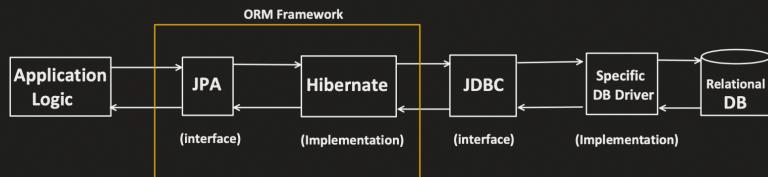


Spring boot: JPA (Part-1)



Before going to JPA, lets recall JDBC

JDBC (Java Database Connectivity) provides an **Interface** to:

- Make connection with DB
- Query DB
- and process the result

Actual implementation is provided by **Specific DB Drivers**.

Report Abuse X

For example:

MySQL

- o Driver : Connector/J
- o Class : *com.mysql.cj.jdbc.Driver*

PostgreSQL

- o Driver : PostgreSQL JDBC Driver
- o Class : *org.postgresql.Driver*

H2 (in-memory)

- o Driver : H2 Database Engine
- o Class : *org.h2.Driver*

Using JDBC without Springboot

```

public class DatabaseConnection {
    public Connection getConnection() {
        try {
            // H2 Driver loading
            Class.forName("org.h2.Driver");
            // Establish connection with DB
            return DriverManager.getConnection("jdbc:h2:mem:userDB", "sa", "sa");
        } catch (ClassNotFoundException | SQLException e) {
            //handle exception
        }
        return null;
    }
}

public class UserDAO {
    public void createTable() {
        try {
            Connection connection = new DatabaseConnection().getConnection();
            Statement statement = connection.createStatement();
            String sql = "CREATE TABLE user(id INT AUTO_INCREMENT PRIMARY KEY, user_name VARCHAR(100), age INT)";
            statement.executeUpdate(sql);
        } catch (SQLException e) { /* handle exception */ }
        finally { /* close statement,query and db connection */ }
    }

    public void createUsers(String userName, int userAge) {
        try {
            Connection connection = new DatabaseConnection().getConnection();
            String sqlQuery = "INSERT INTO user(user_name, age) VALUES (?, ?)";
            PreparedStatement preparedStatement = connection.prepareStatement(sqlQuery);
            preparedStatement.setString(1, userName);
            preparedStatement.setInt(2, userAge);
            preparedStatement.executeUpdate();
        } catch (SQLException e) { /* handle exception */ }
        finally { /* close preparedStatement and db connection */ }
    }

    public void readUsers() {
        try {
            Connection connection = new DatabaseConnection().getConnection();
            String sqlQuery = "SELECT * FROM user";
            PreparedStatement preparedStatement = connection.prepareStatement(sqlQuery);
            ResultSet output = preparedStatement.executeQuery();
            while (output.next()) {
                String user_id = output.getInt("user_id") +
                    " - " + output.getString("user_name") +
                    " - " + output.getInt("user_age");
                System.out.println(userDetails);
            }
        } catch (SQLException e) { /* handle exception */ }
        finally { /* close preparedStatement and db connection */ }
    }
}
  
```

If we see above example:

- Connection
- Statement
- PreparedStatement
- ResultSet etc.

All are interfaces which JDBC provide and each specific driver provide the implementation for it.

But there are so much of **BOILERCODE** present like:

- Driver class loading
- DB Connection Making
- Exception Handling
- Closing of the DB connection and other objects like Statement etc.
- Manual handling of DB Connection Pool
- Etc..

Using JDBC with Springboot

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Springboot provides **JdbcTemplate** class, which helps to remove all the boiler code.

```
@Component
public class UserService {
    @Autowired
    UserRepository userRepository;

    public void createTable() {
        userRepository.createTable();
    }

    public void insertUser(String userName, int age) {
        userRepository.insertUser(userName, age);
    }

    public List getUsers() {
        List users = userRepository.getUsers();
        for(User user : users) {
            System.out.println(user.userId + ":" + user.getUserName() + ":" + user.getAge());
        }
        return users;
    }
}

@Repository
public class UserRepository {
    @Autowired
    JdbcTemplate jdbcTemplate;

    public void createtable() {
        jdbcTemplate.execute("CREATE TABLE users (user_id INT AUTO_INCREMENT PRIMARY KEY, " +
                "user_name VARCHAR(100), age INT)");
    }

    public void insertUser(String name, int age) {
        String insertQuery = "INSERT INTO users (user_name, age) VALUES (?, ?)";
        jdbcTemplate.update(insertQuery, name, age);
    }

    public List<User> getusers() {
        String selectQuery = "SELECT * FROM users";
        return jdbcTemplate.query(selectQuery, (rs, rowNum) -> {
            User user = new User();
            user.setId(rs.getInt("user_id"));
            user.setName(rs.getString("user_name"));
            user.setAge(rs.getInt("age"));
            return user;
        });
    }
}
```

application.properties

```
spring.datasource.url=jdbc:h2:mem:userDB
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
```

public class User {

```
    int userId;
    String userName;
    int age;

    //getters and setters
}
```

Driver class loading -

JdbcTemplate load it at the time of application startup in DriverManager class.

DB Connection Making -

JdbcTemplate takes care of it, whenever we execute any query.

Exception Handling -

In Plain JDBC, we get very abstracted 'SQLException' but in jdbcTemplate, we get granular error like DuplicateKeyException, QueryTimeoutException etc.. (defined in org.springframework.dao package).

Closing of the DB connection and other resources

Closing of the DB Connection and other Resources

when we invoke update or query method, after success or failure of the operation, jdbcTemplate takes care of either closing or return the connection to Pool itself.

- Manual handling of DB Connection Pool -

Springboot provides default jdbc connection pool i.e. 'HikariCP' with Min and Max pool size of 10. And we can change the configuration in '*application.properties*'

```
spring.datasource.hikari.maximum-pool-size=10  
spring.datasource.hikari.minimum-idle=5
```

We can also configure different jdbc connection pool if we want like below:

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public DataSource dataSource() {  
        HikariDataSource dataSource = new HikariDataSource();  
        dataSource.setDriverClassName("org.h2.Driver");  
        dataSource.setJdbcUrl("jdbc:h2:mem:userDB");  
        dataSource.setUsername("sa");  
        dataSource.setPassword("");  
        return dataSource;  
    }  
}
```

JdbcTemplate frequently used methods

Method Name	Use For	Sample
update(String sql, Object... args)	Insert Update Delete	<pre>String insertQuery = "INSERT INTO users (user_name, age) VALUES (?, ?); int rowsAffected = jdbcTemplate.update(insertQuery, "X", 27); String updateQuery= "UPDATE users SET age = ? WHERE user_id = ?"; int rowsAffected = jdbcTemplate.update(updateQuery, 29, 1);</pre>
update(String sql, PreparedStatementSetter ps)	Insert Update Delete	<pre>String insertQuery= "INSERT INTO users (user_name, age) VALUES (?, ?); jdbcTemplate.update(insertQuery, (PreparedStatement ps) -> { ps.setString(1, "X"); ps.setInt(2, 25);}); String updateQuery= "UPDATE users SET age = ? WHERE user_id = ?"; jdbcTemplate.update(updateQuery, (PreparedStatement ps) -> { ps.setString(1, 29); ps.setInt(2, 1);});</pre>
query(String sql, RowMapper<T> rowMapper)	Get multiple Rows	<pre>List<User> users = jdbcTemplate.query("SELECT * FROM users", (rs, rowNum) -> { User user = new User(); user.setId(rs.getInt("user_id")); user.setName(rs.getString("user_name")); user.setAge(rs.getInt("age")); return user;});</pre>
queryForList(String sql, Class<T> elementType)	Get Single Column of Multiple Rows	<pre>List<String> userName = jdbcTemplate.queryForList("SELECT user_name FROM users", String.class);</pre>
queryForObject(String sql, Object[] args, Class<T> requiredType)	Get single Row	<pre>User user = jdbcTemplate.queryForObject("SELECT * FROM users WHERE user_id = ?", new Object[]{1}, User.class);</pre>
queryForObject(String sql, Class<T> requiredType)	Get Single Value	<pre>int userCount = jdbcTemplate.queryForObject("SELECT COUNT(*) FROM users", Integer.class);</pre>