Java Database Connectivity

Baraneetharan Ramasamy

JDBC

JDBC stands for **Java Database Connectivity**. It's an essential part of Java programming when you need to interact with databases. Let me break it down for you:

1. **Definition of JDBC**:
    - JDBC is an **API (Application Programming Interface)** used in Java to interact with databases.
    - It allows Java applications to **connect** to and **execute queries** against various databases.
    - JDBC provides a **standard abstraction** for database communication, making it easier to work with different databases.
2. **Components of JDBC**:
    - **JDBC API**: Provides methods and interfaces for communication with the database.
        - `java.sql`: Contains interfaces and classes for data access in a relational database (Java SE).
        - `javax.sql`: Extends functionality for connection pooling (Java EE).
    - **JDBC Driver Manager**: Loads database-specific drivers to establish connections.

- **JDBC Test Suite**: Used to test JDBC drivers' operations.
- **JDBC-ODBC Bridge Drivers**: Connects JDBC to ODBC for certain databases.

3. **JDBC Driver Types**:
   - **Type 1**: JDBC-ODBC bridge (uses ODBC).
   - **Type 2**: Written in a language other than Java (e.g., C++ or C).
   - **Type 3**: Talks to a middleware server first, not the database directly.
   - **Type 4**: Pure Java-to-database implementation.

JDBC DriverManager and DataSource in Java:

JDBC DriverManager:

- The `DriverManager` class is part of the Java Database Connectivity (JDBC) API. It's used to manage a list of database drivers.
- When you want to connect to a database, you typically use `DriverManager.getConnection(url, username, password)` to obtain a connection.
- However, this approach has limitations, such as being tightly coupled to a specific database driver and lacking features like connection pooling.

JDBC DataSource:

- `DataSource` provides a more flexible and feature-rich way to manage database connections.
- It's an interface present in the `javax.sql` package.
- Key benefits of using `DataSource`:
  - **Loose coupling**: You can switch databases easily without changing your code.
  - **Connection pooling**: Reusing existing connections for better performance.
  - **Distributed systems support**: Useful for applications running in distributed environments.
- Different database vendors provide their own implementations of the `DataSource` interface. For example:
  - MySQL JDBC Driver uses `com.mysql.jdbc.jdbc2.optional.MysqlDataSource`.
  - Oracle database driver uses `oracle.jdbc.pool.OracleDataSource`.

java.sql vs javax.sql

1. **java.sql Package**:
    - **Purpose**: The `java.sql` package is a core component of Java's database connectivity APIs.
    - **Functionality**: It provides classes and interfaces for connecting to and interacting with relational databases.
    - **Key Classes**:
        - `Connection`: Represents a database connection, allowing execution of SQL queries and transactions.
        - `Statement`: Executes static SQL statements.
        - `ResultSet`: Represents a set of results returned by a SQL query.
        - `DriverManager`: Manages database drivers for establishing connections.
    - **Example**:
      Output (based on data in the "employees" table):

```
import java.sql.*;

public class JavaSqlDemo {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            Connection connection = DriverManager.getCon
nection("jdbc:mysql://localhost:3306/mydatabase", "usern
ame", "password");
            Statement statement = connection.createState
ment();
            ResultSet resultSet = statement.executeQuery
("SELECT * FROM employees");
            while (resultSet.next()) {
                System.out.println("Employee ID: " + res
ultSet.getInt("employee_id") + ", Name: " + resultSet.ge
tString("name"));
            }
            resultSet.close();
            statement.close();
            connection.close();
        } catch (ClassNotFoundException | SQLException
e) {
            e.printStackTrace();
        }
```

```
    }
}
```

```
Employee ID: 1, Name: John Smith
Employee ID: 2, Name: Emily Johnson
Employee ID: 3, Name: Michael Williams
Employee ID: 4, Name: Sarah Davis
Employee ID: 5, Name: Robert Johnson
```

2. **javax.sql Package**:
    - **Purpose**: The `javax.sql` package provides the API for server-side data source access and processing.
    - **Usage**: Primarily used in Java EE (Enterprise Edition) applications.
    - **Historical Note**: Previously, it was exclusively for JDBC extensions in Java EE, but as of JDBC 3, both `java.sql` and `javax.sql` are part of JDBC in Java SE.

`java.sql` is for general database connectivity, while `javax.sql` is more relevant in Java EE contexts.

Connection pooling in javax.sql:

Connection pooling in `javax.sql` serves a crucial purpose in database-driven applications.

1. **Reducing Connection Overhead**:
    - Opening a database connection involves several steps, such as establishing a TCP socket, reading/writing data, and closing the connection.
    - These operations are **expensive** in terms of time and resources.
    - Connection pooling allows us to **reuse existing connections**, minimizing the need to create new ones.
    - By doing so, we significantly reduce the overhead of establishing connections.
2. **Transparent Optimization**:
    - Connection pooling is **transparent** to the application code.
    - In a Java EE configuration, it's handled automatically in the middle tier.

- Applications don’t need to change their code; the pool manages connections behind the scenes.

3. **Boosting Performance**:
    - By maintaining a pool of reusable connections, we avoid the cost of frequent database trips.
    - This **improves overall performance** by minimizing connection setup and teardown.

connection pooling optimizes database access, making applications more efficient and responsive!

Connection pool libraries for Java:

When it comes to connection pooling in Java applications, there are several popular libraries you can use. Let’s explore a few of them:

1. **Apache Commons DBCP 2**:
    - **Description**: Apache Commons DBCP (Database Connection Pool) is a full-featured connection pooling framework.
    - **Features**:
        - Configurable properties for pool size, timeouts, and more.
        - Efficient management of database connections.
    - **Example**:
      Usage:

```
import org.apache.commons.dbcp2.BasicDataSource;

public class DBCPDataSource {
    private static BasicDataSource ds = new BasicDataSou
rce();

    static {
        ds.setUrl("jdbc:h2:mem:test");
        ds.setUsername("user");
        ds.setPassword("password");
        ds.setMinIdle(5);
        ds.setMaxIdle(10);
        ds.setMaxOpenPreparedStatements(100);
    }
```

```
    public static Connection getConnection() throws SQLE
xception {
        return ds.getConnection();
    }
}
```

```
Connection con = DBCPDataSource.getConnection();
```

2. **HikariCP**:
   - **Description**: HikariCP is a lightweight, high-performance JDBC connection pooling library.
   - **Features**:
     - Lightning-fast connection management.
     - Easy configuration.
   - **Example**:
     Usage:

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

public class HikariCPDataSource {
    private static HikariConfig config = new HikariConfi
g();

    // Configure properties (e.g., JDBC URL, username, p
assword)

    public static Connection getConnection() throws SQLE
xception {
        HikariDataSource ds = new HikariDataSource(confi
g);
        return ds.getConnection();
    }
}
```

```
Connection con = HikariCPDataSource.getConnection();
```

3. **C3P0**:

- **Description**: C3P0 is another mature connection pooling library.
- **Features**:
  - Supports both JDBC and JNDI data sources.
  - Fine-tuned configuration options.
- **Example**:
  Usage:

```
import com.mchange.v2.c3p0.ComboPooledDataSource;

public class C3P0DataSource {
    private static ComboPooledDataSource ds = new ComboP
ooledDataSource();

    // Configure properties (e.g., JDBC URL, username, p
assword)

    public static Connection getConnection() throws SQLE
xception {
        return ds.getConnection();
    }
}
```

```
Connection con = C3P0DataSource.getConnection();
```

Spring JdbcTemplate

Spring JdbcTemplate is a library that simplifies working with relational databases and JDBC in Spring applications. It handles low-level details such as transactions, resource management, and exception handling. The JdbcTemplate class, part of the Spring JDBC module, provides a fluent API for executing SQL queries, iterating over result sets, and catching JDBC exceptions.

https://stackoverflow.com/questions/34226406/alternative-to-static-jdbctemplate-in-spring

https://www.javatpoint.com/spring-JdbcTemplate-tutorial

Jakarta Persistence (JPA)

Jakarta Persistence (JPA), formerly known as the **Java Persistence API**, is a Jakarta EE application programming interface specification that manages relational data in enterprise Java applications. Here's a brief overview and version history:

1. **JPA 1.0**:
    - Released on **11 May 2006** as part of **Java Community Process JSR 220**.
    - Initial version with basic features for ORM (Object-Relational Mapping).
2. **JPA 2.0**:
    - Released on **10 December 2009** as part of **Java EE 6** (which requires JPA 2.0).
    - Introduced significant enhancements, including criteria queries, improved caching, and standardized metamodels.
3. **JPA 2.1**:
    - Released on **22 April 2013** as part of **Java EE 7** (which requires JPA 2.1).
    - Added features like stored procedures, schema generation, and entity graphs.
4. **JPA 2.2**:
    - Released in the **summer of 2017**.
    - Focused on minor improvements and clarifications.
5. **JPA 3.1**:
    - Released in the **spring of 2022** as part of **Jakarta EE 10**.
    - The latest version, offering further enhancements and compatibility with modern Java standards

https://en.wikipedia.org/wiki/Jakarta_Persistence

https://jakarta.ee/specifications/persistence/

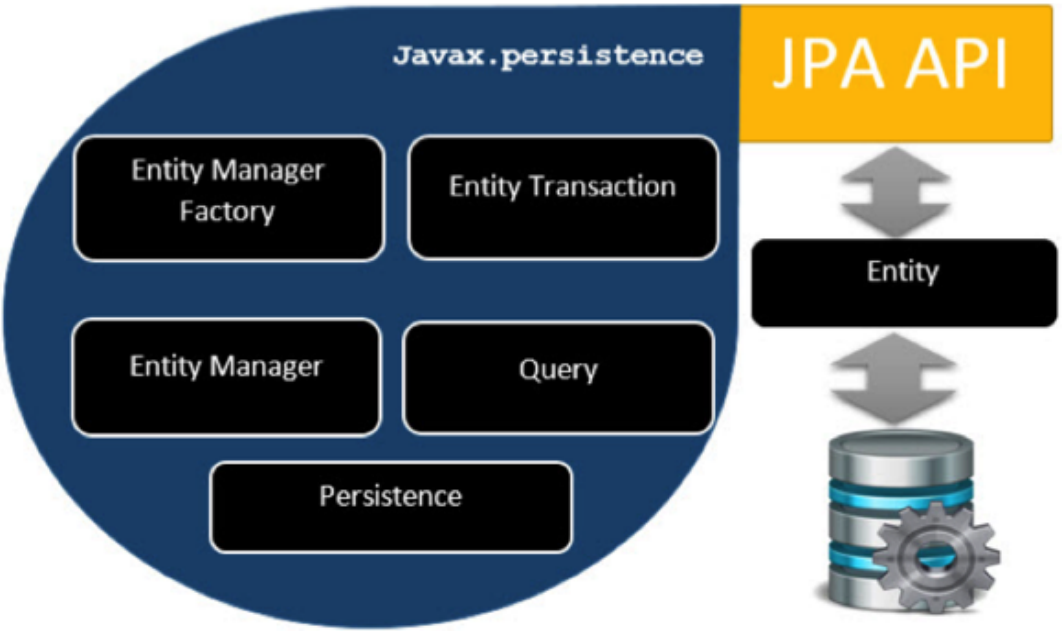
Object-Relational Mapping (ORM) in Java:

1. **What is ORM?**
    - ORM is a technique that bridges the gap between Java objects and relational databases.

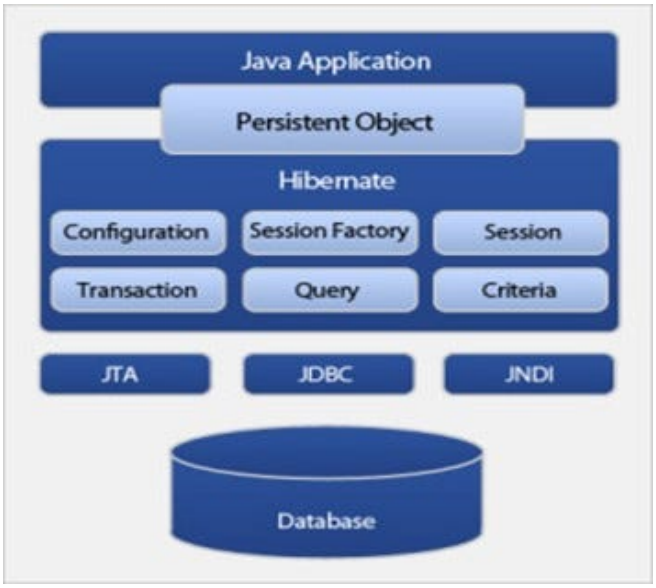
- It allows you to work with databases using object-oriented principles, avoiding complex SQL queries.
- Each model class becomes a table in the database, and each instance corresponds to a row in that table.

2. **Advantages of ORM**:
    - **Saves time and effort**: No need to write low-level database interaction code.
    - **Development pace**: Speeds up development by handling database operations.
    - **Cost reduction**: Reduces development costs and maintenance efforts.
    - **Object-oriented**: Aligns with Java's object-oriented nature.
    - **Easy transaction management**: Simplifies handling transactions.
    - **No manual database implementation**: ORM tools handle it for you.
3. **Commonly Used ORM Tools in Java**:
    - **Hibernate**: An open-source, lightweight, and powerful ORM tool. It supports JPA (Java Persistence API) and simplifies database interactions.
    - **TopLink**: Developed by Oracle, it provides flexible mapping types and independence between object model and SQL schema.
    - **OpenJPA**: Part of Apache, it can be used as a stand-alone POJO persistence layer or integrated into Java EE containers.
    - **MyBatis (formerly iBatis)**: A simpler alternative that uses SQL mappings.
    - **EclipseLink**: Another option for JPA-based ORM.

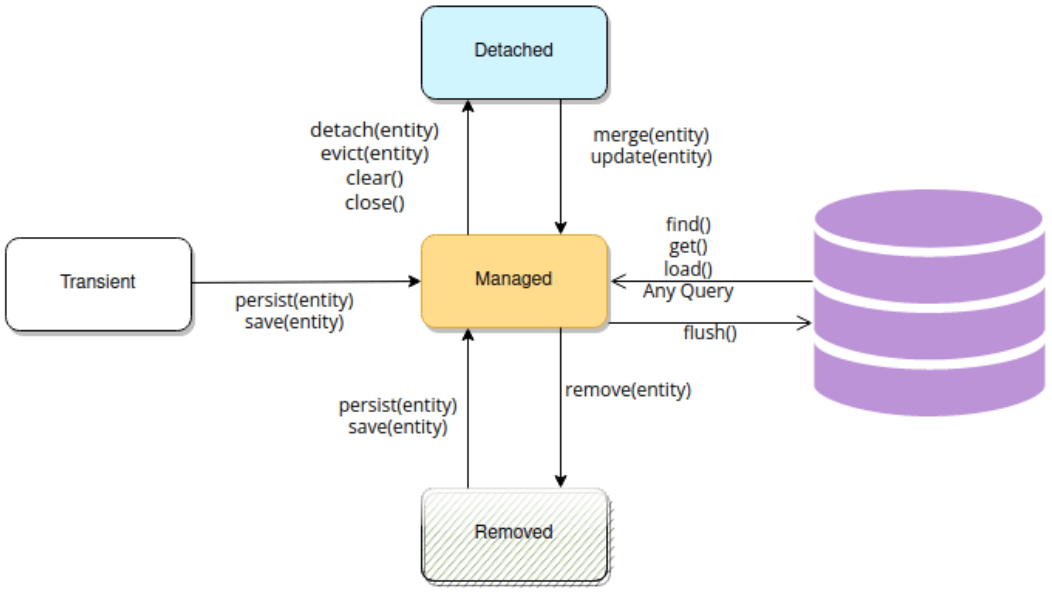
JPA - Entity Manager Factory and Entity Manager



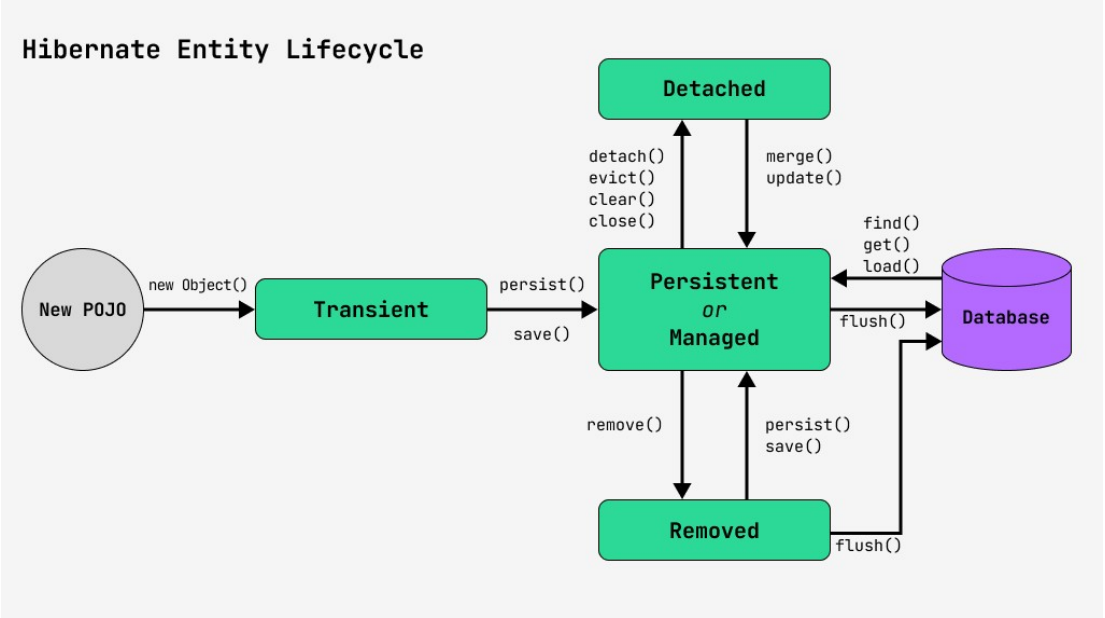
Hibernate Session Factory vs Session



JPA Entity Life Cycle



Hibernate Entity Life Cycle



SessionFactoryBean

1. **LocalSessionFactoryBean** (Hibernate-specific):
    - Provides an alternative to **LocalContainerEntityManagerFactoryBean** for common JPA purposes.

- Exposes the **Hibernate SessionFactory**, which natively implements the JPA **EntityManagerFactory** interface.
- Integrates Hibernate BeanContainer out of the box.

2. **LocalContainerEntityManagerFactoryBean**:
   - Follows JPA's standard container bootstrap contract.
   - Typically used in Spring applications.
   - Supports both local and global transactions.
   - Can link to an existing JDBC DataSource.
   - More flexible in terms of configuration within the application.

> LocalSessionFactoryBean is Hibernate specific, while LocalContainerEntityManagerFactoryBean adheres to JPA standards and is commonly used in Spring applications.

Persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
    xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
a.ee/xml/ns/persistence/persistence_3_1.xsd"
    version="3.0">

    <persistence-unit name="SimpleJPAProjectPU" transaction-t
        <!-- <provider>org.eclipse.persistence.jpa.Persistenc
        <class>com.kgisl.SampleJPAProject.Book</class>
        <properties>
            <property name="jakarta.persistence.jdbc.driver"
            <property name="jakarta.persistence.jdbc.url" val
            <property name="jakarta.persistence.jdbc.user" va
            <property name="eclipselink.logging.level.sql" va
            <!-- <property name="eclipselink.logging.paramete
            <!-- <property name="jakarta.persistence.jdbc.pas
            <property name="jakarta.persistence.schema-genera
        </properties>
    </persistence-unit>
</persistence>
```

Persistence.xml with Hibernate

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence version="3.0" xmlns="https://jakarta.ee/xml/ns/p
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence

    <persistence-unit name="StudentPU">
        <provider>org.hibernate.jpa.HibernatePersistenceProvi
        <properties>
            <property name="hibernate.connection.url" value="
            <property name="hibernate.connection.driver_class
            <property name="hibernate.connection.username" va
            <!-- <property name="hibernate.connection.passwor
            <property name="hibernate.archive.autodetection"
            <property name="hibernate.show_sql" value="true"
            <property name="hibernate.format_sql" value="true
            <property name="hbm2ddl.auto" value="update" />
        </properties>
    </persistence-unit>
</persistence>
```

Hibernate configuration file

Let's create a sample **hibernate.cfg.xml** (Hibernate configuration file) and an example **example.hbm.xml** (Hibernate mapping file) for a one-to-many relationship. We'll assume you're using MySQL as the database. Here's how you can set them up:

1. **Hibernate Configuration File (hibernate.cfg.xml)**:
    - The **hibernate.cfg.xml** contains configuration settings for Hibernate, including database connection details, dialect, and other properties.
    - Below is an example **hibernate.cfg.xml** for MySQL 8 using Hibernate 5:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibe
rnate Configuration DTD 3.0//EN" "http://www.hibernate.org/
dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- JDBC driver class for MySQL 8 -->
```

```
        <property name="connection.driver_class">com.mysql.
cj.jdbc.Driver</property>
        <!-- Database URL -->
        <property name="connection.url">jdbc:mysql://localh
ost/database</property>
        <!-- MySQL 8 dialect -->
        <property name="dialect">org.hibernate.dialect.MySQ
L8Dialect</property>
        <!-- Database credentials -->
        <property name="connection.username">root</property
>
        <property name="connection.password">password</prop
erty>
        <!-- Other optional settings -->
        <property name="connection.pool_size">3</property>
        <property name="current_session_context_class">thre
ad</property>
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>
        <property name="hbm2ddl.auto">update</property>
        <!-- Mapping class (example.hbm.xml) -->
        <!-- <mapping class="com.javacodegeeks.example.Ite
m"/> -->
    </session-factory>
</hibernate-configuration>
```

1. **Example Mapping File (example.hbm.xml)**:
    - The **example.hbm.xml** file maps Java classes to database tables. In this case, we'll create a one-to-many relationship.
    - You'll need to define your own entities (Java classes) and corresponding mappings.
    - Below is a simplified example of an **example.hbm.xml** for a one-to-many relationship:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernat
e-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="com.javacodegeeks.example.Order" table="or
ders">
```

```
        <id name="orderId" column="order_id">
            <generator class="increment"/>
        </id>
        <property name="orderName" column="order_name"/>
        <!-- One-to-many relationship with OrderItem -->
        <set name="orderItems" table="order_items" inverse
="true" cascade="all">
            <key column="order_id"/>
            <one-to-many class="com.javacodegeeks.example.O
rderItem"/>
        </set>
    </class>
    <class name="com.javacodegeeks.example.OrderItem" table
="order_items">
        <id name="itemId" column="item_id">
            <generator class="increment"/>
        </id>
        <property name="itemName" column="item_name"/>
        <!-- Many-to-one relationship with Order -->
        <many-to-one name="order" class="com.javacodegeeks.
example.Order" column="order_id"/>
    </class>
</hibernate-mapping>
```

Reference pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:s
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.kgisl.springmvchibernate</groupId>
  <artifactId>springmvchibernate</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>springmvchibernate Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <properties>
    <failOnMissingWebXml>false</failOnMissingWebXml>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>
  <dependencies>
```

```
    <!-- https://mvnrepository.com/artifact/org.springframewo
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>6.1.8</version>
</dependency>

    <!-- https://mvnrepository.com/artifact/org.springframewo
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-orm</artifactId>
  <version>6.1.8</version>
</dependency>

    <!-- https://mvnrepository.com/artifact/jakarta.servlet/j
<dependency>
  <groupId>jakarta.servlet</groupId>
  <artifactId>jakarta.servlet-api</artifactId>
  <version>6.1.0-M2</version>
  <scope>provided</scope>
</dependency>

    <!-- https://mvnrepository.com/artifact/jakarta.servlet.j
<dependency>
    <groupId>jakarta.servlet.jsp.jstl</groupId>
    <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
    <version>3.0.0</version>
</dependency>

    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>6.5.2.Final</version>
      <!-- <type>pom</type> -->
    </dependency>
    <dependency>
      <groupId>com.mysql</groupId>
      <artifactId>mysql-connector-j</artifactId>
      <version>8.2.0</version>
    </dependency>

  </dependencies>
```

```
<build>
  <finalName>springmvchibernate</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <version>3.3.2</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
    </plugin>
    <plugin>
      <groupId>org.eclipse.jetty</groupId>
      <artifactId>jetty-maven-plugin</artifactId>
      <version>11.0.11</version>
      <configuration>
        <webApp>
          <contextPath>/app</contextPath>
        </webApp>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.4.1</version>
      <configuration>
        <!-- put your configurations here -->
      </configuration>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
```

```
  </build>
</project>
```

JDBC vs. R2DBC vs. Spring JDBC vs. Spring Data JDBC

Feature	JDBC	R2DBC	Spring JDBC	Spring Data JDBC
API	Low level	Low level	High level	High level
Performance	Good	Excellent	Good	Good
Communication	Synchronous	Reactive	Synchronous	Asynchronous
Maturity	Mature	Newer	Mature	Newer
Features	Fewer features	Few features	More features	More features
Ease of use	Easy	Moderate	Easy	Easy
Support	Widespread	Growing	Widespread	Growing