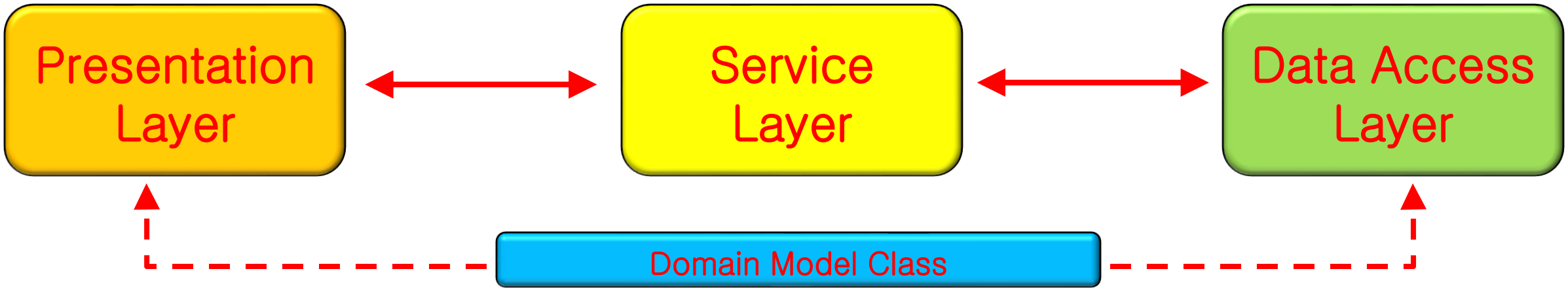
Spring JDBC

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/Spring5

Spring JDBC Architecture

- 대부분의 중/대규모 Web Application은 효율적인 개발 및 유지 보수를 위하여 계층화(Layering)하여 개발하는 것이 원칙.
- Project Architecture에서 기본적으로 가지는 계층은 ***Presentation Layer***, ***Service Layer***, ***Data Access Layer*** 3계층과 모든 계층에서 사용되는 ***Domain Model Class***로 구성.
- 각각의 계층은 계층마다 독립적으로 분리하여 구현하는 것이 가능해야 하며, 각 계층에서 담당해야 할 기능들이 있다.



Spring JDBC Architecture (Cont.)

- Presentation Layer
  - Browser상의 Web Client의 요청 및 응답을 처리
  - 상위계층(Service 계층, Data Access 계층)에서 발생하는 Exception에 대한 처리
  - 최종 UI에서 표현해야 할 Domain Model을 사용
  - 최종 UI에서 입력한 Data에 대한 유효성 검증(Validation) 기능을 제공
  - Business Logic과 최종 UI를 분리하기 위한 Controller 기능을 제공
  - **@Controller** Annotation을 사용하여 작성된 **Controller** Class가 이 계층에 속함

Spring JDBC Architecture (Cont.)

- Service Layer
  - Application Business Logic 처리와 Business와 관련된 Domain Model의 적합성 검증
  - Transaction 처리
  - Presentation Layer와 Data Access Layer 사이를 연결하는 역할로서 두 계층이 직접적으로 통신하지 않게 하여 Application의 유연성을 증가
  - 다른 계층들과 통신하기 위한 Interface 제공
  - **Service** interface와 **@Service** Annotation을 사용하여 작성된 Service 구현 Class가 이 계층에 속함.

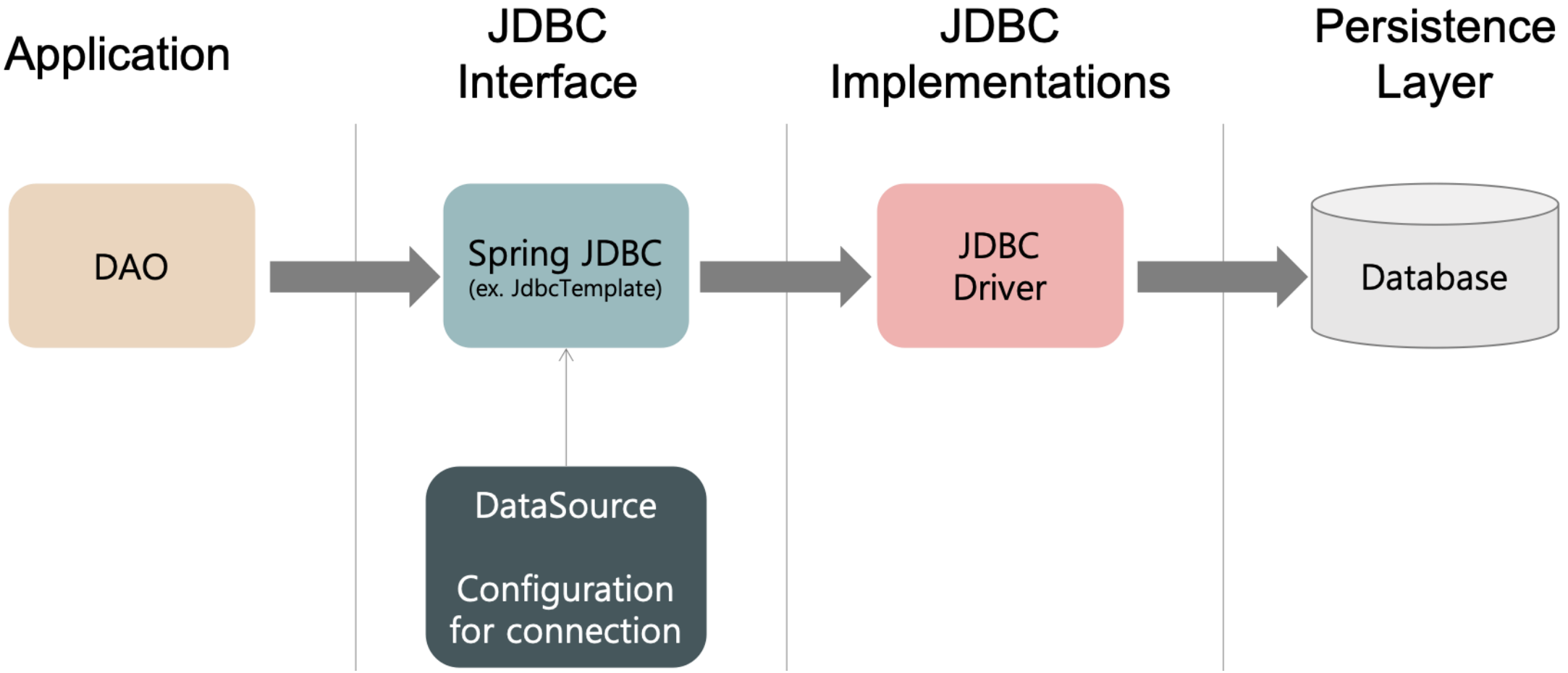
Spring JDBC Architecture (Cont.)

- Data Access Layer
  - 영구 저장소(RDBMS)의 Data를 조작하는 Data Access Logic을 객체화
  - 영구 저장소의 Data를 조회, 등록, 수정, 삭제함
  - ORM(Object Relational Mapping) Framework(MyBatis, Hibernate)를 주로 사용하는 계층
  - **Dao** interface와 **@Repository** Annotation을 사용하여 작성된 DAO 구현 Class가 이 계층에 속함.

Spring JDBC Architecture (Cont.)

- Domain Model Class
  - 관계형 Database의 Entity와 비슷한 개념을 가지는 것으로 실제 VO(Value Object) 혹은 DTO(Data Transfer Object) 객체에 해당
  - Domain Model Class는 3개의 계층 전체에 걸쳐 사용
  - **private**으로 선언된 멤버변수가 있고, 그 변수에 대한 **getter**와 **setter** Method를 가진 Class를 말함.

Spring JDBC Architecture (Cont.)

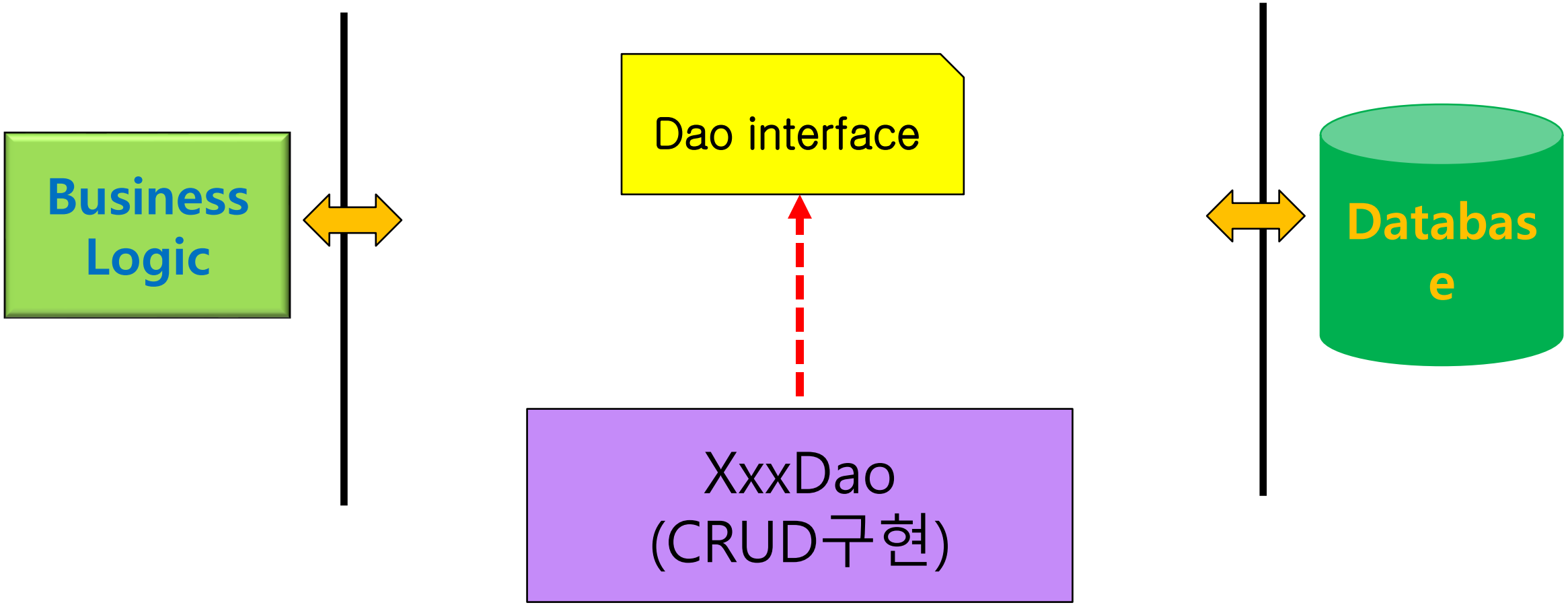


Data access 공통 개념

- DAO(Data Access Object) Pattern
  - Data Access 계층은 DAO pattern을 적용하여 Business Logic과 Data Access Logic을 분리하는 것이 원칙.
  - Database 접속과 SQL 발행 같은 Data Access 처리를 DAO라고 불리는 Object로 분리하는 Pattern.
  - Business Logic이 없거나 단순하면 DAO와 Service 계층을 통합할 수도 있지만, 의미 있는 Business Logic을 가진 Enterprise Application이라면 Data Access 계층을 DAO Pattern으로 분리해야.
  - DAO Pattern은 Service계층에 영향을 주지 않고 Data Access 기술을 변경할 수 있는 장점을 가지고 있다.

Data access 공통 개념 (Cont.)

- DAO(Data Access Object) Pattern



Data access 공통 개념 (Cont.)

- DAO(Data Access Object) Pattern
  - Dao Class에 Data Access 처리를 기술하겠지만, 그 처리를 구현하는 Java기술은 여러 가지.
  - JDBC, Hibernate, MyBatis(iBATIS), JPA, JDO 등
  - Spring에서는 새로운 Data Access 기술을 제공하는 것이 아니라 기존의 5가지 방법 기술들을 좀 더 쉽게 만드는 기능 제공.
  - JDBC는 Spring JDBC로, Hibernate는 Hibernate 연계로, JPA는 JPA 연계로, MyBatis는 MyBatis 연계로, JDO는 JDO 연계이다.
  - Spring의 기능을 이용해서 얻을 수 있는 장점.
    - Data Access 처리를 간결하게 기술할 수 있다.
    - Spring이 제공하는 범용적이고 체계적인 Data Access 예외를 이용할 수 있다.
    - Spring의 Transaction 기능을 이용할 수 있다.

Data access 공통 개념 (Cont.)

- Connection Pooling을 지원하는 **DataSource**
  - Connection Pooling은 미리 정해진 갯수만큼 DB Connection을 Pool에 준비해두고, application이 요청할 때마다 Pool에서 꺼내서 하나씩 할당해주고 다시 돌려받아서 Pool에 넣는 식의 기법.
  - 다중 사용자를 갖는 Enterprise System에서는 반드시 DB Connection Pooling 기능을 지원하는 **DataSource**를 사용해야.
  - Spring에서는 **DataSource**를 공유 가능한 Spring Bean으로 등록해서 사용할 수 있도록 해준다.

DataSource 구현 Class의 종류

- Test 환경을 위한 **DataSource**
  - **SimpleDriverDataSource**
    - **org.springframework.jdbc.datasource.SimpleDriverDataSource**
    - Spring이 제공하는 가장 단순한 **DataSource** 구현 Class
    - **getConnection()**을 호출할 때마다 매번 DB Connection을 새로 만들고 따로 Pool을 관리하지 않으므로 단순한 Test용으로만 사용해야.
  - **SingleConnectionDriverDataSource**
    - **org.springframework.jdbc.datasource.SingleConnectionDataSource**
    - 순차적으로 진행되는 통합 Test에서는 사용 가능.
    - 매번 DB Connection을 생성하지 않기 때문에 **SimpleDriverDataSource**보다 빠르게 동작한다.

DataSource 구현 Class의 종류 (Cont.)

- Open Source **DataSource**
  - **Apache Commons DBCP**
    - 가장 유명한 Open Source DB Connection Pool Library.
    - Apache의 Commons Project(http://commons.apache.org/dbcp)
  - **c3p0** JDBC/DataSource Resource Pool
    - c3p0는 JDBC 3.0 spec을 준수하는 Connection과 Statement Pool을 제공하는 Library이다.
    - http://www.mchange.com/projects/c3p0/
  - 두 가지 모두 Setter method를 제공하므로 Spring Bean으로 등록해서 사용하기 편리.

DataSource 구성하기

- Spring으로 독립형 Application을 개발하는 경우 Application Context XML File에서 Data Source를 구성할 수 있다.
- Enterprise Application을 개발하는 경우 Application Server의 JNDI에 Binding되는 Data Source를 정의한 다음 Application에서 사용할 JNDI Binding Data Source를 Application Context XML File에서 검색할 수 있다.

DataSource 구성하기 (Cont.)

```
<context:property-placeholder location="classpath:dbinfo.properties" />
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
        <property name="driverClassName" value="${db.driverClass}" />
        <property name="url" value="${db.url}" />
        <property name="username" value="${db.username}" />
        <property name="password" value="${db.password}" />
</bean>
```

DataSource 구성하기 (Cont.)

- Java EE 환경에서 **DataSource** 구성
  - Application Server로 배포되는 Enterprise Application을 개발하는 경우 Spring jee schema의 **<jndi-lookup>** 요소를 사용해 **ApplicationContext**의 Spring Bean에 JNDI Binding Data Source를 제공할 수 있다.

```
<jee:jndi-lookup jndi-name="java:comp/env/jdbc/bankAppDb“
                 id="dataSource" />
```

  - 여기서 **jndi-name** 속성에는 **javax.sql.DataSource** 객체를 JNDI에 Binding할 JNDI 이름을 지정하며 **id** 속성에는 **javax.sql.DataSource** 객체를 **ApplicationContext**에 등록할 Bean 이름을 지정한다.

Spring JDBC

- JDBC란?
  - 모든 Java의 Data Access 기술의 근간
  - Entity Class와 Annotation을 이용하는 최신 ORM 기술도 내부적으로는 DB와의 연동을 위해 JDBC를 이용
  - 안정적이고 유연한 기술이지만, Low level 기술로 인식되고 있다.
  - 간단한 SQL을 실행하는데도 중복된 Code가 반복적으로 사용되며, DB에 따라 일관성 없는 정보를 가진 채 Checked Exception으로 처리.
  - 장점
    - 대부분의 개발자가 잘 알고 있는 친숙한 Data Access 기술로 별도의 학습 없이 개발이 가능
  - 단점
    - Connection과 같은 공유 Resource를 제대로 Release 해주지 않으면 System의 자원이 바닥나는 bug 발생.

Spring JDBC (Cont.)

- Spring JDBC란?
  - JDBC의 장점과 단순성을 그대로 유지하면서도 기존 JDBC의 단점을 극복
  - 간결한 형태의 API 사용법을 제공
  - JDBC API에서 지원되지 않는 편리한 기능 제공
  - 반복적으로 해야 하는 많은 작업들을 대신 해줌.
  - Spring JDBC를 사용할 때는 실행할 SQL과 Binding 할 Parameter를 넘겨주거나, Query의 실행 결과를 어떤 객체에서 넘겨 받을지를 지정하는 것만 하면 된다.
  - Spring JDBC를 사용하려면 먼저, DB Connection을 가져오는 **DataSource**를 Bean으로 등록해야.

Spring JDBC (Cont.)

- 개발자가 JDBC방식으로 연결시 문제점들
  - 직접 개발자가 JDBC를 사용하면 Source Code가 너무 길어지고 또한 **Connection**이나 **PreparedStatement**를 얻고 나면 반드시 연결 해제를 처리해야 하지만 깜빡 잊어버리는 개발자도 있을 수 있다.
  - 그래서 연결이 해제되지 않으면 **Database**의 Resource 고갈이나 Memory 누수의 원인이 되어 최악의 경우에는 System이 정지할 가능성도 있다.
  - Data Access 오류시 오류 원인을 특정하고 싶을 때는 **SQLException**의 오류 Code를 가져와 값을 조사할 필요가 있다.
  - 심지어 오류 Code는 Database 제품마다 값이 다르므로 Database 제품이 바뀌면 다시 수정해야만 한다.
  - 또한 **SQLException**은 Compile 시 예외 처리 유무를 검사하므로 Source Code 상에서 반드시 Catch 문을 기술해야만 한다.

Spring JDBC (Cont.)

- Spring JDBC가 해주는 작업들
  - **Connection** 열기와 닫기
    - **Connection**과 관련된 모든 작업을 Spring JDBC가 필요한 시점에서 알아서 진행한다.
    - 진행 중에 예외가 발생했을 때도 열린 모든 **Connection** 객체를 닫아준다.
  - **Statement** 준비와 닫기
    - SQL 정보가 담긴 **Statement** 또는 **PreparedStatement**를 생성하고 필요한 준비 작업을 한다.
    - **Statement**도 **Connection**과 마찬가지로 사용이 끝나면 Spring JDBC가 알아서 닫아준다.
  - **Statement** 실행
    - SQL이 담긴 **Statement**를 실행
    - **Statement**의 실행결과를 다양한 형태로 가져올 수 있다.

Spring JDBC (Cont.)

- Spring JDBC가 해주는 작업들
  - **ResultSet** Loop 처리
    - **ResultSet**에 담긴 Query 실행 결과가 한 건 이상이면 **ResultSet** Loop를 만들어서 반복한다.
  - **Exception** 처리와 반환
    - JDBC 작업 중 발생하는 모든 예외는 Spring JDBC 예외 변환기가 처리한다.
    - Checked Exception인 **SQLException**을 Runtime Exception인 **DataAccessException** Type으로 변환
  - **Transaction** 처리
    - **Transaction**과 관련된 모든 작업에 대해서는 신경 쓰지 않아도 된다.

Spring JDBC (Cont.)

- Spring JDBC의 **JdbcTemplate** Class
  - Spring JDBC가 제공하는 Class 중 하나
  - JDBC의 모든 기능을 최대한 활용할 수 있는 유연성을 제공하는 Class
  - **Connection**, **Statement**, **ResultSet** 객체의 관리, JDBC 예외 포착 및 이해하기 쉬운 예외로 변환(i.e. : **IncorrectResultSetColumnCountException** 및 **CannotGetJdbcConnectionException**), 일괄 작업 수행 등을 관리.
  - Application 개발자는 **JdbcTemplate** Class로 SQL을 제공하고 SQL이 실행된 후 결과를 가져오기만 하면 된다.
  - **javax.sql.DataSource** 객체의 Wrapper 역할을 하므로 **javax.sql.DataSource** 객체를 직접 다룰 필요가 없다.

Spring JDBC (Cont.)

- Spring JDBC의 **JdbcTemplate** Class
  - 일반적으로 **JdbcTemplate** Instance는 연결을 얻을 **javax.sql.DataSource** 객체에 대한 참조를 사용해 초기화된다.

```
<bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
</bean>
<bean id="dataSource“
        class="org.apache.commons.dbcp.BasicDataSource" ...>
        ...
</bean>
```

Spring JDBC (Cont.)

- Spring JDBC의 **JdbcTemplate** Class
  - Application에서 JNDI Binding Data Source를 사용하는 경우 Jee Schema의 **<jndi-lookup>** 요소를 사용해 JNDI Binding Data Source를 Spring Container에 Bean으로 등록할 수 있다.
  - **JdbcTemplate** Class는 다음 Code와 같이 **<jndi-lookup>** 요소로 등록한 **javax.sql.DataSource bean**을 참조할 수 있다.

Spring JDBC (Cont.)

- Spring JDBC의 **JdbcTemplate** Class

```
<beans ...
      xmlns:jee="http://www.springframework.org/schema/jee"
                                          xsi:schemaLocation="...
      http://www.springframework.org/schema/jee
      http://www.springframework.org/schema/jee/spring-jee-4.0.xsd">
      <bean id="jdbcTemplate"
            class="org.springframework.jdbc.core.JdbcTemplate">
             <property name="dataSource" ref="dataSource" />
      </bean>
      <jee:jndi-lookup jndi-name="java:comp/env/jdbc/bankAppDb“
            id="dataSource" />
      ...
</beans>
```

Spring JDBC (Cont.)

- Spring JDBC의 **JdbcTemplate** Class
  - 실행, 조회, 배치의 3가지 작업 제공
    - 실행 : **INSERT**나 **UPDATE**같이 DB의 Data에 변경이 일어나는 Query를 수행하는 작업
    - 조회 : **SELECT**를 이용해 Data를 조회하는 작업
    - Batch : 여러 개의 Query를 한번에 수행해야 하는 작업
  - **JdbcTemplate** Instance는 Thread로 부터 안전하다.
  - 즉, Application의 여러 DAO에서 **JdbcTemplate** Class의 동일한 Instance를 공유함으로써 Database와 상호작용할 수 있다.

Spring JDBC (Cont.)

- Spring JDBC의 **JdbcTemplate** Class 생성
  - **JdbcTemplate**은 **DataSource**를 Parameter로 받아서 아래와 같이 생성한다.

```
JdbcTemplate template = new JdbcTemplate(dataSource);
```

  - **DataSource**는 보통 Bean으로 등록해서 사용하므로 **JdbcTemplate**이 필요한 DAO Class에서 **DataSource** Bean을 ***DI*** 받아서 **JdbcTemplate**을 생성할 때 인자로 넘겨주면 된다.
  - **JdbcTemplate**은 Multithread 환경에서도 안전하게 공유해서 쓸 수 있기 때문에 DAO Class의 Instance 변수에 저장해 두고 사용할 수 있다.

Spring JDBC (Cont.)

- Spring JDBC의 **JdbcTemplate** Class 생성
- 생성 例

```
public class UserDAOJdbc{
    JdbcTemplate jdbcTemplate;

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

Spring JDBC (Cont.)

- **JdbcTemplate**의 **update()** Method
  - **INSERT**, **UPDATE**, **DELETE**와 같은 SQL을 실행할 때 사용.

    int update(String sql, [SQL 파라미터])

  - 이 Method를 호출할 때는 SQL과 함께 Binding할 Parameter는 Object Type 가변 인자(**Object ... args**)를 사용할 수 있다.
  - 이 Method의 **return**값은 SQL 실행으로 영향받은 Record의 갯수.

Spring JDBC (Cont.)

- **JdbcTemplate**의 **update()** Method
  - 사용 例.

```
public int update(User user){
        StringBuffer updateQuery = new StringBuffer();
        updateQuery.append("UPDATE USERS SET ");
        updateQuery.append("password=?, name=? ");
        updateQuery.append("WHERE id=? ");
        int result=this.jdbcTemplate.update(updateQuery.toString(),
                                user.getName(), user.getPassword(),
                                user.getId());
        return result;
}
```

Spring JDBC (Cont.)

- **JdbcTemplate**의 **queryForObject()** Method
  - **SELECT** SQL을 실행하여 하나의 Row를 가져올 때 사용.

    <T> T queryForObject(String sql, [SQL parameter], RowMapper<T> rm)

  - SQL 실행 결과는 여러 개의 Column을 가진 하나의 Row
  - **T**는 VO 객체의 Type에 해당
  - SQL 실행 결과로 돌아온 여러 개의 Column을 가진 한 개의 Row를 **RowMapper** Callback을 이용해 VO 객체로 Mapping.

Spring JDBC (Cont.)

- **JdbcTemplate**의 **queryForObject()** Method
  - 사용 例.

```
public User findUser(String id){
        return this.jdbcTemplate.queryForObject(
          "SELECT * FROM users WHERE id=?",
                new Object [] {id},
                new RowMapper<User>(){
                        public User mapRow(ResultSet rs, int rowNum) throws SQLException{
                        User user = new User();
                        user.setId(rs.getString("id"));
                        user.setName(rs.getString("name"));
                        user.setPassword(rs.getString("password"));
                        return user;
                }
        }
}
```

Spring JDBC (Cont.)

- **JdbcTemplate**의 **query()** Method
  - **SELECT** SQL을 실행하여 여러 개의 Row를 가져올 때 사용.

<T> List<T> query(String sql, [SQL parameter], RowMapper<T> rm)

  - SQL 실행 결과로 돌아온 여러 개의 Column을 가진 여러 개의 Row를 **RowMapper** Callback을 이용해 VO 객체로 Mapping해준다.
  - 결과 값은 Mapping한 VO 객체를 포함하고 있는 **List** 형태로 받는다.
  - **List**의 각 요소가 하나의 Row에 해당한다.

Spring JDBC 환경설정

- Spring JDBC 설치
  - Maven Repository에서 ***Spring jdbc***라고 검색
  - **JdbcTemplate**를 사용하기 위해 **pom.xml**에 다음 **dependency**를 추가해야 함.

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.0.RELEASE</version>
</dependency>
```

Spring JDBC 환경설정 (Cont.)

- Oracle Jdbc Driver library 검색 및 설치
  - Oracle의 경우 어떤 Driver를 **pom.xml**에 넣어도 Error 발생.
  - 원래는 Oracle 12C인 경우 Maven Repository에서 ***oracle ojdbc8***으로, Oracle 11g인 경우는 ***oracle ojdbc6***로 검색해야 한다.
  - Maven에서 Oracle Driver를 찾지 못하는 것은 아마도 저작권 문제로 보인다.
  - 그래서 Oracle Site에서 직접 Driver를 Download 받아서 Maven을 이용해서 Maven Local Repository에 Install을 하고
  - Install된 Version으로 **pom.xml**에 Dependency 설정을 해야 한다.

Spring JDBC 환경설정 (Cont.)

- Oracle Jdbc Driver library 검색 및 설치
  - ojdbc8.jar
  - Download
  - https://download.oracle.com/otn/utilities_drivers/jdbc/122010/ojdbc8.jar
  - Maven installer를 이용해서 Maven repository에 설치한다.

mvn install:install-file -Dfile="file이름(위치까지)" -DgroupId=그룹아이디 -DartifactId=파일이름 -Dversion=버전 -Dpackaging=jar

> mvn install:install-file -Dfile="C:₩Downloads₩ojdbc8.jar" -DgroupId=com.oracle -DartifactId=ojdbc8 -Dversion=12.2 -Dpackaging=jar

```
C:\Windows\system32>mvn install:install-file -Dfile="C:\Downloads\ojdb
om.oracle -DartifactId=ojdbc8 -Dversion=12.2 -Dpackaging=jar
```

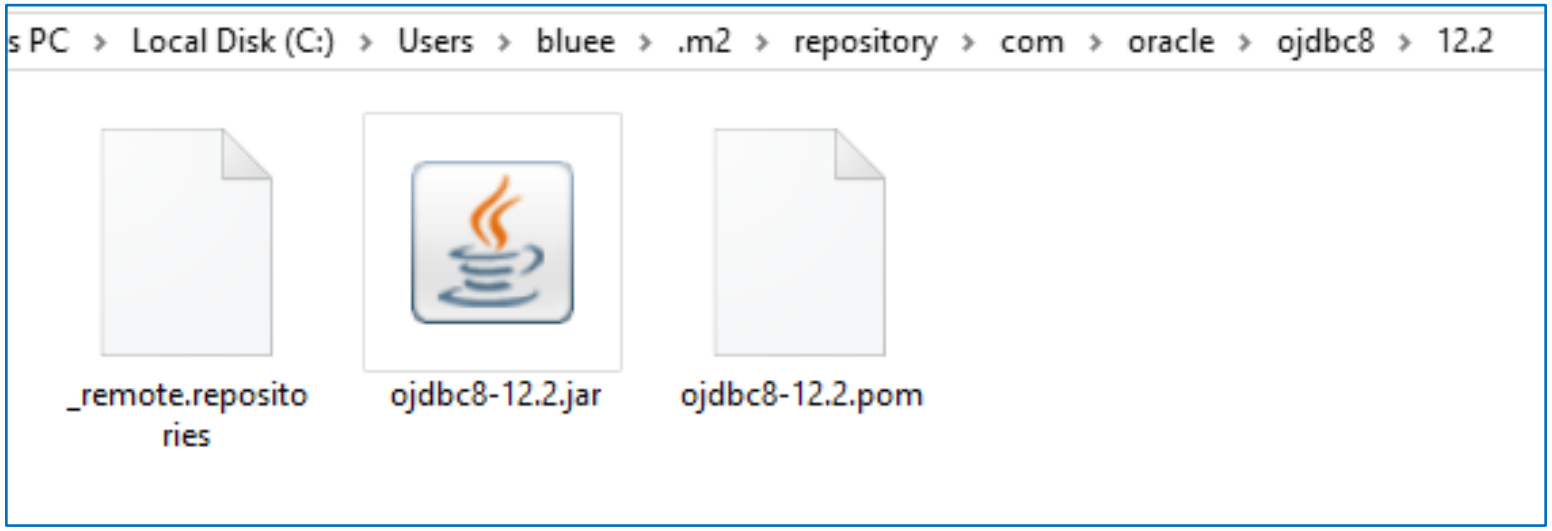
Spring JDBC 환경설정 (Cont.)

- Oracle Jdbc Driver library 검색 및 설치

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< org.apache.maven:standalone-pom >------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] --------------------------------[ pom ]---------------------------------
[INFO]
[INFO] --- maven-install-plugin:2.4:install-file (default-cli) @ standalone-pom ---
[INFO] Installing C:\Downloads\ojdbc8.jar to C:\Users\bluee\.m2\repository\com\oracle\ojdb
c8\12.2\ojdbc8-12.2.jar
[INFO] Installing C:\Users\bluee\AppData\Local\Temp\mvninstall4483871289242274317.pom to C
:\Users\bluee\.m2\repository\com\oracle\ojdbc8\12.2\ojdbc8-12.2.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.248 s
[INFO] Finished at: 2019-10-30T11:47:16+09:00
[INFO] ------------------------------------------------------------------------
```

Spring JDBC 환경설정 (Cont.)

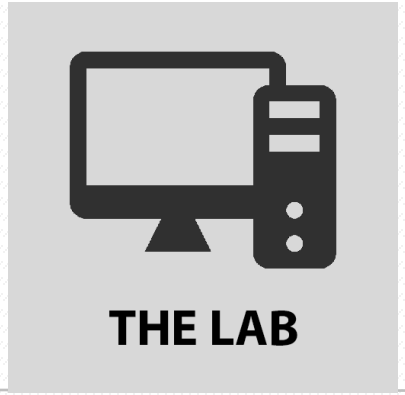
- Oracle Jdbc Driver library 검색 및 설치
  - Install 명령을 실행하면 Maven depository에 해당 driver가 설치된다.
  - 앞 Slide에서는 C:₩Users₩계정₩.m2₩repository₩com₩oracle₩ojdbc8₩12.2 에 설치된 것이다.
  - 해당 Directory로 이동하면 jar File과 pom File이 있다.
  - pom file의 **groupId**, **artifactId**, **version**을 pom.xml에 dependency로 설정하면 된다.



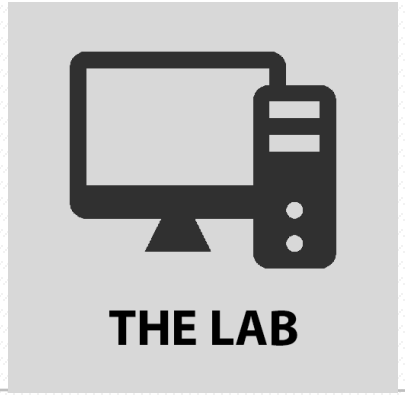
Spring JDBC 환경설정 (Cont.)

- Oracle Jdbc Driver library 검색 및 설치
  - pom.xml에 dependency 설정.
  - pom.xml에 추가한 후, pom.xml **clean** 한 다음, 다시 **install** 한다.

```
<dependency>
        <groupId>com.oracle</groupId>
        <artifactId>ojdbc8</artifactId>
        <version>12.2</version>
</dependency>
```

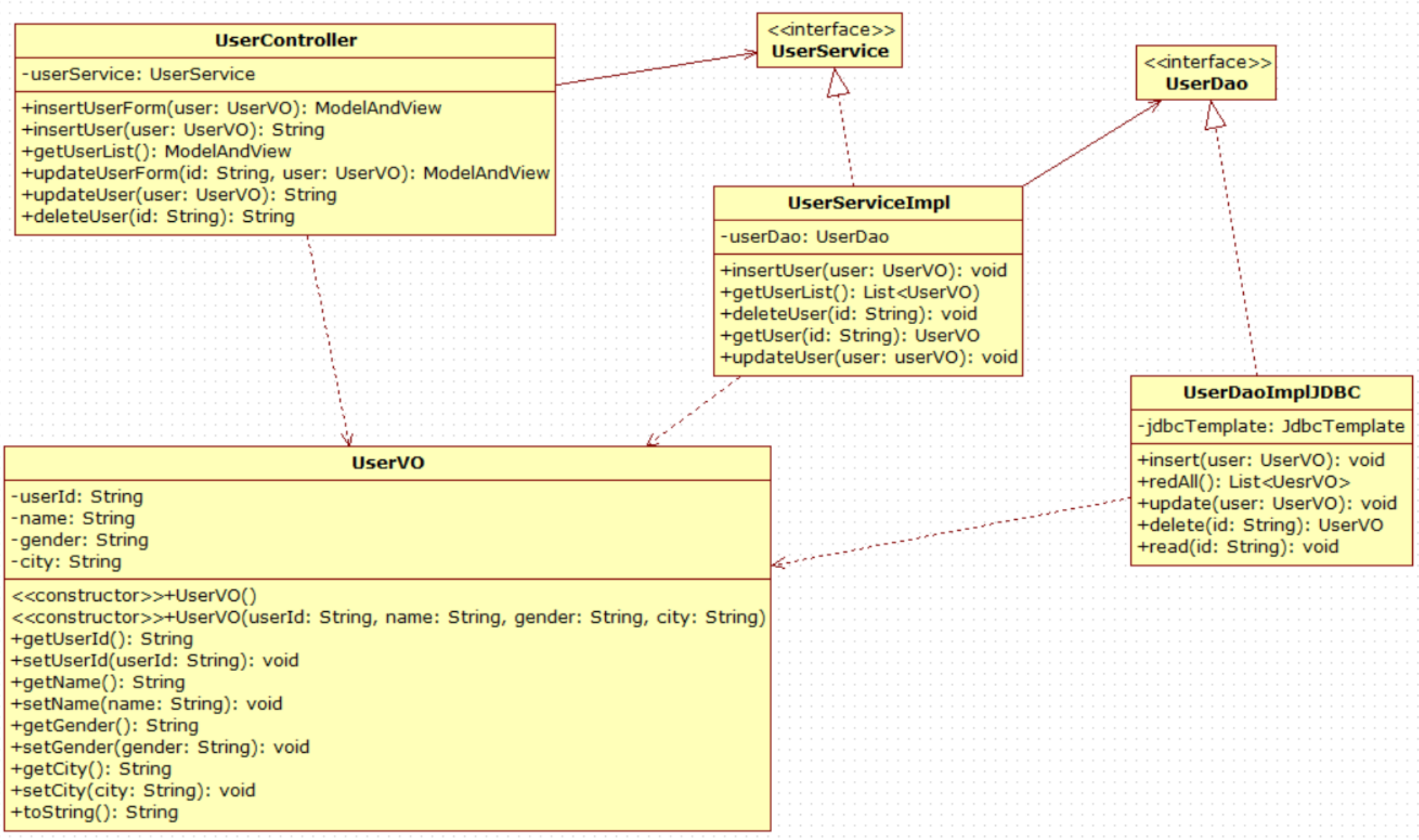


Task 1. Spring Jdbc Demo



Task 2. Membership Project

Membership Project Class Diagram



Membership Project 각 Class의 역할

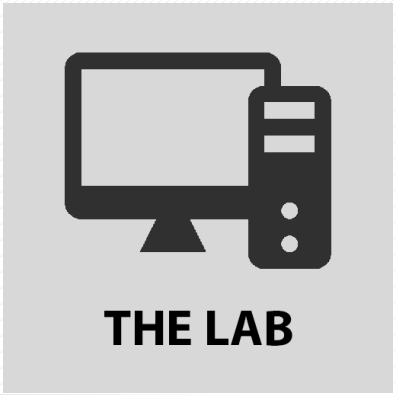
- Presentation Layer
- UserController : Class
  - UI계층과 Service 계층을 연결하는 역할을 하는 Class
  - JSP에서 **UserController**를 통해서 Service 계층의 **UserService**를 사용하게 된다.
  - Service 계층의 **UserService** Interface를 구현하나 객체를 IoC Container가 주입해 준다.

Membership Project 각 Class의 역할 (Cont.)

- Service Layer
- **UserService** : Interface
  - Service 계층에 속한 상위 Interface
- **UserServiceImpl** : Class
  - **UserService** Interface를 구현한 Class
  - 복잡한 업무 Logic이 있을 경우에는 이 Class에서 업무 Logic을 구현하면 된다.
  - Data Access 계층의 **UserDao** Interface를 구현한 객체를 IoC Container가 주입해준다.

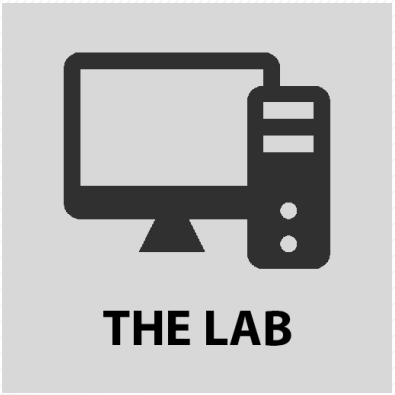
Membership Project 각 Class의 역할 (Cont.)

- Data Access Layer
- **UserDao** : Interface
  - Data access 계층에 속한 상위 Interface
- **UserDaoImplJDBC** : Class - Spring JDBC 구현
  - **UserDao** Interface를 구현한 Class로 이 Class에서는 Data Access Logic을 구현하면 된다.
  - Spring JDBC를 사용하는 경우에는 **DataSource**를 IoC Container가 주입해준다.
  - **MyBatis**를 사용하는 경우에는 **SqlSession**을 IoC Container가 주입해준다.



Task 3. Membership Project Using JdbcTemplate

Task 4. Membership Project Using iBatis



Task 5. Membership Project Using MyBatis

Spring JdbcTemplate

Spring JdbcTemplate

- https://www.javatpoint.com/spring-JdbcTemplate-tutorial
- https://docs.spring.io/spring/docs/5.2.0.RELEASE/javadoc-api/
- Spring **JdbcTemplate** is
  - A powerful mechanism to connect to the database and execute SQL queries.
  - **org.springframework.jdbc.core.JdbcTemplate**
  - Internally uses JDBC api, but eliminates a lot of problems of JDBC API.

public class JdbcTemplate extends JdbcAccessor implements JdbcOperations

Spring JdbcTemplate (Cont.)

- Problems of JDBC API
  - We need ***to write a lot of code*** before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
  - We need ***to perform exception handling code*** on the database logic.
  - We need ***to handle transaction***.
  - ***Repetition of all these codes*** from one to another database logic is a time consuming task.
- Advantage of Spring **JdbcTemplate**
  - ***Eliminates*** all the above mentioned problems of JDBC API.
  - Provides methods to write the queries directly, so it ***saves*** a lot of work and time.

Spring JdbcTemplate (Cont.)

- Spring Jdbc Approaches
  - Spring framework provides following approaches for JDBC database access:
  - **JdbcTemplate**
    - Is the classic Spring JDBC approach and the most popular.
    - This ***lowest level*** approach and all others use a **JdbcTemplate** under the covers.
  - **NamedParameterJdbcTemplate**
    - Wraps a **JdbcTemplate** to provide named parameters instead of the traditional JDBC **?** placeholders.
    - This approach provides better documentation and ease of use when you have multiple parameters for an SQL statement.
  - **SimpleJdbcTemplate**

Spring JdbcTemplate (Cont.)

- Spring Jdbc Approaches
  - **SimpleJdbcInsert** and **SimpleJdbcCall**
    - Optimize database metadata to limit the amount of necessary configuration.
    - This approach simplifies coding so that you only need to provide the name of the table or procedure and provide a map of parameters matching the column names.
    - This only works if the database provides adequate metadata.
    - If the database doesn't provide this metadata, you will have to provide explicit configuration of the parameters.
  - RDBMS Objects including **MappingSqlQuery**, **SqlUpdate** and **StoredProcedure** requires you to create reusable and thread-safe objects during initialization of your data access layer.
  - This approach is modeled after JDO Query wherein you define your query string, declare parameters, and compile the query.
  - Once you do that, execute methods can be called multiple times with various parameter values passed in.

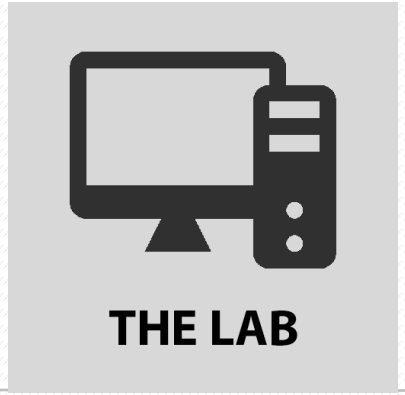
Spring JdbcTemplate (Cont.)

- **JdbcTemplate** class
  - Is the central class in the Spring JDBC support classes.
  - Takes care of creation and release of resources such as creating and closing of connection object etc.
  - So it will not lead to any problem if you forget to close the connection.
  - It handles the exception and provides the informative exception messages by the help of exception classes defined in the **org.springframework.dao** package.
  - We can perform all the database operations by the help of **JdbcTemplate** class such as insertion, updation, deletion and retrieval of the data from the database.

Spring JdbcTemplate (Cont.)

- **JdbcTemplate** class's Methods
  - **public int update(String query)**
    - Is used to insert, update and delete records.
  - **public int update(String query, Object ... args)**
    - Is used to insert, update and delete records using **PreparedStatement** using given arguments.
  - **public void execute(String query)**
    - Is used to execute DDL query.
  - **public T execute(String sql, PreparedStatementCallback action)**
    - Executes the query by using **PreparedStatement** callback.
  - **public T query(String sql, ResultSetExtractor rse)**
    - Is used to fetch records using **ResultSetExtractor**.
  - **public List query(String sql, RowMapper rse)**
    - Is used to fetch records using **RowMapper**.

Task 6. Example of Spring JdbcTemplate



Task 7. Example of PreparedStatement in Spring JdbcTemplate

PreparedStatement in Spring JdbcTemplate

- We can execute parameterized query using Spring **JdbcTemplate** by the help of **execute()** method of **JdbcTemplate** class.
- To use parameterized query, we pass the instance of **PreparedStatementCallback** in the **execute()** method.
- Syntax of execute method to use parameterized query

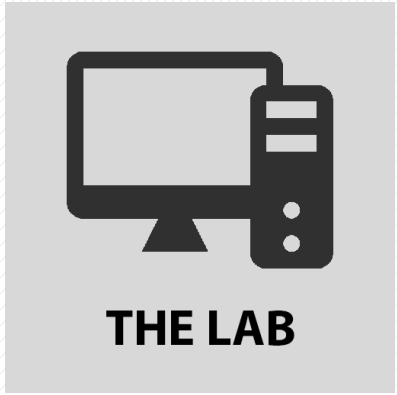
  public T execute(String sql, PreparedStatementCallback<T>);

- **PreparedStatementCallback** interface
  - It processes the input parameters and output results.
  - In such case, you don't need to care about single and double quotes.

PreparedStatement in Spring JdbcTemplate (Cont.)

- Method of **PreparedStatementCallback** interface
  - It has only one method **doInPreparedStatement**.
  - Syntax of the method is given below:

```
public T doInPreparedStatement(PreparedStatement ps)
        throws SQLException, DataAccessException
```



Task 8. ResultSetExtractor Example | Fetching Records by Spring JdbcTemplate

ResultSetExtractor Example | Fetching Records by Spring JdbcTemplate

- We can easily fetch the records from the database using **query()** me thod of **JdbcTemplate** class where we need to pass the instance of **R esultSetExtractor**.
- Syntax of query method using **ResultSetExtractor**

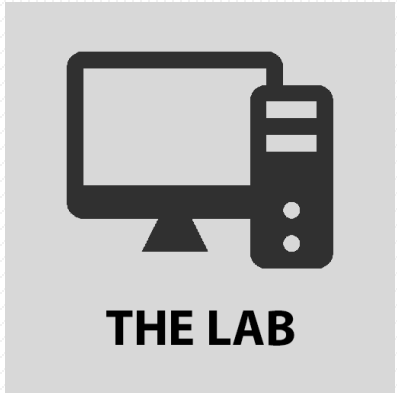
  public T query(String sql,ResultSetExtractor<T> rse)

- **ResultSetExtractor** Interface
  - **ResultSetExtractor** interface can be used to fetch records from the database.
  - It accepts a **ResultSet** and returns the list.

ResultSetExtractor Example | Fetching Records by Spring JdbcTemplate (Cont.)

- Method of **ResultSetExtractor** interface
  - It defines only one method **extractData** that accepts **ResultSet** instance as a parameter.
  - Syntax of the method is given below:

```
public T extractData(ResultSet rs)throws SQLException, DataAccessException
```



Task 9. RowMapper Example | Fetching records by Spring JdbcTemplate

RowMapper Example | Fetching records by Spring JdbcTemplate

- Like **ResultSetExtractor**, we can use **RowMapper** interface to fetch the records from the database using **query()** method of **JdbcTemplate** class .
- In the execute of we need to pass the instance of **RowMapper** now.
- Syntax of query method using **RowMapper**

  public T query(String sql, RowMapper<T> rm)

-
- **RowMapper** Interface
  - **RowMapper** interface allows to map a row of the relations with the instance of user-defined class.
  - It iterates the **ResultSet** internally and adds it into the collection.
  - So we don't need to write a lot of code to fetch the records as **ResultSetExtractor**.

RowMapper Example | Fetching records by Spring JdbcTemplate (Cont.)

- Advantage of **RowMapper** over **ResultSetExtractor**
  - **RowMapper** saves a lot of code because it internally adds the data of **ResultSet** into the collection.
- Method of **RowMapper** interface
  - It defines only one method **mapRow** that accepts **ResultSet** instance and int as the parameter list.
  - Syntax of the method is given below:

```
public T mapRow(ResultSet rs, int rowNumber)throws SQLException
```

Task 10. Spring NamedParameterJdbcTemplate Example

Spring NamedParameterJdbcTemplate Example

- Spring provides another way to insert data by named parameter.
- In such way, we use names instead of **?**(question mark).
- So it is better to remember the data for the column.
- Simple example of named parameter query

  insert into employee values (:id, :name, :salary)
- Method of **NamedParameterJdbcTemplate** class
  - In this example, we are going to call only the execute method of **NamedParameterJdbcTemplate** class.
  - Syntax of the method is as follows:

pubic T execute(String sql,Map map, PreparedStatementCallback psc)