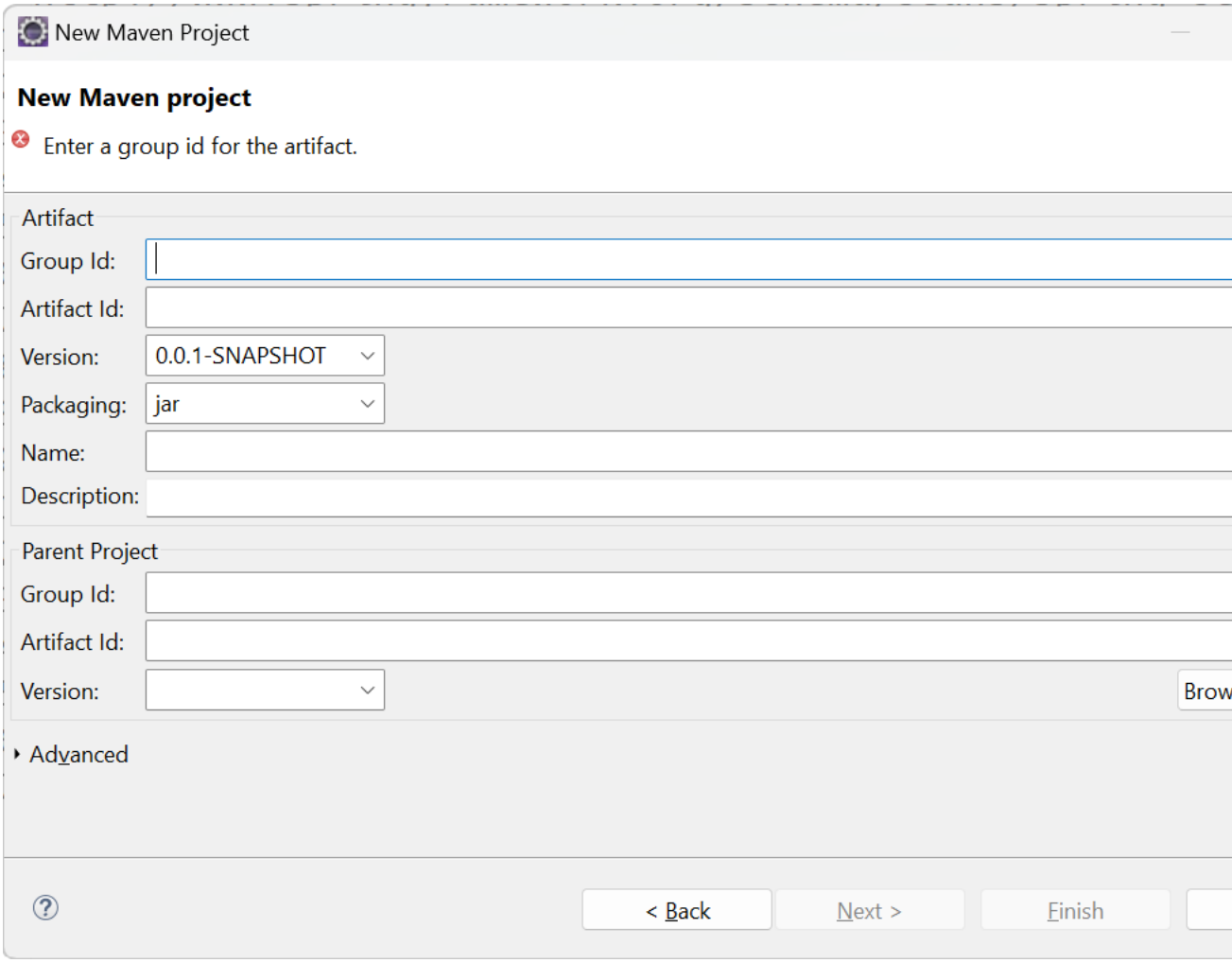
Step By Step Procedure of Spring JDBC Example

1) **Open the eclipse and create Maven Project**

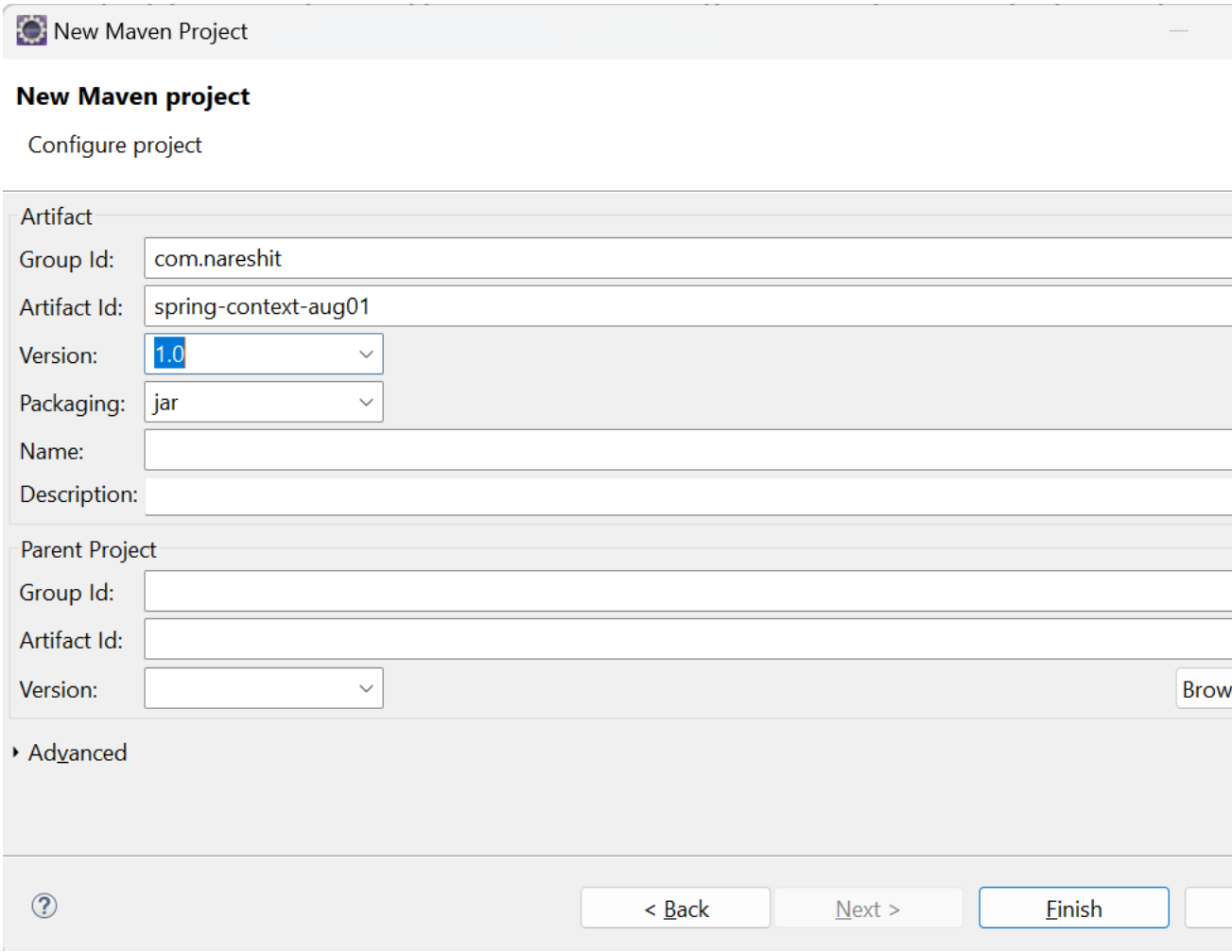


2) **Check the option "Create a simple project"**

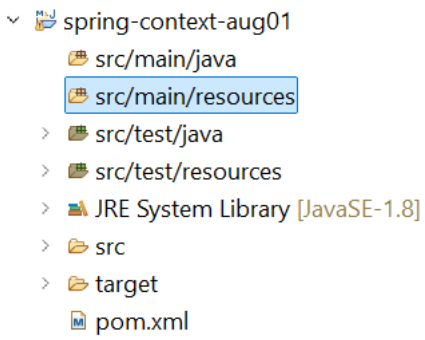

New Maven Project
New Maven project
Select project name and location
Create a simple project (skip archetype selection)
Use default Workspace location
Location:
D:\Naresh-IT\6PM-Batch\workspace\spring-jdbc-june26\src\main\resources\spring.xml
Add project(s) to working set
Working set:
Advanced
< Back
Next >
Finish



3) **Enter**
   GroupId: com.nareshit
   ArtifactId: spring-jdbc-aug01
   Version: 1.0 and click on finish



4) **After clicking on the finish button, it creates a maven project in the project explorer as follows:**



5) **Open pom.xml and it looks as below**

spring-context-aug01/pom.xml

```
https://maven.apache.org/xsd/maven-4.0.0.xsd (xsi:schemaLocation)
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="htt
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.nareshit</groupId>
  <artifactId>spring-context-aug01</artifactId>
  <version>1.0</version>
</project>
```

6) **Add the required dependencies. Because it is a spring JDBC example, we need to add spring-context and spring-jdbc and mysql-connect-j dependencies in pom.xml. Open www.mavenrepository.com web site and look for the above dependencies and enter in pom.xml**

```
<properties>
      <spring.version>6.1.10</spring.version>

      <mysql.version>8.4.0</mysql.version>
</properties>
<dependencies>
      <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
      </dependency>
      <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring.version}</version>
      </dependency>
      <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <version>${mysql.version}</version>
      </dependency>
</dependencies>
```

spring-context-aug01/pom.xml × spring.xml spring-jdbc-june26/pom.xml

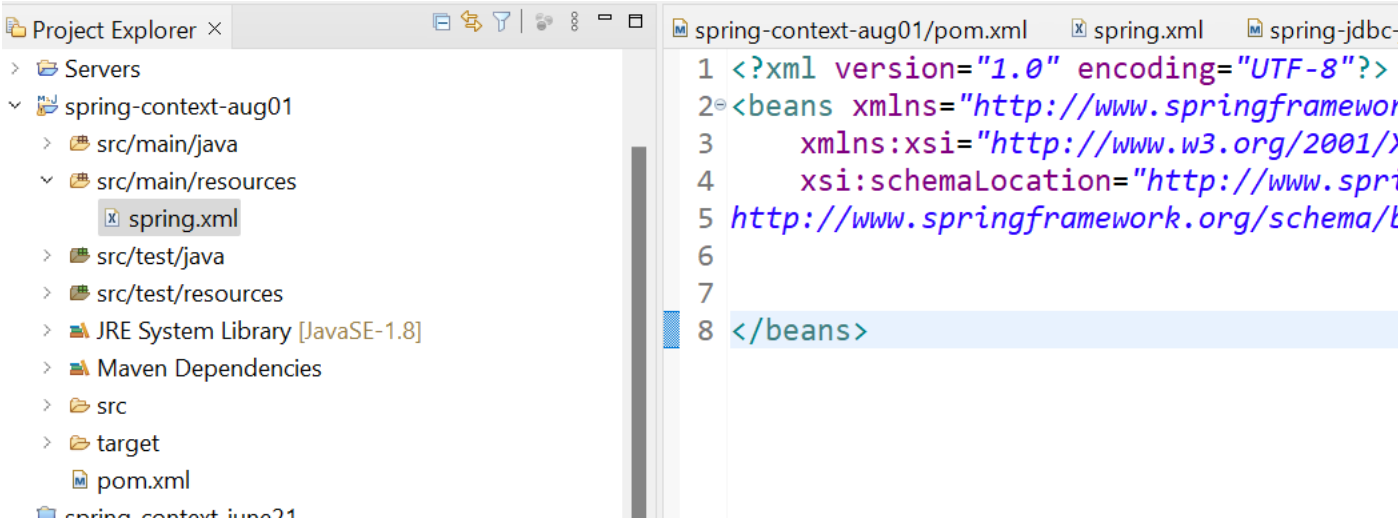
```
https://maven.apache.org/xsd/maven-4.0.0.xsd (xsi:schemaLocation)
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.c
2    <modelVersion>4.0.0</modelVersion>
3    <groupId>com.nareshit</groupId>
4    <artifactId>spring-context-aug01</artifactId>
5    <version>1.0</version>
6    <properties>
7          <spring.version>6.1.10</spring.version>
8          <mysql.version>8.4.0</mysql.version>
9      </properties>
10     <dependencies>
11         <dependency>
12             <groupId>org.springframework</groupId>
13             <artifactId>spring-context</artifactId>
14             <version>${spring.version}</version>
15         </dependency>
16         <dependency>
17             <groupId>org.springframework</groupId>
18             <artifactId>spring-jdbc</artifactId>
19             <version>${spring.version}</version>
20         </dependency>
21         <dependency>
22             <groupId>com.mysql</groupId>
23             <artifactId>mysql-connector-j</artifactId>
24             <version>${mysql.version}</version>
25         </dependency>
26     </dependencies>
27 </project>
```

Overview | Dependencies | Dependency Hierarchy | Effective POM | pom.xml

Writable

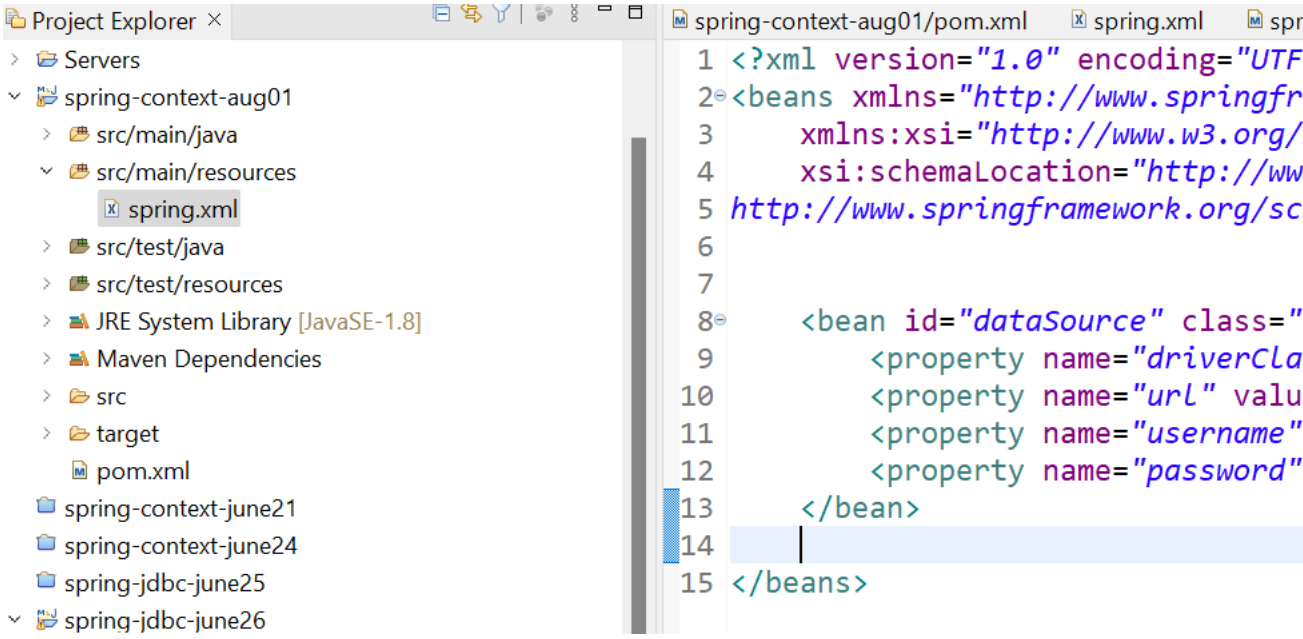
7) **Create a spring configuration file spring.xml. This XML file creation is not available in eclipse tool. It was there earlier but that support is removed from the tool. So download the spring configuration file from google and save that file as spring.xml**

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

      <!— This is where you need to define all the spring beans
-->

</beans>
```



8) **To connect to Database, spring provided a class: org.springframework.jdbc.datasource.DriverManagerDataSource. So define this bean in spring.xml**
With the database properties as below:
DriveClassName: com.mysql.cj.jdbc.Driver
url: *jdbc:mysql://localhost:3306/nareshit-4pm*
username: root
password: <password of your database root user>

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSourc
e">
        <property name="driverClassName"
value="com.mysql.cj.jdbc.Driver"/>
        <property name="url"
value="jdbc:mysql://localhost:3306/nareshit-6pm"/>
        <property name="username" value="root"/>
        <property name="password" value="techno"/>
</bean>
```



9) **Open the Mysqlworkbenach tool and create a table called *student* and define the java class Student.java to represent database table → student, with all the required variables**

```
package com.nareshit.model;

public class Student {
	private Integer stuId;
	private String firstName;
	private String lastName;
	public Integer getStuId() {
		return stuId;
	}
	public void setStuId(Integer stuId) {
		this.stuId = stuId;
	}
	public String getFirstName() {
		return firstName;
	}
	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}
	public String getLastName() {
		return lastName;
	}
	public void setLastName(String lastName) {
		this.lastName = lastName;
	}
```

```
	public Student() {
		super();
	}
	public Student(Integer stuId, String firstName,
String lastName) {
		super();
		this.stuId = stuId;
		this.firstName = firstName;
		this.lastName = lastName;
	}
	@Override
	public String toString() {
		return "Student [stuId=" + stuId + ",
firstName=" + firstName + ", lastName=" + lastName + "]";
	}

}
```

10) To implement CRUD (create, Read, Update and Delete) operations for the table student, It is recommended to interface implemented approach. It is industry standard suggested approach.

11) Create StudentDao.java and StudentDaoImpl.java classes

```
package com.nareshit.dao;

import java.util.List;

import com.nareshit.model.Student;

public interface StudentDao {

	public void saveStudent(Student st);

	public void updateStudent(Student st);

	public void deleteStudent(Integer stuId);

	public Student getStudent(Integer stuId);

	public List<Student> getAllStudents();

}
```

```
package com.nareshit.dao;
import java.util.List
import com.nareshit.model.Student;
public class StudentDaoImpl implements StudentDao{
	@Override
	public void saveStudent(Student st) {
		// TODO Auto-generated method stub
	}
	@Override
	public void updateStudent(Student st) {
		// TODO Auto-generated method stub
	}
	@Override
	public void deleteStudent(Integer stuId) {
		// TODO Auto-generated method stub
	}
	@Override
	public Student getStudent(Integer stuId) {
		// TODO Auto-generated method stub
		return null;
	}
	@Override
	public List<Student> getAllStudents() {
		// TODO Auto-generated method stub
		return null;
	}
```

}

12) **Now, StudentDaoImpl.java needs a database connection to execute the queries. To inject dataSource to StudentDaoImpl.java class, let us define StudentDaoImpl.java into spring.xml so that we can inject dataSource spring bean using *ref keyword***

13) **But How do I inject datasource in StudentDaoImpl.java ???? We already learned in spring-context module that to inject any object into any other object either it should contain either setter method or constructor or both.**

14) **So let us create a setter method → setDataSource(DataSource dataSource), to hold the DataSource object.**

```java
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

import com.nareshit.model.Student;

public class StudentDaoImpl implements StudentDao{

    public void setDataSource(DataSource dataSource) {

    }

    @Override
    public void saveStudent(Student st) {
        // TODO Auto-generated method stub

    }

    @Override
    public void updateStudent(Student st) {
        // TODO Auto-generated method stub

    }

    @Override
    public void deleteStudent(Integer stuId) {
        // TODO Auto-generated method stub

    }

    @Override
    public Student getStudent(Integer stuId) {
        // TODO Auto-generated method stub
        return null;
    }
```

15) **After getting datasource into StudentDaoImpl.java, we need to use that datasource to execute queries, spring provided a class called → org.springframework.jdbc.core.JdbcTemplate**

16) **So let us us define JdbcTemplate object in StudentDaoImpl.java class as below:**

spring-context... | spring.xml | spring-jdbc-ju... | spring.xml | Student.java | S

```java
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

import com.nareshit.model.Student;

public class StudentDaoImpl implements StudentDao{

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }

```

So the following changes you will find in spring.xml and StudentDaoImpl.java components:

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

        <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataS
ource">
                <property name="driverClassName"
value="com.mysql.cj.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/nareshit-6pm"/>
                <property name="username" value="root"/>
                <property name="password" value="techno"/>
        </bean>
```

```
        <bean id="dao" class="com.nareshit.dao.StudentDaoImpl">
                <property name="dataSource" ref="dataSource"/>
        </bean>
</beans>
```

```
package com.nareshit.dao;

import java.util.List

import com.nareshit.model.Student;

public class StudentDaoImpl implements StudentDao{

        private JdbcTemplate jdbcTemplate;

        public void setDataSource(DataSource dataSource) {

                jdbcTemplate = new JdbcTemplate(dataSource);

        }

        @Override

        public void saveStudent(Student st) {

                // TODO Auto-generated method stub

        }

        @Override

        public void updateStudent(Student st) {

                // TODO Auto-generated method stub

        }

        @Override

        public void deleteStudent(Integer stuId) {

                // TODO Auto-generated method stub

        }

        @Override

        public Student getStudent(Integer stuId) {
```

```
		// TODO Auto-generated method stub
		return null;
	}
	@Override
	public List<Student> getAllStudents() {
		// TODO Auto-generated method stub
		return null;
	}
}
```

17) By using jdbcTemplate.update() method, we can execute all the DML queries. For that prepare the queries and implements as follows:

```
package com.nareshit.dao;
import java.util.List;
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import com.nareshit.model.Student;

public class StudentDaoImpl implements StudentDao{

	private String INSERT = "insert into student values(?,?,?)";
	private String UPDATE = "update student set first_name=?,
last_name=? where stu_id=?";
	private String DELETE = "delete from student where
stu_id=?";
	private String GET_ONE = "select * from student where
stu_id=?";
	private String GET_ALL = "select * from student";

	private JdbcTemplate jdbcTemplate;
	private RowMapper<Student> rowMapper;;
```

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

	public void setRowMapper(RowMapper<Student>
rowMapper) {
		this.rowMapper = rowMapper;
	}
	@Override
	public void saveStudent(Student st) {

	jdbcTemplate.update(INSERT,st.getStuId(),st.getFirstName(),
st.getLastName());
	}

	@Override
	public void updateStudent(Student st) {

	jdbcTemplate.update(UPDATE,st.getFirstName(),st.getLastN
ame(),st.getStuId());
	}

	@Override
	public void deleteStudent(Integer stuId) {
		jdbcTemplate.update(DELETE,stuId);
	}


	@Override

	public Student getStudent(Integer stuId) {

		// TODO Auto-generated method stub

		return null;

	}

	@Override
```

```
        public List<Student> getAllStudents() {
            // TODO Auto-generated method stub
            return null;
        }
    }
```

18) To implement getting results from database, Spring provided different approach. Because Spring JDBC is nothing but a wrapper implementation around normal JDBC. So Spring still get the results in the form of ResultSet object only.

19) We need to convert the ResultSet object into our custom Student object. For that we need to implement a class which needs to implement an interface RowMapper.java.

```
package com.nareshit.dao;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
import com.nareshit.model.Student;
public class StudentRowMapper implements RowMapper<Student>{
    @Override
    public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
        Student st = new Student();
        st.setStuId(rs.getInt("stu_id"));
        st.setFirstName(rs.getString("first_name"));
        st.setLastName(rs.getString("last_name"));
        return st;
```

```
        }
}
```

StduentDaoImpl.java

```
package com.nareshit.dao;
import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import com.nareshit.model.Student;
public class StudentDaoImpl implements StudentDao{
        private String INSERT = "insert into student values(?,?,?)";
        private String UPDATE = "update student set first_name=?,
last_name=? where stu_id=?";
        private String DELETE = "delete from student where
stu_id=?";
        private String GET_ONE = "select * from student where
stu_id=?";
        private String GET_ALL = "select * from student";

        private JdbcTemplate jdbcTemplate;
        private RowMapper<Student> rowMapper;;

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

```
	}
	public void setRowMapper(RowMapper<Student>
rowMapper) {
		this.rowMapper = rowMapper;
	}
	@Override
	public void saveStudent(Student st) {

	jdbcTemplate.update(INSERT,st.getStuId(),st.getFirstName(),
st.getLastName());
	}
	@Override
	public void updateStudent(Student st) {

	jdbcTemplate.update(UPDATE,st.getFirstName(),st.getLastN
ame(),st.getStuId());
	}
	@Override
	public void deleteStudent(Integer stuId) {
		jdbcTemplate.update(DELETE,stuId);
	}
	@Override
	public Student getStudent(Integer stuId) {
		List<Student> students =
jdbcTemplate.query(GET_ONE,new Object[] {stuId}, rowMapper);
		return students.get(0);
	}
	@Override
```

```
        public List<Student> getAllStudents() {
                List<Student> students =
jdbcTemplate.query(GET_ALL, rowMapper);
                return students;
        }
}
```

20) Finally, to test whether our implementation is working as expected or not. For that I have created a test class under **src/test/java** folder as follows:

21) To read the Spring bean context, we need to read the **spring.xml** file. For that spring provided a class → org.springframework.context.support.ClassPathXmlApplicationContext

```
package com.nareshit.test
import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlA
pplicationContext;
import
org.springframework.jdbc.datasource.DriverManager
DataSource;

import com.nareshit.dao.StudentDao;
import com.nareshit.model.Student;

public class Main {
     public static void main(String[] args) {
          ApplicationContext ctx = new
ClassPathXmlApplicationContext("spring.xml");
          StudentDao dao =
(StudentDao)ctx.getBean("dao");

          //Test to save Student Object
          Student st = new Student(2, "Hello",
"World");
          dao.saveStudent(st);
```

```
        System.out.println("Student record is
successfully inserted..");

        //Test to get a Student and update the
student too
        Student s1 = dao.getStudent(2);
        System.out.println(s1);
        System.out.println("Student record read
is successfull..");

        s1.setFirstName("Praveen-1");
        s1.setLastName("Praveen-2");
        dao.updateStudent(s1);
        System.out.println("Student record is
successfully updated..");

        //Test to get all the student records
from database
        for(Student s : dao.getAllStudents())
            System.out.println(s);

        //Test to delete a student record from
database table
        dao.deleteStudent(1);
        System.out.println("Student record is
successfully deleted..");
```

```
        }
    }
```