Spring Database

Spring JDBC

- Spring JDBC
  - ✓ 스프링은 데이터베이스 연동을 위한 템플릿 클래스를 제공함으로써 중복된 코드를 입력해야 하는 부분을 줄일 수 있음
  - ✓ 스프링의 데이터베이스 지원
    - 템플릿 클래스를 통한 데이터 접근 지원
    - 의미 있는 예외 클래스 제공
      - 일반 JDBC 프로그래밍의 예외는 SQLException 하나로 처리되기 때문에 Connection을 구하는 과정에서 예외가 발생한 것인지 PreparedStatement를 생성하는 과정에서 예외가 발생한 것인지 SQL을 실행하는 과정에서 예외가 발생했는지 아는 것이 쉽지 않음
      - 스프링은 데이터베이스와 관련된 예외를 처리하기 위해서 SQLException이 발생하면 스프링이 제공하는 예외 클래스 중에서 알맞은 예외 클래스로 변환해서 발생
      - SQL 쿼리를 수행하다가 예외가 발생하면 BadSqlGrammarException이 발생
      - 스프링이 제공하는 Exception은 DataAccessException의 하위 클래스
      - 스프링이 발생시키는 Exception은 대부분 RuntimeException이어서 예외 처리하는 코드를 입력하지 않아도 에러가 발생하지 않음
    - 간단한 트랜잭션 처리
    - MyBatis와 같은 SQL Mapper 프레임워크 지원
    - Hibernate 와 같은 ORM 프레임워크를 직접적으로 지원

Spring JDBC

- Spring JDBC의 예외 클래스
  - ✓ DataAccessException의 하위 클래스: NonTransientDataAccessException, TransientDataAccessException(반복적인 쿼리의 실패로 인한 예외), RecoverableDataAccessException(특정 시간 동안 쿼리를 수행하지 않는 경우 – 주기적으로 쿼리를 던져서 해결)
  - ✓ NonTransientDataAccessException의 하위 클래스: DataIntegrityViolationException, DuplicateKeyException, DataRetrievalFailureException(데이터 검색 예외), PermissionDeniedDataAccessException(접근거부), InvalidDataAccessResourceUsageException(잘못된 SQL 사용), BadSqlGrammarException, TypeMismatchDataAccessException
  - ✓ TransientDataAccessException의 하위 클래스: TransientDataAccessResourceException, ConcurrencyFailureException, OptimisticLockingFailureException, PessimisticLockingFailureException

Spring JDBC

- 데이터베이스 연결 테스트
  - ✓ Java Simple Maven 프로젝트를 생성 - springdatabase
  - ✓ pom.xml 파일에서 properties 태그 변경
    - Spring Version을 5.0.7 로 변경

```
<properties>
    <java.version>1.8</java.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
   8</project.reporting.outputEncoding>
    <!-- Spring -->
    <spring-framework.version>5.0.7.RELEASE</spring-framework.version>
    <!-- Hibernate / JPA -->
    <hibernate.version>4.2.1.Final</hibernate.version>

    <!-- Logging -->
    <logback.version>1.0.13</logback.version>
    <slf4j.version>1.7.5</slf4j.version>

    <!-- Test -->
    <junit.version>4.12</junit.version>
</properties>
```

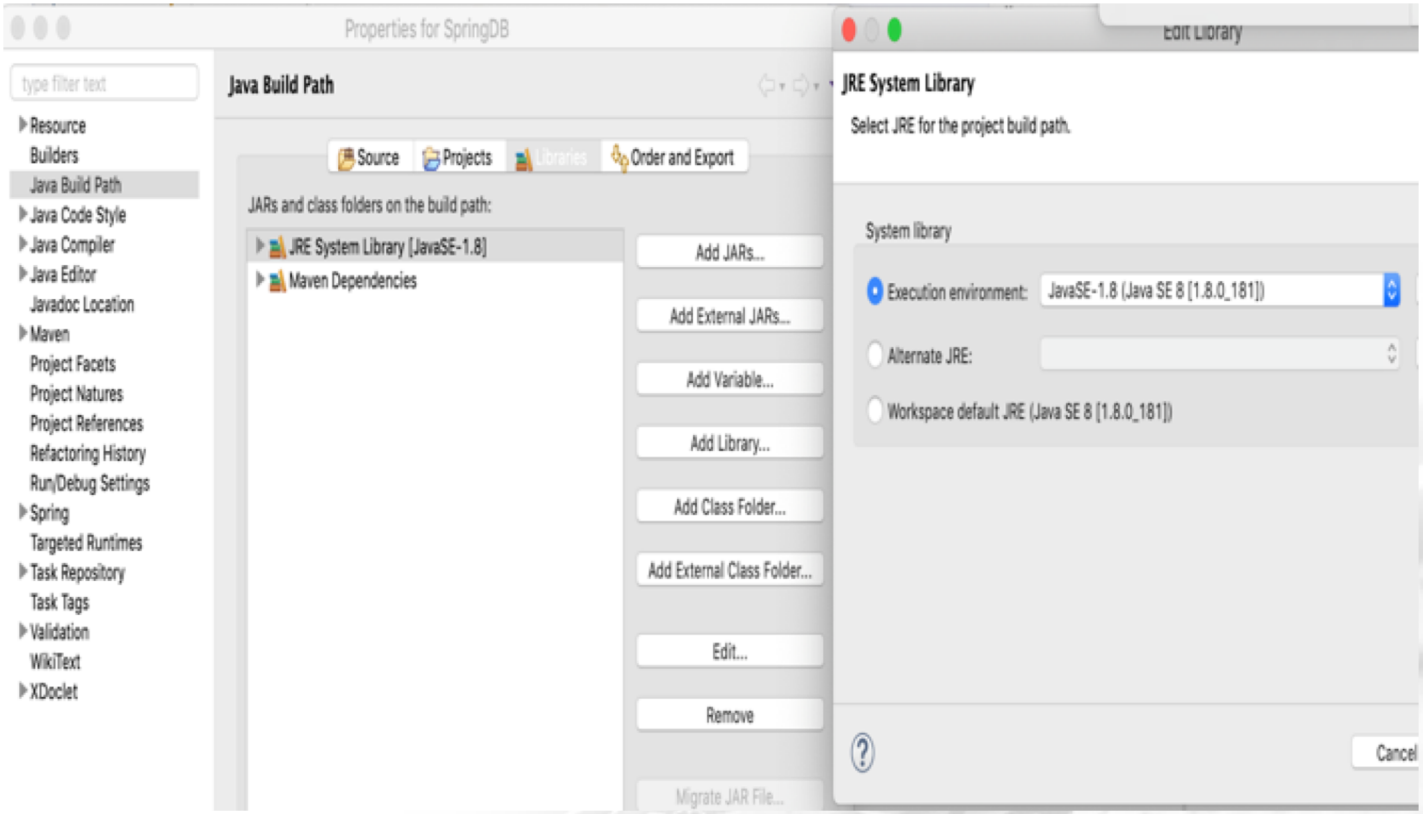
Spring JDBC

- 데이터베이스 연결 테스트
  - ✓ Java Simple Maven 프로젝트를 생성 - springdatabase
  - ✓ pom.xml 파일에서 properties 태그 변경
    - 의존성 수정

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${spring-framework.version}</version>
</dependency>
```

Spring JDBC

- 데이터베이스 연결 테스트
  - ✓ JRE System Library를 선택하고 [Build Path] – [Configure Build Path]를 선택하고 Libraries 탭에서 JRE System Library를 1.8로 변경



Spring JDBC

- 데이터베이스 연결 테스트
  - ✓ MariaDB
    - pom.xml 파일에 Maria DB 라이브러리 종속성 추가

```
<dependency>
    <groupId>org.mariadb.jdbc</groupId>
    <artifactId>mariadb-java-client</artifactId>
    <version>3.1.0</version>
</dependency>
```

Spring JDBC

- 데이터베이스 연결 테스트
  - ✓ MariaDB
    - src/test/java 디렉토리에 ConnectionTest라는 클래스를 만들고 메서드 작성하고 Junit으로 테스트

```
public class ConnectionTest {
    //MariaDB 설정
    private static final String DRIVER = "org.mariadb.jdbc.Driver";
    private static final String URL = "jdbc:mariadb://localhost:3306/adam";
    private static final String USER = "root";
    private static final String PW = "wnddkd";

    @Test
    public void testConnection() throws Exception{
      Class.forName(DRIVER);
      try(Connection con = DriverManager.getConnection(URL, USER, PW)){
                System.out.println(con);
      }catch(Exception e){
                e.printStackTrace();
      }
    }
}
```

Spring JDBC

- 데이터베이스 연결 테스트
  - ✓ MySQL
    - pom.xml 파일에 MySQL 라이브러리 종속성 추가

```
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.27</version>
</dependency>
```

Spring JDBC

- 데이터베이스 연결 테스트
  - ✓ MySQL
    - ConnectionTest 클래스를 수정하고 테스트

```
public class ConnectionTest {
    //MySQL 설정
    private static final String DRIVER = "com.mysql.cj.jdbc.Driver";
    private static final String URL = "jdbc:mysql://localhost:3306/adam";
    private static final String USER = "root";
    private static final String PW = "wnddkd";

    @Test
    public void testConnection() throws Exception{
      Class.forName(DRIVER);
      try(Connection con =
                                  DriverManager.getConnection(URL, USER, PW)){
                System.out.println(con);
      }catch(Exception e){
                e.printStackTrace();
      }
    }
}
```

Spring JDBC

- 데이터베이스 연결 테스트
  - ✓ Oracle
    - pom.xml 파일에 오라클 라이브러리 종속성 추가
      - 저장소 추가

```
<repositories>
    <repository>
            <id>oracle</id>
            <name>ORACLE JDBC Repository</name>
            <url>http://maven.jahia.org/maven2</url>
    </repository>
</repositories>
```

      - 종속성 추가 – dependencies

```
<dependency>
            <groupId>com.oracle</groupId>
            <artifactId>ojdbc7</artifactId>
            <version>12.1.0.2</version>
</dependency>
```

Spring JDBC

- 데이터베이스 연결 테스트
  - ✓ Oracle
    - ConnectionTest라는 클래스를 만들고 메서드를 작성하고 테스트

```
public class ConnectionTest {
    private static final String DRIVER = "oracle.jdbc.driver.OracleDriver";
    private static final String URL = "jdbc:oracle:thin:@localhost:1521:xe";
    private static final String USER = "system";
    private static final String PW = "oracle";
    @Test
    public void testConnection() throws Exception{
      Class.forName(DRIVER);
      try(Connection con =
                                  DriverManager.getConnection(URL, USER, PW)){
                System.out.println(con);
      }catch(Exception e){
                e.printStackTrace();
      }
    }
}
```

Spring JDBC

- DAO 패턴
  - ✓ 데이터 엑세스 계층은 DAO 패턴의 형태로 분리하는 것을 권장
  - ✓ 기술이나 환경에 종속되지 않는 POJO로 개발하는 것을 권장
  - ✓ DI로 생성하는 것을 권장
  - ✓ 데이터 엑세스 도중에 발생하는 예외는 대부분 복구할 수 없는 예외로 외부로 던져질 때는 런타임 예외이어야 하며 선언부에서 throws SQLException 과 같은 형태나 Exception 같은 형태를 만드는 것은 별로 바람직하지 못함

Spring JDBC

- DataSource
  - ✓ jdbc를 이용해 데이터베이스를 사용하려면 Connection 타입의 DB 연결 오브젝트가 필요
  - ✓ Spring 에서는 Connection을 만들 때 DataSource를 이용하도록 강제
  - ✓ 스프링에서는 Connection을 만들기 위한 DataSource를 하나의 독립된 bean으로 생성하는 것을 권장
  - ✓ DriverManagerDataSource나 BasicDataSource 클래스를 이용
  - ✓ DataSource를 생성하는 방법
    - Connection Pool을 이용하는 방법
    - JNDI를 이용하는 방법
    - DriverManager를 이용하는 방법

Spring JDBC

- DataSource
  - ✓ SpringJDBC의 DriverManager를 이용한 DataSource 설정
    - org.springframework.jdbc.datasource.DriverManagerDataSource 클래스
    - Property
      - driverClassName
      - url
      - username
      - password

Spring JDBC

- DataSource
  - ✓ Apache의 DBCP(Jakarta Commons Database Connection Pool) API 커넥션을 이용한 설정
    - org.apache.commons.dbcp2.BasicDataSource 클래스 이용
    - 프로퍼티: http://commons.apache.org/proper/commons-dbcp/configuration.html
      - initialSize: 초기 풀에 생성되는 커넥션 개수
      - maxActive: 커넥션 풀이 제공하는 최대 커넥션 개수
      - maxIdle: 사용되지 않고 풀에 저장될 수 있는 최대 커넥션 개수
      - minIdle: 사용되지 않고 풀에 저장될 수 있는 최소 커넥션 개수
      - maxWait: 풀에 커넥션이 존재하지 않을 때 대기하는 시간으로 1/1000초 단위로 설정하면 -1이면 무한 대기
      - minEvictableIdleTimeMillis: 이 시간 동안 비 활성화 상태이면 커넥션 추출
      - timeBetweenEvictionRunsMillis: 커넥션 추출 주기

Spring JDBC

- DataSource
  - ✓ HikariDataSource API 커넥션을 이용한 설정
    - 의존성 설정

```
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>3.4.1</version>
</dependency>
```

    - xml 설정

```
<bean id="hikariConfig" class="com.zaxxer.hikari.HikariConfig">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>

    <property name="jdbcUrl" value="jdbc:oracle:thin:@ip:port:SID"/>
      <property name="username" value="id"/>
    <property name="password" value="password"/>
</bean>
<bean id="dataSource" class="com.zaxxer.hikari.HikariDataSource" destroy-
    method="close">
    <constructor-arg ref="hikariConfig" />
</bean>
```

Spring JDBC

- DataSource
  - ✓ HikariDataSource API 커넥션을 이용한 설정
    - application.properties

```
spring.datasource.type=com.zaxxer.hikari.HikariDataSource
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@ip:port:SID
spring.datasource.username=id
spring.datasource.password=password
spring.datasource.hikari.idle-timeout=10000
```

Spring JDBC

- DataSource
  - ✓ HikariDataSource API 커넥션을 이용한 설정
    - Java Config

```
@Bean(destroyMethod = "close")
public DataSource dataSource(){
    HikariConfig hikariConfig = new HikariConfig();
      hikariConfig.setDriverClassName("oracle.jdbc.driver.OracleDriver");
      hikariConfig.setJdbcUrl("jdbc:oracle:thin:@ip:port:SID");
      hikariConfig.setUsername("id");
    hikariConfig.setPassword("password");
      //hikariConfig.setMaximumPoolSize(5);
      //hikariConfig.setConnectionTestQuery("SELECT 1");
      //hikariConfig.setPoolName("springHikariCP");
    HikariDataSource dataSource = new HikariDataSource(hikariConfig);

    return dataSource;
}
```

Spring JDBC

- DataSource
  - ✓ Spring JDBC API 커넥션을 이용한 설정
    - ● 프로젝트의 pom.xml 파일에 Spring-JDBC 의존성 설정

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring-framework.version}</version>
</dependency>
```

Spring JDBC

- DataSource
  - ✓ Spring JDBC API 커넥션을 이용한 설정
    - src/main/resources 디렉토리에 Spring Bean Configuration(applicationContext.xml) 파일을 생성하고 context Namespace 추가

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">


</beans>
```

Spring JDBC

- DataSource
  - ✓ Spring JDBC API 커넥션을 이용한 설정
    - applicationContext.xml 파일에 Maria DB DataSource 빈 생성 코드 추가

```
<!-- Maria DB DataSource -->
<bean id="mariaDBDataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName">
                <value>org.mariadb.jdbc.Driver</value>
        </property>
        <property name="url">
                <value>jdbc:mariadb://localhost:3306/adam</value>
        </property>
        <property name="username">
                <value>root</value>
        </property>
        <property name="password">
                <value>wnddkd</value>
        </property>
</bean>
```

Spring JDBC

- DataSource
  - ✓ Spring JDBC API 커넥션을 이용한 설정
    - ConnectionTest 클래스에 메서드를 추가하고 JUnit으로 테스트

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class ConnectionTest {
    @Autowired
    private DataSource ds;

    @Test
    public void testDataSourceConection()throws Exception{
      try(Connection con = ds.getConnection()){
                System.out.println(con);
      }catch(Exception e){
                e.printStackTrace();
      }
    }
}
```

Spring JDBC

- DataSource
  - ✓ Spring JDBC API 커넥션을 이용한 설정
    - ConnectionTest 클래스에 메서드를 추가하고 JUnit으로 테스트

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class ConnectionTest {
    @Autowired
    @Qualifier("mariaDBDataSource")
    private DataSource ds;

    @Test
    public void testDataSourceConection()throws Exception{
      try(Connection con = ds.getConnection()){
                System.out.println(con);
      }catch(Exception e){
                e.printStackTrace();
      }
    }
```

Spring JDBC

- DataSource
  - ✓ Spring JDBC API 커넥션을 이용한 설정
    - ConnectionTest 클래스에 메서드를 추가하고 JUnit으로 테스트

```
    @Test
    public void testMySQLDataSourceConection()throws Exception{
      try(Connection con = ds.getConnection()){
                System.out.println("MySQL:" + con);
      }catch(Exception e){
                e.printStackTrace();
      }
    }
}
```

Spring JDBC

- Spring JDBC Class
  - ✓ JdbcTemplate: 기본적인 JDBC 템플릿으로서 JDBC를 이용해서 데이터베이스에 대한 접근을 제공
  - ✓ NamedParameterJdbcTemplate: PreparedStatement에서 인덱스 기반의 파라미터가 아닌 이름을 가진 파라미터 사용 가능
  - ✓ SimpleJdbcTemplate: Java 1.5 버전의 가변 인자를 이용해서 쿼리를 실행할 때 사용되는 데이터를 전달 가능
  - ✓ SimpleJdbcInsert: 데이터 삽입을 위한 인터페이스를 제공해주는 클래스
  - ✓ SimpleJdbcCall: 프로시저 호출을 위한 인터페이스를 제공해주는 클래스

Spring JDBC

- JdbcTemplate
  - ✓ 생성자
    - JdbcTemplate(DataSource dataSource)
  - ✓ select 수행
    - List query(String sql, RowMapper rowMapper)
    - List query(String sql, Object[]args, RowMapper rowMapper)
    - Object queryForObject(String sql, RowMapper rowMapper)
    - Object queryForObject(String sql, Object[]args, RowMapper rowMapper)
    - Map<String, Object> queryForMap(String sql)
    - Map<String, Object> queryForMap(String sql, Object[]args)
    - Object [] args는 sql에 ?를 사용했을 때의 실제 바인딩되는 데이터 배열
  - ✓ RowMapper는 읽어온 데이터를 반환하기 위한 템플릿으로서 Object mapRow(ResultSet rs, int rowNum)throws SQLException 메서드를 재정의해서 읽어온 데이터를 어떻게 리턴한 것인지를 결정

Spring JDBC

- JdbcTemplate
  - ✓ mapRow(ResultSet rs, int rownum) 재정의

```
public 리턴타입 mapRow(ResultSet rs, int rownum) throws SQLException {
    rs는 java.sql 패키지의 ResultSet 객체
    rownum은 행번호
    리턴 할 클래스 타입의 객체를 생성 – HashMap 또는 Dto 클래스 타입
    rs를 이용해서 데이터를 읽어서 저장한 후 객체를 리턴
}
```

Spring JDBC

- JdbcTemplate
  - ✓ 데이터베이스 작업
    - 테이블 생성

```
create table contact(
    id int primary key auto_increment,
    name varchar(20),
    phonenumber varchar(20),
    birthday date
);
```

    - 샘플 데이터 추가 및 확인

```
insert into contact(name, phonenumber, birthday) values('아담', '01037901997',
    CURDATE());
insert into contact(name, phonenumber, birthday) values('군계', '01031391997',
    '1970-06-30');

commit;

select * from contact;
```

Spring JDBC

- JdbcTemplate
  - ✓ pom.xml 파일에 lombok 사용을 위한 의존성 설정 코드 추가

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.8</version>
</dependency>
```

Spring JDBC

- JdbcTemplate
  - ✓ domain 패키지에 DTO 클래스 생성(Contact.java) - @Data를 했을 때 getter 와 setter 가 자동으로 만들어지지 않으면 getter&setter 그리고 toString을 추가

```
import java.sql.Date;

import lombok.Data;

@Data
public class Contact {
    private int id;
    private String name;
    private String phoneNumber;
    private Date birthday;
}
```

Spring JDBC

- JdbcTemplate
  - ✓ persistence 패키지에 Dao 클래스 생성(JDBCTemplateDAO.java)

```
@Repository
public class JDBCTemplateDAO {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    // 테이블의 데이터 개수를 리턴하는 메서드
    public Map<String, Object> count() {
            return jdbcTemplate.queryForMap("select count(*) count from contact");
    }
```

Spring JDBC

- JdbcTemplate
  - ✓ persistence 패키지에 Dao 클래스 생성(JDBCTemplateDAO.java)

```
// 테이블에서 전체 데이터를 읽어서 List로 리턴하는 메서드
public List<Contact> select() {
        return jdbcTemplate.query("select * from contact order by id", new
RowMapper<Contact>() {
                @Override
                public Contact mapRow(ResultSet rs, int rownum) throws
SQLException {
                        Contact person = new Contact();
                        person.setId(rs.getInt("id"));
                        person.setName(rs.getString("name"));

        person.setPhoneNumber(rs.getString("phonenumber"));
                        person.setBirthday(rs.getDate("birthday"));
                        return person;
                }
        });
}
```

Spring JDBC

- JdbcTemplate
  - ✓ persistence 패키지에 Dao 클래스 생성(JDBCTemplateDAO.java)

```
// name을 받아서 name과 일치하는 데이터를 검색해서 1개의 데이터를 리턴하는 메서드
public Contact selectOne(String name) {
        return jdbcTemplate.queryForObject("select * from contact where name=?",
new Object[] { name }, new RowMapper<Contact>() {
                @Override
                public Contact mapRow(ResultSet rs, int rownum) throws
SQLException {
                        Contact person = new Contact();
                        person.setId(rs.getInt("id"));
                        person.setName(rs.getString("name"));

        person.setPhoneNumber(rs.getString("phonenumber"));
                        person.setBirthday(rs.getDate("birthday"));
                        return person;
                }
        });
    }
}
```

Spring JDBC

- JdbcTemplate
  - ✓ applicationContext.xml 파일 수정

```
<bean class="org.springframework.jdbc.core.JdbcTemplate"
        id="jdbcTemplate">
        <constructor-arg ref="mariaDBDataSource"></constructor-arg>
</bean>

<!-- bean을 자동 생성해야 하는 패키지 설정 -->
<context:component-scan base-package="springdb" />
```

Spring JDBC

- JdbcTemplate
  - ✓ applicationContext.xml 파일 수정
    - SpringJDBCMain.java 파일을 생성하고 작성

```
public class SpringJDBCMain {
    public static void main(String[] args) {
      GenericXmlApplicationContext context = new
    GenericXmlApplicationContext(
                                    "applicationContext.xml");
      JDBCTemplateDAO dao = context.getBean(JDBCTemplateDAO.class);

      System.out.println(dao.count().get("count"));
      System.out.println("====================");

      System.out.println(dao.select());
      System.out.println("====================");

      System.out.println(dao.selectOne("아담"));

      context.close();
    }
}
```

Spring JDBC

- JdbcTemplate
  - ✓ insert, update, delete 쿼리 수행 – 영향 받은 행의 개수를 리턴
    - public int insert(String sql)
    - public int insert(String sql, Object [] args)
    - public int update(String sql)
    - public int update(String sql, Object [] args)
    - public int delete(String sql)
    - public int delete(String sql, Object [] args)

Spring JDBC

- JdbcTemplate
  - ✓ JDBCTemplateDAO에 메서드 추가

```
public int insert(Contact contact) {
    return jdbcTemplate.update("insert into contact(name, phonenumber, birthday)
    values(?,?,?)",
    new Object[] {contact.getName(), contact.getPhoneNumber(), contact.getBirthday()
    });
}
```

Spring JDBC

- JdbcTemplate
  - ✓ SpringJBDCMain.java 파일의 main 메서드 수정

```
GenericXmlApplicationContext context = new GenericXmlApplicationContext(
                                                    "applicationContext.xml");
JDBCTemplateDAO dao = context.getBean(JDBCTemplateDAO.class);

Contact contact = new Contact();
contact.setId(0);
contact.setName("jessica");
contact.setPhoneNumber("01037901997");
contact.setBirthday(new Date(new GregorianCalendar().getTimeInMillis()));
int result = dao.insert(contact);
if(result > 0 )
    System.out.println("삽입 성공");
else
    System.out.println("삽입 실패");

context.close();
```

Spring JDBC

- NamedParameterJdbcTemplate
  - ✓ JdbcTemplate 클래스와 거의 동일한 메서드를 소유하고 있으며 바인딩되는 매개변수에 Object 배열 대신에 Map 또는 MapSqlParameterSource 객체를 이용해서 파라미터를 설정하는 것이 다름
  - ✓ ? 대신에 실제 키 이름을 기재하는데 이름을 기재할 때는 :키이름 의 형태로 기재해야 하며 이름이 Map에서의 키가 되어야 함

Spring JDBC

- NamedParameterJdbcTemplate
  - ✓ persistence 패키지에 NamedContactDAO 클래스 생성

```
@Repository
public class NamedContactDAO {

    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;
```

Spring JDBC

- NamedParameterJdbcTemplate
  - ✓ persistence 패키지에 NamedContactDAO 클래스 생성

```
public Contact selectOne(String name) {
        //Map<String, Object> map = new HashMap<String, Object>();
        //map.put("name", name);

        MapSqlParameterSource map = new MapSqlParameterSource();
        map.addValue("name", name);

        return jdbcTemplate.queryForObject("select * from contact where
name=:name", map, new RowMapper<Contact>() {
                @Override
                public Contact mapRow(ResultSet rs, int rownum) throws
SQLException {
                        Contact contact = new Contact();
                        contact.setId(rs.getInt("id"));
                        contact.setName(rs.getString("name"));
                contact.setPhoneNumber(rs.getString("phonenumber"));
                        contact.setBirthday(rs.getDate("birthday"));
                        return contact;
                }
        });
    }
}
```

Spring JDBC

- NamedParameterJdbcTemplate
  - ✓ applicationContext.xml 파일에 추가

```
<bean
    class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate"
    id="namedParameterJdbcTemplate">
    <constructor-arg ref="mariaDBDataSource"></constructor-arg>
</bean>
```

Spring JDBC

- NamedParameterJdbcTemplate
  - ✓ SpringJDBCMain.java 파일의 main 메서드 수정

```
GenericXmlApplicationContext context = new
    GenericXmlApplicationContext("applicationContext.xml");

NamedContactDAO dao = context.getBean(NamedContactDAO.class);
System.out.println(dao.selectOne("jessica"));

context.close();
```

Spring JDBC

- SimpleJdbcInsert
  - ✓ sql 구문없이 Map을 이용해서 데이터를 자동으로 삽입해주는 클래스
  - ✓ DataSource 객체를 매개변수로 해서 객체를 생성
  - ✓ withTableName 메서드를 이용해서 테이블을 설정
  - ✓ 각 컬럼 이름에 해당하는 데이터를 맵(키 이름이 컬럼 이름)으로 설정한 후 execute 메서드에 매개변수로 넘겨주면 데이터가 삽입
  - ✓ 자동 생성(autoincrement)되는 컬럼을 설정하고자 하는 경우에는 usingGerneratedKeyColumns 메서드를 이용하는데 값을 알고자 할 때는 executeAndReturnKey 메서드를 이용해서 삽입을 수행

Spring JDBC

- SimpleJdbcInsert
  - ✓ SimpleJdbcInsertDAO 클래스 생성

```
@Repository
public class SimpleJdbcInsertDAO {
    @Autowired
    private SimpleJdbcInsert simpleJdbcInsert;

    public int insert(Map<String, Object> map) {
            simpleJdbcInsert.withTableName("contact");
            return simpleJdbcInsert.execute(map);
    }
}
```

Spring JDBC

- SimpleJdbcInsert
  - ✓ applicationContext.xml 파일에 추가

```
<bean
    class="org.springframework.jdbc.core.simple.SimpleJdbcInsert"
    id="simpleJdbcInsert">
    <constructor-arg ref="mariaDBDataSource"></constructor-arg>
</bean>
```

Spring JDBC

- SimpleJdbcInsert
  - ✓ SpringJDBCMain 클래스의 main 메서드 수정

```
GenericXmlApplicationContext context = new GenericXmlApplicationContext(
                "applicationContext.xml");

SimpleJdbcInsertDAO dao = context.getBean(SimpleJdbcInsertDAO.class);
Map<String, Object>map = new HashMap<String, Object>();
map.put("name", "jessica");
map.put("phonenumber", "01031391997");
GregorianCalendar cal = new GregorianCalendar();
map.put("birthday", new Date(cal.getTimeInMillis()));
System.out.println(dao.insert(map));

context.close();
```

Spring JDBC

- Database Framework
  - ✓ 스프링은 Hibernate 와 같은 ORM 기능을 제공하는 프레임워크 뿐 아니라 iBatis(MyBatis)와 같은 SQL 기반 Mapper 프레임워크와의 연동도 지원
  - ✓ 데이터베이스 연동을 할 때는 자바의 JDBC 코드를 이용해서 작성이 가능하고 스프링의 데이터베이스 연동 클래스를 이용해서 작성해도 되고 MyBatis나 Hibernate와 같은 데이터베이스 연동 프레임워크와 같이 사용 가능

MyBatis

- MyBatis
  - ✓ SQL Mapper Framework(SQL과 Java 코드의 분리)
  - ✓ SQL은 개발자가 작성하고 MyBatis는 JDBC를 이용해서 이를 실행
  - ✓ SQL 문장을 PreparedStatement로 생성하고 파라미터를 Domain 객체(VO 또는 Map)와 자동 매핑
  - ✓ Select 구문의 결과를 Domain 객체(VO 또는 Map)로 자동 매핑
  - ✓ 스프링 연동 모듈을 제공해서 스프링과 연동 가능
  - ✓ 동적 SQL을 이용한 제어 가능
  - ✓ XML 과 인터페이스를 이용하는 방법 2가지
  - ✓ 구현이 쉬워서 SI 업계에서 많이 사용
  - ✓ 필요한 의존성: 데이터베이스, spring-jdbc, mybatis, mybatis-spring 과 데이터베이스 라이브러리 의존성을 설정

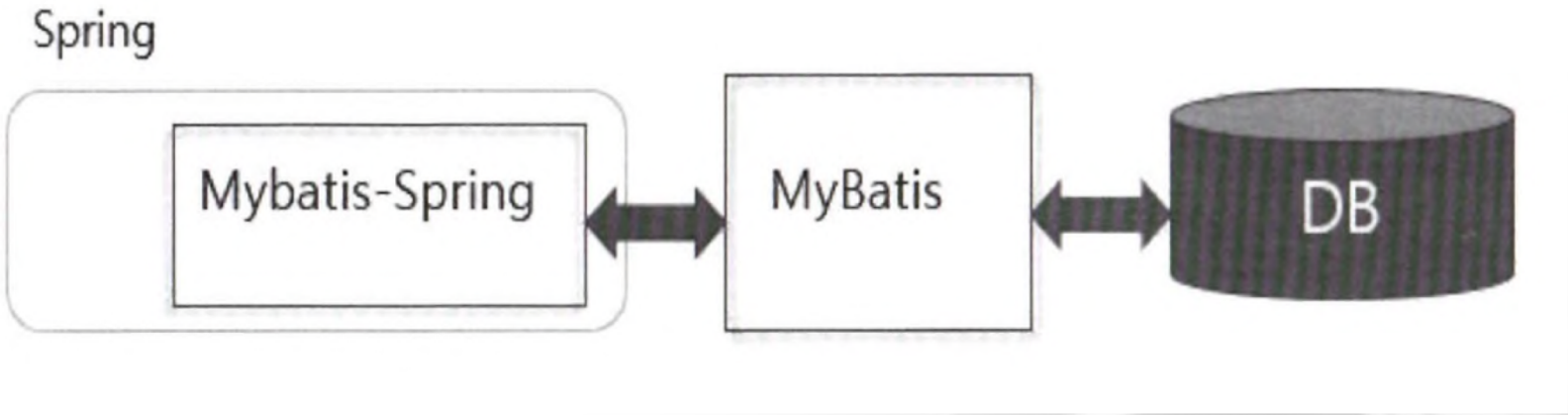
MyBatis

- MyBatis
  - ✓ 전통적인 JDBC 프로그래밍의 구조와 비교

전통적인 JDBC 프로그램	MyBatis
직접 Connection을 맺고 마지막에 close() 호출	자동으로 Connection 을 맺고 close()를 호출
PreparedStatement 직접 생성 및 처리	MyBatis 내부적으로 PreparedStatement 처러
SQL에 데이터를 직접 바인딩	#{prop} 와 같이 속성을 지정하면 내부적으로 자동 처리
ResultSet을 직접 처리	결과 타입을 지정하는 경우 자동으로 객체를 생성하고 ResultSet 처리

MyBatis

- MyBatis
  - ✓ MyBatis는 기존의 SQL을 그대로 활용할 수 있다는 장점이 있고 진입장벽이 낮은 편이어서 JDBC의 대안으로 많이 사용
  - ✓ 스프링 프레임워크의 특징 중 하나는 다른 프레임워크들을 배척하는 대신에 다른 프레임워크들과의 연동을 쉽게 하는 추가적인 라이브러리들이 많다는 것
  - ✓ MyBatis 역시 mybatis-spring이라는 라이브러리를 통해서 쉽게 연동 작업을 처리할 수 있음



MyBatis

- MyBatis의 자료형 매핑
  - ✓ number(int, float) – Integer, Double, Float, Byte, Short, Long, BigDecimal
  - ✓ char, varchar2, clob, text – String
  - ✓ date – java.sql.Date, java.util.Date
  - ✓ time – java.sql.Time, java.util.Date
  - ✓ timestamp – java.sql.Timestamp
  - ✓ blob – byte []

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ 사용되는 클래스는 SqlSessionFactoryBean 클래스와 SqlSessionTemplate 클래스
  - ✓ MyBatis의 SqlSession 사용을 위한 템플릿 객체 생성을 위한 IoC와 DI 설정

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="configLocation" value="환경설정파일위치" />
        <property name="mapperLocations" value="Mapper 파일 위치" />
</bean>

<bean id="sqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate">
        <constructor-arg index="0" ref="sqlSessionFactory" />
</bean>
```

  - mapperLocations 속성은 하나의 값을 대입해도 되고 *을 이용해서 특정 디렉토리 내의 전체 파일을 설정해도 됨

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ MyBatis의 환경 설정 파일

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

  </configuration>
```

- parameterType 이나 resultType을 패키지까지 포함된 클래스 이름을 작성하는 것이 번거로우면 환경 설정 파일에 <typeAliases>를 작성해서 사용

```
<typeAliases>
    <package name="com.gmail.ggangpae1" />
</typeAliases>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ MyBatis의 Mapper 파일 작성 – 여러 개의 파일일 때 Mapper 파일의 경로를 디렉토리 형태로 설정하기 위해서 별도의 디렉토리를 만들어서 작성
    - namespace로 감싸고 SQL을 작성
    - id는 필수
    - 모든 구문에서 parameterType은 생략 가능 – parameter 전체를 하나로 저장할 수 있는 자료형을 parameterType으로 설정
    - Select 구문에 resultType은 생략 불가하고 나머지 구문에서는 작성하면 안됨 - resultType은 select 절의 컬럼들을 모두 저장할 수 있는 자료형으로 설정
    - type 대신에 Map 과 VO를 매핑시켜서 resultMap 이나 parameterMap 으로 설정 가능

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ MyBatis의 Mapper 파일 작성

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="Mapper이름">

    <select id= " 호출할 이름 " resultType= " 결과 자료형" 또는 parameterType="자료
    형" >
            select 구문
    </select>

    <insert id= " 호출할 이름" parameterType= " 매개변수 자료형">
            INSERT INTO
                    AddressBook
                    VALUES (#{프로퍼티이름 또는 키이름})
    </insert>
</mapper>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ MyBatis의 Mapper 파일 작성 시 유의 사항
    - wildcard 문자를 이용해서 검색하는 경우에 필드명이 "title" 이고 해당 필드에서 검색할 내용을 "keyword"를 포함하는 내용이라고 하면 아래와 같이 작성

```
[MySQL]
title like CONCAT('%',#{keyword},'%')

[Oracle]
title like '%' ||  #{keyword} || '%'

[MSSQL]
title like '%' + #{keyword} + '%'
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ MyBatis의 Mapper 파일 작성 시 유의 사항
    - MyBatis에서 SQL을 작성할 때 비교 연산자를 직접 사용하면 에러가 발생하는데 < 나 > 를 태그의 시작과 종료로 인식

      where #{num} < 5

      - 태그를 있는 그대로 해석하도록 해서 해결

        where #{num} <![CDATA[<]]> 5

      - <, > 를 이용해서 해결

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ 파라미터 설정 - #{이름}
    - MyBatis는 기본적으로 PreparedStatement 클래스를 이용해서 SQL을 작성하고 실행
    - 파라미터를 ? 대신에 #{이름}으로 작성
    - 파라미터가 1개인 경우는 wrapper 클래스나 String, Date 클래스로 parameterType을 설정하고 이름은 아무것이나 입력
    - 파라미터가 여러 개 인 경우는 parameterType을 DTO 나 Map으로 설정
    - DTO 클래스를 설정한 경우는 프로퍼티(접근자 메서드 – getter) 이름으로 파라미터 이름을 설정
    - Map으로 parameterType을 설정한 경우는 키의 이름을 설정

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ SQL 실행: SqlSessionTemplate 클래스는 SqlSession 클래스를 상속받았기 때문에 SqlSession의 메서드를 그대로 이용
    - sqlSession.insert, delete, update, selectOne, selectList(Mapper이름.id, 매개변수)를 호출
    - Mapper 파일의 sql 구문이 parameter를 받지 않는 경우라면 매개변수는 생략하고 호출 가능
    - select 실행
      - selectOne은 설정한 resultType 객체를 리턴
      - selectList는 설정한 resultType의 List를 리턴
      - selectOne 메서드는 결과가 2개 이상 나오면 예외를 발생
      - Map으로 select 구문의 결과를 받는 경우 컬럼 이름이 키가 되고 데이터가 값
      - 오라클의 데이터의 타입이 Number이면 Map에서는 BigDecimal로 저장
      - Map으로 select 구문의 결과를 받는 경우 키의 이름은 오라클의 경우는 대문자 MySQL은 소문자
    - insert와 delete, update는 영향 받은 행의 개수를 정수로 리턴

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ pom.xml 파일에 dependency 추가

```
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.7</version>
</dependency>

<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>2.0.6</version>
</dependency>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ src/main/resources 디렉토리에 mybatis 디렉토리를 생성하고 mybatis-config.xml 파일을 추가하고 작성

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

</configuration>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ applicationContext.xml 파일에 코드를 추가

```
<bean class="org.mybatis.spring.SqlSessionFactoryBean"
        id="sqlSessionFactory">

        <property name="dataSource" ref="mariaDBDataSource" />
        <property value="classpath:/mybatis/mybatis-config.xml"
                name="configLocation" />

</bean>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ test 디렉토리에 MyBatisTest 클래스를 만들고 작성한 후 테스트

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:applicationContext.xml" })
public class MyBatisTest {
    @Autowired
    private SqlSessionFactory sqlFactory;

    @Test
    public void testFactory() {
            System.out.println(sqlFactory);
    }

    @Test
    public void testSession() throws Exception {
            try (SqlSession session = sqlFactory.openSession()) {
                    System.out.println(session);
            } catch (Exception e) {
                    System.out.println(e.getMessage());
                    e.printStackTrace();
            }
    }
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ Maria DB에 접속해서 데이터베이스 테이블 작성 – 기존의 테이블이 있으면 삭제

```
DROP TABLE goods;

CREATE TABLE goods (
    code      int not null,
    name      varchar(50) not null,
    manufacture    varchar(20),
    price     int not null,
    shelflife DATE,
    primary key(code)
);

SELECT * FROM goods;
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ Goods 테이블의 데이터를 표현할 DTO 클래스 작성(springdb.mybatis.domain.Good) - @Data를 했을 때 getter 와 setter 가 자동으로 만들어지지 않으면 getter&setter 그리고 toString을 추가

```
import java.sql.Date;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Good {
    private int code;
    private String name;
    private String manufacture;
    private int price;
    private Date shelflife;
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ mybatis-config.xml 파일에 DTO 클래스의 패키지를 등록 – Mapper 파일에서 패키지를 생략하면 여기 설정한 패키지에서 클래스를 찾도록 함

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
        <typeAliases>
                <package name="springdb.mybatis.domain"/>
        </typeAliases>
</configuration>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ Goods 테이블에 수행할 sql 문장을 가진 mapper 파일을 작성 (src/main/resources/mybatis 디렉토리에 mappers 디렉토리를 만들고 good.xml 파일 생성)
  - ✓ https://mybatis.org/mybatis-3/ko/sqlmap-xml.html

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="good">
    <insert id="insertGood" parameterType="Good"> insert into goods(code,
            name, manufacture, price, shelflife) values(#{code}, #{name}, #{manufacture},
            #{price}, #{shelflife})
    </insert>
</mapper>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ Good 테이블 작업을 수행할 GoodRepository 클래스를 생성하고 작성 (mybatis.persistence.GoodRepository)

```
@Repository
public class GoodRepository {
    @Autowired
    private SqlSession sqlSession;

    public void setSqlSession(SqlSession sqlSession) {
            this.sqlSession = sqlSession;
    }

    public int insertGood(Good good){
            return sqlSession.insert("insertGood", good);
    }
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodService 인터페이스를 생성하고 데이터 삽입 작업을 수행할 메서드를 선언 (mybatis.service.GoodService)

```
public interface GoodService {
    public void insertGood();
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodService를 implements 하는 GoodServiceImpl 클래스를 생성하고 메서드 구현 (mybatis.service.GoodServiceImpl)

```
@Service
public class GoodServiceImpl implements GoodService {

    @Autowired
    private GoodRepository goodRepository;
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodService를 implements 하는 GoodServiceImpl 클래스를 생성하고 메서드 구현 (mybatis.service.GoodServiceImpl)

```
public void insertGood() {
        Scanner sc = new Scanner(System.in);
        while(true) {
                System.out.print("삽입할 코드를 정수로 입력하세요(종료는 -1):");
                String input = sc.nextLine();
                int code = -1;
                try {
                        code = Integer.parseInt(input);
                } catch (Exception e) {
                        System.out.println("코드는 숫자로 입력하세요!!!!");
                        continue;
                }

                if (code == -1) {
                        break;
                }
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodService를 implements 하는 GoodServiceImpl 클래스를 생성하고 메서드 구현 (mybatis.service.GoodServiceImpl)

```
System.out.print("이름을 입력하세요:");
String name = sc.nextLine();
System.out.print("삽입할 원산지를 입력하세요:");
String manufacture = sc.nextLine();
System.out.print("삽입할 가격를 입력하세요:");
String price = sc.nextLine();

System.out.print("유통 기한을 입력하세요(YYYY-MM-DD 형식):");
String life = sc.nextLine();

int year = Integer.parseInt(life.substring(0, 4));
int month = Integer.parseInt(life.substring(5,7));
int date = Integer.parseInt(life.substring(8));
Calendar cal = new GregorianCalendar(year, month-1, date);
Date shelflife = new Date(cal.getTimeInMillis());
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodService를 implements 하는 GoodServiceImpl 클래스를 생성하고 메서드 구현 (mybatis.service.GoodServiceImpl)

```
                    Good insert = new Good(code, name, manufacture,
        Integer.parseInt(price),shelflife);
                    int r = goodRepository.insertGood(insert);
                    if (r > 0)
                            System.out.println("삽입 성공");
                    else
                            System.out.println("삽입 실패");
                    break;
            }
            sc.close();
        }
    }
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ applicationContext.xml 파일에 MyBatis 연동을 위한 설정을 변경

```
<!-- MyBatis 설정을 위한 설정 -->
<bean class="org.mybatis.spring.SqlSessionFactoryBean"
        id="sqlSessionFactory">
        <property name="dataSource" ref="mariaDBDataSource" />
        <property value="classpath:/mybatis/mybatis-config.xml"
                name="configLocation" />
        <property value="classpath:/mybatis/mappers/good.xml"
                name="mapperLocations" />
</bean>

<bean class="org.mybatis.spring.SqlSessionTemplate"
        id="sqlSession" destroy-method="clearCache">
        <constructor-arg name="sqlSessionFactory"
                ref="sqlSessionFactory" />
</bean>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ MyBatisMain 클래스를 생성한 후 main 메서드를 작성한 후 실행

```
import org.springframework.context.support.GenericXmlApplicationContext;

import db.mybatis.service.GoodService;

public class MyBatisMain {

    public static void main(String[] args) {
            GenericXmlApplicationContext context =
                                        new
    GenericXmlApplicationContext("classpath:applicationContext.xml");
            GoodService goodService = context.getBean(GoodService.class);

            goodService.insertGood();

            context.close();
            System.exit(0);
    }
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ good.xml 파일에 Goods 테이블의 모든 데이터를 가져오는 SQL을 작성

```
<select id="listGood" resultType="Good">
    select * from
    goods
</select>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodRepository 클래스에 모든 데이터를 가져오는 메서드를 작성

```
public List<Good> listGood(){
    return sqlSession.selectList("good.listGood");
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodRepository 클래스에 모든 데이터를 가져오는 메서드를 작성

```
public void listGood();
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodServiceImpl 클래스에 모든 데이터를 가져오는 메서드를 구현

```
public void listGood() {
        List<Good> list = goodRepository.listGood();
        for (Good good : list) {
                System.out.println(good);
        }
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ MyBatisMain 클래스의 main 메서드에서 전체 데이터를 가져오는 메서드를 호출 구현

```
GenericXmlApplicationContext context =
    new GenericXmlApplicationContext("classpath:applicationContext.xml");

GoodService goodService = context.getBean(GoodService.class);
//goodService.insertGood();

goodService.listGood();

context.close();
System.exit(0);
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ good.xml 파일에 code를 가지고 데이터를 검색한 후 Good 객체로 리턴하는 sql 작성

```
<select id="getGood"
    resultType="Good"
    parameterType="int">
            select *
            from goods
            where code = #{code}
</select>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodRepository 클래스에 code를 가지고 데이터를 검색한 후 Good 객체로 리턴하는 메서드 작성

```
public Good getGood(int code){
    return sqlSession.selectOne("good.getGood", code);
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodService 인터페이스에 code를 입력받은 후 데이터를 검색한 후 출력하는 메서드 선언

```
public void getGood();
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodServiceImpl 클래스에 code를 입력받은 후 데이터를 검색한 후 출력하는 메서드 작성

```
public void getGood() {
    Scanner sc = new Scanner(System.in);
    while (true) {
        System.out.print("검색할 코드를 입력하세요(종료는 -1):");
        String input = sc.nextLine();
        int code = -1;
        try {
            code = Integer.parseInt(input);
        } catch (Exception e) {
            System.out.println("코드는 숫자로 입력하세요!!!!");
            continue;
        }

        if (code == -1) {
            break;
        }
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodServiceImpl 클래스에 code를 입력받은 후 데이터를 검색한 후 출력하는 메서드 작성

```
                Good good = goodRepository.getGood(code);
                if (good == null) {
                        System.out.println("없는 코드 번호입니다.");
                } else {
                        System.out.println(good);
                }
                break;
        }
        sc.close();
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ MybatisMain 클래스의 main 메서드를 수정

```
GenericXmlApplicationContext context =
                new GenericXmlApplicationContext("classpath:applicationContext.xml");
GoodService goodService = context.getBean(GoodService.class);

goodService.getGood();

context.close();
System.exit(0);
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ good.xml 파일에 Good 객체를 받아서 데이터를 수정하는 sql 작성

```
<update
    id="updateGood"
    parameterType="Good">
            update goods
            set name=#{name},manufacture=#{manufacture}, price = #{price}
            where code = #{code}
</update>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodRepository 클래스에 Good 객체를 받아서 데이터를 수정하는 메서드 작성

```
public int updateGood(Good good) {
    return sqlSession.update("updateGood", good);
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodService 인터페이스에 데이터를 입력받은 후 데이터를 수정하는 메서드 선언

```
public void updateGood();
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodServiceImpl 클래스에 데이터를 수정하는 메서드 구현

```
public void updateGood() {
    Scanner sc = new Scanner(System.in);
    while (true) {
        System.out.print("수정할 코드를 정수로 입력하세요(종료는 -1):");
        String input = sc.nextLine();
        int code = -1;
        try {
            code = Integer.parseInt(input);
        } catch (Exception e) {
            System.out.println("코드는 숫자로 입력하세요!!!!");
            continue;
        }

        if (code == -1) {
            break;
        }
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodServiceImpl 클래스에 데이터를 수정하는 메서드 구현

```
Good good = goodRepository.getGood(code);
if (good != null) {
        System.out.print("수정할 이름을 입력하세요:");
        String name = sc.nextLine();
        System.out.print("수정할 원산지를 입력하세요:");
        String manufacture = sc.nextLine();
        System.out.print("수정할 가격를 입력하세요:");
        String price = sc.nextLine();
        Good update = new Good(code, name, manufacture,
Integer.parseInt(price), null);
        int r = goodRepository.updateGood(update);
        if (r > 0)
                System.out.println("수정 성공");
        else
                System.out.println("수정 실패");
        break;
}
```

MyBatis

❖ XML을 이용한 MyBatis 사용

✓ GoodServiceImpl 클래스에 데이터를 수정하는 메서드 구현

```
                else {
                        System.out.println("존재하지 않는 코드 번호 입니다.");
                        continue;
                }
            }
            sc.close();
        }
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ main 메서드에 데이터를 수정하는 메서드를 호출하는 코드를 작성

```
GenericXmlApplicationContext context =
                new GenericXmlApplicationContext("classpath:applicationContext.xml");
GoodService goodService = context.getBean(GoodService.class);

goodService.updateGood();

context.close();
System.exit(0);
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ good.xml 파일에 code를 받아서 데이터를 삭제하는 sql 작성

```
<delete
    id="deleteGood"
     parameterType="int">
            delete from goods
            where code = #{code}
</delete>
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodRepository 클래스에 Good 객체를 삭제하는 메서드 작성

```
public int deleteGood(int code){
    return sqlSession.delete("good.deleteGood", code);
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodService 인터페이스에 code를 입력받은 후 데이터를 삭제하는 메서드 선언

```
public void deleteGood();
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ GoodServiceImpl 클래스에 code를 입력받은 후 데이터를 삭제하는 메서드 작성

```
public void deleteGood() {
    Scanner sc = new Scanner(System.in);
    while (true) {
        System.out.print("삭제할 코드를 정수로 입력하세요(종료는 -1):");
        String input = sc.nextLine();
        int code = -1;
        try {
            code = Integer.parseInt(input);
        } catch (Exception e) {
            System.out.println("코드는 숫자로 입력하세요!!!!");
            continue;
        }

        if (code == -1) {
            break;
        }
```

MyBatis

❖ XML을 이용한 MyBatis 사용

✓ GoodServiceImpl 클래스에 code를 입력받은 후 데이터를 삭제하는 메서드 작성

```
                int r = goodRepository.deleteGood(code);
                if (r != 0)
                        System.out.println("삭제 성공");
                else
                        System.out.println("삭제 실패");
                break;
        }
        sc.close();
}
```

MyBatis

- XML을 이용한 MyBatis 사용
  - ✓ main 메서드에 삭제하는 메서드를 호출하는 코드 작성

```
GenericXmlApplicationContext context =
    new GenericXmlApplicationContext("classpath:applicationContext.xml");
GoodService goodService = context.getBean(GoodService.class);

goodService.deleteGood();

context.close();
System.exit(0);
```

MyBatis

- MyBatis Mapper 연동 방식
  - ✓ XML 만을 이용하는 방식
    - SQL을 XML 파일에 작성하는 방식으로 SQL 문의 수정이나 유지보수에 유리
    - 개발 시 코드의 양이 많아지고 복잡성이 증가
  - ✓ Annotation과 인터페이스 만을 이용하는 방식
    - 별도의 Dao 없이도 개발이 가능하기 때문에 생산성이 증가
    - SQL문을 annotation으로 작성하므로 수정이 일어나는 경우 다시 컴파일 수행
  - ✓ 인터페이스 와 XML로 작성된 SQL 문 혼용
    - 간단한 SQL문은 annotation으로 복잡한 sql 문은 XML로 처리하는 형태로 작성하는 방식으로 상황에 따라 유연하게 처리
    - 개발자에 따라 개발 방식의 차이가 있을 수 있기 때문에 유지보수가 중요한 프로젝트에서는 부적합

MyBatis

- 인터페이스를 이용한 MyBatis 활용
  - ✓ 기존 MyBatis 환경 설정 파일 과 Mapper 파일 및 Dao 대신에 하나의 인터페이스를 이용해서 작성
  - ✓ 기존 GoodRepository의 @Repository Annotation 제거

```
//@Repository
public class GoodRepository {
    //@Autowired
    private SqlSession sqlSession;
```

MyBatis

❖ 인터페이스를 이용한 MyBatis 활용

✓ mybatis.persistence 패키지에 GoodMapper 인터페이스를 생성하고 작성

```
@Repository
public interface GoodMapper {
    @Select("select * from goods")
    public List<Good> listGood();

    @Select("select * from goods where code = #{code}")
    public Good getGood(int code);

    @Insert("insert into goods(code, name, manufacture, price, shelflife) values(#{code},
    #{name}, #{manufacture}, #{price}, #{shelflife})")
    public int insertGood(Good good);

    @Update("update goods set name=#{name}, manufacture=#{manufacture}, price =
    #{price} where code = #{code}")
    public int updateGood(Good good);

    @Delete("delete from goods where code = #{code}")
    public int deleteGood(int code);
}
```

MyBatis

- 인터페이스를 이용한 MyBatis 활용
  - ✓ GoodServiceImpl 클래스의 주입받는 Dao의 클래스를 GoodRepository에서 GoodMapper로 변경

```
@Service
public class GoodServiceImpl implements GoodService {
    // @Autowired
    // private GoodRepository goodRepository;

    @Autowired
    private GoodMapper goodRepository;
```

MyBatis

- 인터페이스를 이용한 MyBatis 활용
  - ✓ applicationContext.xml 파일에서 Mapper 인터페이스를 bean으로 생성 – MyBatis 연동 빈을 수정

```
<!-- XML을 사용하는 MyBatis 설정을 위한 설정 -->
    <!--
    <bean class="org.mybatis.spring.SqlSessionFactoryBean"
    id="sqlSessionFactory">
            <property name="dataSource" ref="dataSource" />
            <property value="classpath:/mybatis/mybatis-config.xml"
    name="configLocation" />
            <property value="classpath:/mybatis/mappers/good.xml"
    name="mapperLocations" />
    </bean>

    <bean class="org.mybatis.spring.SqlSessionTemplate"
            id="sqlSession" destroy-method="clearCache">
            <constructor-arg name="sqlSessionFactory" ref="sqlSessionFactory" />
    </bean>
-->
```

MyBatis

- 인터페이스를 이용한 MyBatis 활용
  - ✓ applicationContext.xml 파일에서 Mapper 인터페이스를 bean으로 생성 – MyBatis 연동 빈을 수정

```
<!-- MyBatis 인터페이스를 이용한 설정 -->
<bean id="sqlSessionFactory"
    class="org.mybatis.spring.SqlSessionFactoryBean">
            <property name="dataSource" ref="mariaDBDataSource" />
</bean>
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
            <property name="basePackage" value="springdb.mybatis.persistence" />
</bean>
```

MyBatis

- 인터페이스를 이용한 MyBatis 활용
  - ✓ main 메서드를 수정해서 이전 메서드를 전부 다시 호출

```
GenericXmlApplicationContext context =
    new GenericXmlApplicationContext("classpath:applicationContext.xml");
GoodService goodService = context.getBean(GoodService.class);

goodService.insertGood();
goodService.getGood();

//goodService.updateGood();
//goodService.deleteGood();
//goodService.listGood();

context.close();
System.exit(0);
```

MyBatis

- MyBatis 연동(인터페이스 + XML)
  - ✓ src/main/resources/mybatis/mappers 디렉토리의 good.xml 파일을 수정

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="springdb.mybatis.persistence.GoodMapper">
    <select id="listGood"
            resultType="springdb.mybatis.domain.Good">
            select * from
            goods
    </select>
    <select id="getGood"
            resultType="springdb.mybatis.domain.Good" parameterType="int">
            select *
            from goods
            where code = #{code}
    </select>
</mapper>
```

MyBatis

- MyBatis 연동(인터페이스 + XML)
  - ✓ applicationContext.xml 파일의 MyBatis 설정 bean 수정

```
<!-- MyBatis 인터페이스를 이용한 설정 -->
<bean id="sqlSessionFactory"
    class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="mariaDBDataSource" />
    <property name="mapperLocations"
            value="classpath:/mybatis/mappers/good.xml" />
</bean>

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage"
            value="springdb.mybatis.persistence" />
</bean>
```

MyBatis

- MyBatis 연동(인터페이스 + XML)
  - ✓ GoodMapper 인터페이스 수정

```
@Repository
public interface GoodMapper {
    public List<Good> listGood();

    public Good getGood(int code);

    @Insert("insert into goods(code, name, manufacture, price, shelflife) values(#{code},
    #{name}, #{manufacture}, #{price}, #{shelflife})")
    public int insertGood(Good good);

    @Update("update goods set name=#{name}, manufacture=#{manufacture}, price =
    #{price} where code = #{code}")
    public int updateGood(Good good);

    @Delete("delete from goods where code = #{code}")
    public int deleteGood(int code);
}
```

MyBatis

- MyBatis 연동(인터페이스 + XML)
  - ✓ GoodMapper 인터페이스 수정

```
GenericXmlApplicationContext context =
    new GenericXmlApplicationContext("classpath:applicationContext.xml");
GoodService goodService = context.getBean(GoodService.class);

goodService.listGood();

context.close();
System.exit(0);
```

Database 로그 기록

- Database 로그 기록
  - ✓ 데이터베이스를 사용해서 개발하다가 보면 가끔 잘못된 SQL이나 잘못된 속성의 이름으로 인해서 예외가 발생하는 경우가 있는데 이런 경우를 대비해서 로그를 보다 자세히 조사할 수 있도록 로그를 설정해 주는 것이 좋음
  - ✓ log4jdbc-log4j2라는 라이브러리를 이용하면 기존의 jdbc datasource 보다 더 상세한 로그를 콘솔에서 확인
  - ✓ log4jdbc를 이용하는 경우 속도가 기존보다 저하될수 있고, 데이터베이스에 따라서 지원되지 않는 경우도 있으므로 설정 후에 정상적 동작을 반드시 확인
  - ✓ pom.xml 파일에 의존성 설정

```
<dependency>
 <groupId>org.bgee.log4jdbc-log4j2</groupId>
 <artifactId>log4jdbc-log4j2-jdbc4</artifactId>
 <version>1.16</version>
</dependency>
```

Database 로그 기록

- Database 로그 기록
  - ✓ src/main/resources 디렉토리에 로그 설정 파일인 log4jdbc.log4j2.properties 을 생성하고 아래 코드 추가

```
log4jdbc.spylogdelegator.name=net.sf.log4jdbc.log.slf4j.Slf4jSpyLogDelegator
```

Database 로그 기록

- Database 로그 기록
  - ✓ applicationContext.xml 파일에서 DataSource 의 데이터베이스 접속 URL을 변경

```
<!-- Maria DB DataSource -->
<bean id="mariaDBDataSource"
            class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
    value="net.sf.log4jdbc.sql.jdbcapi.DriverSpy"></property>
    <property name="url" value="jdbc:log4jdbc:mariadb://localhost:3306/adam" />
    <property name="username" value="root"/>
    <property name="password" value="wnddkd"  />
</bean>
```

Database 로그 기록

- Database 로그 기록
  - ✓ 실행을 다시 한 후 콘솔 확인

```
11:20:14.719 [main] INFO  jdbc.resultsettable -
-----|--------------|-----------|------|
code |name          |manufacture |price |
-----|--------------|-----------|------|
1    |apple         |korea       |1500  |
2    |watermelon    |korea       |15000 |
3    |oriental melon |korea      |1000  |
4    |banana        |Philippines |500   |
5    |lemon         |korea       |1500  |
7    |무화과         |전남영암      |2000  |
-----|--------------|-----------|------|
```

Database 로그 기록

- 출력할 로그 레벨을 설정할려면 src/main/resources 디렉토리에 log4j.xml 파일을 작성하고 설정

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration>
<configuration>
<include resource="org/springframework/boot/logging/logback/base.xml" />
<logger name="jdbc.splonly" level="DEBUG" />
<logger name="jdbc.sqltiming" level="INFO" />
<logger name="jdbc.audit" level="WARN" />
<logger name="jdbc.resultset" level="ERROR" />
<logger name="jdbc.resultsettable" level="ERROR" />
<logger name="jdbc.connection" level="INFO" />
</configuration>
```

Transaction

- Transaction
  - ✓ 시스템에서 사용되는 분해할 수 없는 업무 처리의 단위 또는 시스템에서 한꺼번에 수행되어야 하는 작업의 논리적 단위
  - ✓ NoSQL 과 Big Data가 급부상하면서 RDBMS가 제공하는 엄격한 Transaction보다는 대용량 데이터 처리를 위한 느슨한 방식의 무결성 처리 기법을 적용하는 곳이 늘고 있으나 관계형 데이터베이스(RDBMS)가 제공하는 Transaction의 개념은 여전히 중요
  - ✓ 하나의 Transaction 내의 모든 작업은 전부 수행되거나 하나도 수행되지 않아야 함 – All or Nothing(원자성)
  - ✓ Transaction 작업
    - Commit: 현재 Transaction을 완료 – 되돌릴 수 없음
    - Rollback: 현재 Transaction을 철회 – 초기 상태로 되돌림
  - ✓ Transaction이 commit 되는 경우
    - 명시적 commit 호출
    - DDL, DCL 명령문을 정상적으로 수행한 경우
    - 데이터베이스 접속 프로그램이 정상 종료된 경우
  - ✓ Transaction이 rollback 되는 경우
    - 명시적 rollback 호출
    - 데이터베이스 접속 프로그램이 비 정상 종료된 경우

Transaction

- Spring의 Transaction 적용
  - ✓ 스프링은 코드 기반의 Transaction 처리 뿐 아니라 선언적 처리와 Annotation 기반의 Transaction 처리를 지원
  - ✓ 스프링은 org.springframework.transaction.PlatformTransactionManager 인터페이스를 이용해서 Transaction 처리를 추상화 한 후 데이터베이스 연동 기술에 따라 알맞은 PlatformTransactionManager 구현 클래스를 제공하므로 사용하는 데이터베이스 연동 방법에 따라 적절하게 Transaction 매니저를 구현
  - ✓ Spring JDBC 또는 MyBatis를 사용하는 경우의 TransactionManager

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```

Transaction

- Spring의 Transaction 적용
  - ✓ Transaction 설정 방법
    - @Transactional Annotation을 이용해서 Transaction을 설정
    - 설정 파일에 AOP와 <tx:advice> 와 <tx:method> 태그를 이용해서 설정
  - ✓ Annotation을 이용한 설정
    - 하나의 Transaction으로 처리되어야 하는 메서드 위에 Annotation을 기재하고 속성을 설정 – 보통 Service 클래스의 메서드에 적용
    - 속성
      - propagation
      - isolation
      - readonly
      - rollbackFor: Transaction을 Rollback 할 예외 타입을 설정(rollbackFor={Exception.class}
      - noRollbackFor: Transaction을 Rollback하지 않을 예외 타입을 설정
      - timeout: Transaction의 타임아웃 시간을 초 단위로 설정
    - Annotation을 이용할 때는 메서드 위에 설정하고 설정 파일에 <tx:annotation-driven transaction-manager="TransactionManager 빈의 id"/> 태그를 설정

Transaction

- Spring의 Transaction 적용
  - ✓ 속성
    - propagation: <tx:method> 에서는 propagation Attribute 값으로 @Transactional 에서는 propagation Element로 지정
    - propagation Element의 값: org.springframework.transaction.annotation.Propagation에 정의된 것을 사용
      - REQUIRED: 디폴트 속성으로 모든 Transaction 매니저가 지원하며 보통은 이 속성이면 충분한데 미리 시작된 Transaction이 있으면 참여하고 없으면 새로 시작
      - SUPPORTS: 시작된 Transaction이 있으면 참여하고 그렇지 않으면 Transaction 없이 진행
      - MANDATORY: REQUIRED와 비슷하게 이미 시작된 Transaction이 있으면 참여하는 것으로 Transaction이 없으면 새로 시작하는 대신 예외를 발생
      - REQUIRES_NEW: 항상 새로운 Transaction을 시작하는 것으로 이미 진행 중인 Transaction이 있으면 Transaction을 잠시 보류
      - NOT_SUPPORTED: Transaction을 사용하지 않게 하는 것으로 이미 진행 중인 Transaction이 있으면 보류
      - NEVER: Transaction을 사용하지 않도록 강제하는 것으로 이미 진행 중인 Transaction도 존재하면 안되며 있다면 예외를 발생

Transaction

- Spring의 Transaction 적용
  - ✓ 속성
    - propagation Element의 값:
      org.springframework.transaction.annotation.Propagation에 정의된 것을 사용
      - REQUIRED: 디폴트 속성으로 모든 Transaction 매니저가 지원하며 보통은 이
        NESTED: 이미 진행중인 Transaction이 있으면 중첩 Transaction을 시작하는 것으로 중첩 Transaction은 Transaction 안에 다시 Transaction을 만드는 것으로 중첩된 Transaction은 먼저 시작된 부모 Transaction의 Commit과 Rollback에는 영향을 받지만 자신의 Commit과 Rollback은 부모 Transaction에게 영향을 주지 않는데 Savepoint를 지원하는 데이터베이스에서 사용 가능

Transaction

- ❖ Spring의 Transaction 적용
  - ✓ 속성
    - isolation: Transaction 격리 수준은 동시에 여러 Transaction이 진행될 때에 Transaction의 작업 결과를 여타 Transaction에게 어떻게 노출할 것인지를 결정하는 기준으로 <tx:method>의 isolation Attribute와 @Transactional 의 isolation Element로 지정
      - ◆ Default: 사용하는 데이터 액세스 기술 또는 DB 드라이버의 디폴트 설정을 따르는 것으로 보통 드라이버의 격리 수준은 DB의 격리 수준을 따르는 것이 일반적인데 대부분의 DB는 READ_COMMITTED가 기본 격리수준
      - ◆ READ_UNCOMMITTED: 가장 낮은 격리수준으로 하나의 Transaction이 Commit되기 전에 그 변화가 다른 Transaction에 그대로 노출되는 형태로 가장 빠르기 때문에 데이터의 일관성이 조금 떨어지더라도 성능을 극대화할 때 의도적으로 사용
      - ◆ READ_COMMITTED: 실제로 가장 많이 사용되는 격리수준으로 READ_UNCOMMITTED와 달리 다른 Transaction이 Commit하지 않은 정보는 읽을 수 없으며 대신 하나의 Transaction이 읽은 로우를 다른 Transaction이 수정할 수 있는데 Transaction이 같은 로우를 읽을 경우 다른 내용이 발견될 수 있음

Transaction

- Spring의 Transaction 적용
  - ✓ 속성
    - isolation: Transaction 격리 수준은 동시에 여러 Transaction이 진행될 때에 Transaction의 작업 결과를 여타 Transaction에게 어떻게 노출할 것인지를 결정하는 기준으로 <tx:method>의 isolation Attribute와 @Transactional 의 isolation Element로 지정
      - REPEATABLE_READ: 하나의 Transaction이 읽은 로우를 다른 Transaction이 수정하는 것을 금지하지만 새로운 로우를 추가하는 것은 제한하지 않아서 SELECT로 조건에 맞는 로우를 전부 가져오는 경우 Transaction이 끝나기 전에 추가된 로우가 발견될 수 있음
      - SERIALIZABLE: 가장 강력한 Transaction 격리수준으로 Transaction을 순차적으로 진행시켜 주기 때문에 여러 Transaction이 동시에 같은 테이블의 정보를 액세스하지 못하게 하는 것으로 가장 안전한 격리수준이지만 가장 성능이 떨어지기 때문에 극단적으로 안전한 작업이 필요한 경우가 아니라면 자주 사용되지 않음

Transaction

- ❖ Spring의 Transaction 적용
  - ✓ 속성
    - readonly
      - ◆ Transaction을 읽기 전용으로 설정할 수 있는데 성능을 최적화하기 위해 사용할 수도 있고 특정 Transaction 작업 안에서 쓰기 작업이 일어나는 것을 의도적으로 방지하기 위해 사용
      - ◆ 읽기 전용 Transaction이 시작된 이후 INSERT, UPDATE, DELETE 같은 쓰기 작업이 진행되면 예외가 발생
      - ◆ read-only Attribute 또는 readOnly Element로 지정
    - Transaction 제한시간(timeout)
      - ◆ 이 속성을 이용하면 Transaction에 제한시간을 지정할 수 있는 속성으로 값은 초 단위로 지정
      - ◆ 디폴트는 Transaction 시스템의 제한시간을 따르는 것
      - ◆ Transaction 제한시간을 직접 지정하는 경우 이 기능을 지원하지 못하는 일부 Transaction 매니저는 예외 발생
      - ◆ XML에서는 <tx:method> 의 timeout Attribute를 이용하고 @Transactional Annotation에는 timeout 엘리먼트로 지정

Transaction

- ❖ Spring의 Transaction 적용
  - ✓ 속성
    - Transaction Rollback 예외(rollback-for, rollbackFor, rollbackForClassName)
      - ◆ 선언적 Transaction에서는 런타임 예외가 발생하면 Rollback되지만 예외가 전혀 발생하지 않거나 체크 예외가 발생하면 Commit
      - ◆ 체크 예외를 Commit 대상으로 삼은 이유는 체크 예외가 예외적인 상황에서 사용되기보다는 리턴 값을 대신해서 비즈니스적인 의미를 담은 결과를 돌려주는 용도로 많이 사용되기 때문
      - ◆ 스프링에서는 데이터 액세스 기술의 예외는 런타임 예외로 전환돼서 던져지므로 런타임 예외만 Rollback 대상으로 삼은 것이지만 원한다면 기본 동작방식을 바꿀 수 있는데 체크 예외지만 Rollback 대상으로 삼아야 하는 것이 있다면 XML의 rolback-for Attribute나 Annotation의 rollbackFor 또는 rollbackForClassName Element를 이용해서 예외를 지정
      - ◆ rollback-for 나 rollbackForClassName은 예외 이름을 넣으면 되고 rollbackFor는 예외 클래스를 직접 기재

Transaction

- ❖ Spring의 Transaction 적용
  - ✓ 속성
    - Transaction Rollback 예외(rollback-for, rollbackFor, rollbackForClassName)
      - ◆ <tx:method> 라면 다음과 같이 지정

```
<tx:method name="get*" read-only="true" rollback-
    for="NoSuchMemberException" />
```

      - ◆ @Transactional 에서는 클래스를 직접 사용하는 것이 가능
      - ◆ @Transactional(readOnly=true, rollbackFor=NoSuchMemberException.class)

Transaction

- ❖ Spring의 Transaction 적용
  - ✓ 속성
    - Transaction Commit 예외(no-rollback-for, noRollbackFor, noRollbackForClassName)
      - ◆ rollback-for 속성과는 반대로 기본적으로는 Rollback 대상인 런타임 예외를 Transaction Commit 대상으로 지정
      - ◆ 사용 방법은 rollback-for 와 동일
    - 6가지 Transaction 속성은 모든 Transaction 경계 설정 속성에 사용 가능
    - Transaction마다 일일이 Transaction 속성을 지정하는 건 매우 번거롭고 불편
    - 세밀하게 튜닝해야 하는 시스템이 아니라면 메서드 이름 패턴을 이용해서 Transaction 속성을 한 번에 지정하는 aop/tx 스키마 태그 방식이 편리
    - read-only 속성 정도만 사용하고 나머지는 디폴트로 지정하는 경우가 많은데 세밀한 속성은 DB나 WAS의 Transaction 매니저의 설정을 이용해도 되기 때문
    - 세밀한 Transaction 속성 지정이 필요한 경우에는 @Transactional을 사용하는 것이 좋은데 Transaction 속성이 전체적으로 어떻게 지정되어 있는지 한눈에 보기 힘들다는 단점이 있고 개발자가 코드를 만들 때 Transaction 속성을 실수로 잘못 지정하는 등의 위험이 있기 때문에 사전에 Transaction 속성 지정에 관한 정책이나 가이드라인을 잘 만들어 두어야 함

Transaction

❖ mybatis.persistence 패키지에 TransactionRepository 클래스를 생성하고 작성

```
@Repository
public class TransactionRepository {
    // 맵을 이용해서 데이터를 추가하는 클래스의 변수
    @Autowired
    private SimpleJdbcInsert template;

    public void insert() {
       template.withTableName("goods");
       Map<String, Object> map = new HashMap<String, Object>();
       map.put("code", 300);
       map.put("name", "쌀");
       map.put("manufacture", "이천");
       map.put("price", 50000);

       template.execute(map);
       template.execute(map);
    }
}
```

Transaction

- applicationContext.xml 파일에 추가

```
<bean
        class="org.springframework.jdbc.core.simple.SimpleJdbcInsert"
        id="simpleJdbcInsert">
        <constructor-arg ref="dataSource"></constructor-arg>
</bean>
```

Transaction

❖ main 메서드를 소유한 TransactionMain.java 파일을 만들고 main 메서드 수정

```
import org.springframework.context.support.GenericXmlApplicationContext;

public class TransactionMain {
    public static void main(String[] args) {
      GenericXmlApplicationContext context =
                                    new
    GenericXmlApplicationContext("classpath:applicationContext.xml");
      TransactionRepository tDao = context.getBean(TransactionRepository.class);
      tDao.insert();
      context.close();
    }
}
```

Transaction

- 실행 한 후 결과를 확인하면 하나의 데이터는 삽입됨

Transaction

- applicationContext.xml 파일에 tx 네임스페이스를 추가하고 bean 코드 추가

```
<bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="mariaDBDataSource" />

</bean>

<tx:annotation-driven />
```

Transaction

❖ TransactionRepository 클래스의 데이터 삽입 메서드 수정

```
@Transactional
public void insert() {
    template.withTableName("goods");
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("code", 100);
    map.put("name", "쌀");
    map.put("manufacture", "이천");
    map.put("price", 50000);

    template.execute(map);
    template.execute(map);
}
```

Transaction

- 프로그램을 다시 실행 시켜서 확인하면 2개의 삽입 중 하나만 실패해도 모두 실패

Hibernate

- Hibernate
  - ✓ 자바 기반의 ORM(Object Relationship Mapper): 자바 객체를 RDBMS의 하나의 ROW와 맵핑
  - ✓ ORM 개념은 오래 전부터 나왔는데 처음의 시초는 EJB Entity Beans라고 할 수 있는데 EJB는 여러 가지 강력한 기능을 가졌음에도 불구하고 사용법이 너무 어려워서 근래에는 거의 사용되지 않고 있는데 사용법은 편하게 하면서도 더 많은 기능을 지원하게 한 것이 ORM인데 그 중에서 대표적인 것이 Hibernate
  - ✓ Hibernate가 개발된 후에 Java Spec에도 ORM의 표준 Spec을 정의했는데 이것이 JPA(Java Persistence API)
  - ✓ JPA는 하나의 Spec으로 JPA에 대한 구현체가 필요
  - ✓ JPA 구현체는 TopLink, OpenJPA 등 여러가지가 있는데 이렇게 ORM API를 JPA를 통해서 추상화 시키고 필요에 따라 다른 ORM으로 쉽게 교체가 가능하게 만든 개념이지만 실제 ORM은 거의 Hibernate만 사용
  - ✓ JPA는 기본적으로 Jboss나 Weblogic과 같은 JEE 컨테이너를 가정으로 개발되어서 JEE 컨테이너 없이 Tomcat과 같은 Servlet 컨테이너만 사용할 경우 큰 이득을 보기 어려움

Hibernate

- Hibernate
  - ✓ 자체적으로 쿼리를 생성하고 객체들을 DB에서 로딩할 때 레퍼런스 된 객체를 로딩하는 동작 등을 수행하게 되는데 이 때 제대로 특성을 이해하고 사용하지 않으면 장애를 일으키는 경우가 많고 데이터베이스에 대한 전문적인 지식을 갖추고 사용해야 하기 때문에 SI 업계에서는 기피하고 솔루션 개발업체에서는 선호
  - ✓ Hibernate의 장점
    - 복잡한 JDBC 코드의 제거
    - 성숙된 ORM 기능 제공
    - 기존 데이터베이스와 연동이 수월
    - Spring Framework와 통합기능 제공
    - 성능이 우수

Hibernate

- Hibernate 와 MyBatis
  - ✓ Hibernate
    - 객체 <--도메인 설계,일부HQL--> DB테이블/뷰
    - 장점: 자동화. ER/클래스 다이어그램에 의한 자동 JOIN/Cascade, Middlegen, MyEclipse 등 자동화 툴 존재
    - 단점: 기술적 난이도 높음(아키텍쳐 이해, 설계, 트랜젝션, 캐쉬 구성, HQL 등등)
  - ✓ MyBatis
    - 객체 <--sql 구현--> DB테이블/뷰
    - 장점: 기술적 난이도 낮음
    - 단점: 효율이 좋다고 할 수 없음
  - ✓ SI업체, 이미 DB가 존재, DB설계가 엉망 인 경우 --> MyBatis
  - ✓ 신규개발, 다양한DB에서 동작, 제대로 된 DB설계(ER다이어그램, 클래스/도메인 다이어그램) 인 경우 --> Hibernate

Hibernate

- Hibernate 프레임워크 구성
  - ✓ Configuration 객체: 데이터베이스 연결과 클래스 매핑 설정
  - ✓ SessionFactory 객체: Configuration 객체를 이용해서 생성
    - 데이터베이스 사용을 위한 Session 객체를 생성하기 위한 객체
    - 무거운 객체이므로 애플리케이션 시작 시에 생성하고 유지
    - 데이터베이스 당 하나의 객체를 구성 파일을 사용하여 생성
  - ✓ Session: 물리적인 데이터베이스 연결 객체
    - 가벼운 객체이므로 필요할 때 마다 생성하여 사용하고 소멸 시킬 수 있음
  - ✓ Transaction: 트랜잭션 처리를 위한 객체
  - ✓ Query: SQL 또는 HQL을 사용하여 데이터베이스에서 데이터를 가져오거나 생성함
  - ✓ Criteria: Criteria 질의를 사용하여 데이터베이스에서 데이터를 가져오거나 생성

Hibernate

- 매핑 타입
  - ✓ Integer: int, java.lang.Integer
  - ✓ Long: long, java.lang.Long
  - ✓ Double: double, java.lang.Double
  - ✓ Boolean, yes/no. true/false: boolean, java.lang.Boolean
  - ✓ Date: java.util.Date, java.sql.Date
  - ✓ Time: java.util.Date, java.sql.Time
  - ✓ Timestamp: java.util.Date, java.sql.Timestamp
  - ✓ String, Char, Varchar, Text, CLOB: java.lang.String
  - ✓ Blob: java.sql.Blob

Hibernate

- 매핑 방식
  - ✓ Hibernate 매핑 XML 이용(id로 PK 매핑, property로 컬럼 매핑)

```
<hibernate-mapping package="com.choongang.hibernate.dto">
    <class name="MyContact" table="MYCONTACT">
            <id name="num" column="num">
            </id>
            <property name="name" column="name" />
            <property name="addr" column="addr" />
    </class>
</hibernate-mapping>
```

Hibernate

- 매핑 방식
  - ✓ 복합키 설정

```
<composite-id>
      <key-property name="user_id" column="user_id" />
      <key-property name="acc_id" column="acc_id" />
</composite-id>
```

  - ✓ Sequence 설정

```
<id name="id" column="item_id">
    <generator class="sequence">
        <param name="sequence">NAME_OF_YOUR_SEQUENCE</param>
    </generator>
</id>
```

Hibernate

❖ 하이버네이트 bean 설정

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
      <property name="dataSource" ref="DataSource Bean의 id" />
      <property name="mappingResources">
            <list>
                  <value>설정파일경로</value>
            </list>
      </property>
      <property name="hibernateProperties">
            <value>
                  hibernate.dialect=데이터베이스 종류
            </value>
      </property>
</bean>
```

Hibernate

❖ Hibernate - 하이버네이트 DB 별 Dialect 설정

RDBMS	Dialect
DB2	org.hibernate.dialect.DB2Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MariaDB	org.hibernate.dialect.MariaDB103Dialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
MySQL8	org.hibernate.dialect.MySQL8Dialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i/10g	org.hibernate.dialect.Oracle9Dialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect

Hibernate

- 질의 방식
  - ✓ SQL 질의

    select * from product where product_id=:id
  - ✓ HQL 질의(select 절을 사용하지 않고 테이블 이름 대신 매핑 된 클래스 이름을 대입)

    from ProductEntity where id=:id
  - ✓ Criteria 질의

    Criterion eqCriterion = Restriction.eq("id", id);

Hibernate

- Session 객체의 주요 메서드
  - ✓ Transaction beginTransaction()
  - ✓ SQLQuery createSQLQuery(String queryString)
    list(), executeUpdate()를 호출
  - ✓ Transaction getTransaction()
  - ✓ Object get(clazz, 조건)
  - ✓ save(Object): 데이터 삽입 – 삽입 후 아이디로 설정된 값 리턴
  - ✓ update(Object)
  - ✓ delete(Object)

Hibernate

- Maria DB 사용
  - ✓ pom.xml 파일의 properties 태그 안에서 Hibernate(5.4.2) 버전 변경

```
<!-- Hibernate / JPA -->
<hibernate.version>5.4.2.Final</hibernate.version>
```

Hibernate

- Maria DB 사용
  - ✓ pom.xml 파일의 dependencies 태그에 Hibernate 사용을 위한 의존성 라이브러리 추가

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${spring-framework.version}</version>
</dependency>
```

Hibernate

- Maria DB 사용
  - ✓ 테이블을 표현할 VO 클래스 생성(springdb.hibernate.domain.Good) - @Data를 했을 때 getter 와 setter 가 자동으로 만들어지지 않으면 getter&setter 그리고 toString을 추가

```
@Entity
@Table(name="goods")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Good {
    @Id
    private int code;
    @Column
    private String name;
    @Column
    private String manufacture;
    @Column
    private int price;
    @Column
    private Date shelflife;
}
```

Hibernate

- Maria DB 사용
  - ✓ applicationContext.xml 파일에 hibernate 연결 정보 설정

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="mariaDBDataSource" />
    <property name="annotatedClasses">
            <list>
                    <value>springdb.hibernate.domain.Good</value>
            </list>
    </property>
    <property name="hibernateProperties">
            <value>
                    hibernate.dialect=org.hibernate.dialect.MySQLDialect
            </value>
    </property>
</bean>
```

Hibernate

- Maria DB 사용
  - ✓ applicationContext.xml 파일에 Hibernate 사용을 위한 설정과 트랜잭션 사용을 위한 bean 설정 태그 추가

```
<!-- @Repository 클래스들에 대해 자동으로 예외를 Spring의 DataAccessException으로
    일괄 변환해주는 설정 -->
<bean
    class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProc
    essor" />

<bean id="transactionManager"
            class="org.springframework.orm.hibernate5.HibernateTransactionManager">
            <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Hibernate

- Maria DB 사용
  - ✓ HibernateTest 클래스를 생성하고 Hibernate 설정을 확인하는 코드를 작성하고 테스트

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:applicationContext.xml" })
public class HibernateTest {
    @Autowired
    private SessionFactory sessionFactory;

    @Test
    public void hibernateTest() {
            try {
                    System.out.println("Hibernate:" + sessionFactory);
            }catch(Exception e) {
                    System.out.println(e.getMessage());
                    e.printStackTrace();
            }
    }
}
```

Hibernate

- Maria DB 사용
  - ✓ 데이터베이스 관련 메서드를 구현한 GoodHibernateRepository 클래스를 psringdb.hibernate.persistence 패키지에 생성하고 데이터를 삽입하는 메서드를 구현

```
@Repository
public class GoodHibernateRepository {
    @Autowired
    private SessionFactory sessionFactory;

    public void insertGood(Good good) {
            Session session = sessionFactory.getCurrentSession();
            session.save(good);
    }
}
```

Hibernate

- Maria DB 사용
  - ✓ Good 관련된 요청을 처리할 메서드를 소유한 GoodHibernateService 인터페이스를 생성하고 데이터를 삽입하는 메서드를 선언

```
public interface GoodHibernateService {
    public void insertGood();
}
```

Hibernate

- Maria DB 사용
  - ✓ Good 관련된 요청을 처리할 메서드를 구현한 GoodHibernateService 인터페이스를 implements 한 GoodHibernateServiceImpl 클래스를 생성하고 데이터를 삽입하는 메서드를 구현

```
@Service
public class GoodHibernateServiceImpl implements GoodHibernateService {

    @Autowired
    private GoodHibernateRepository goodHibernateRepository;

    @Override
    @Transactional
    public void insertGood() {
            Good good = new Good();
            good.setCode(101);
            good.setName("로렉스 시계");
            good.setManufacture("스위스");
            good.setPrice(1500000);

            Calendar cal = new GregorianCalendar();
            good.setShelflife(new Date(cal.getTimeInMillis()));

            goodHibernateRepository.insertGood(good);
    }
}
```

Hibernate

- Maria DB 사용
  - ✓ main 메서드를 소유한 HibernateMain 클래스를 만들어서 실행

```
public class HibernateMain {
    public static void main(String[] args) {
            GenericXmlApplicationContext context =
                    new GenericXmlApplicationContext(
                            "classpath:applicationContext.xml");
            GoodHibernateService goodService =
    context.getBean(GoodHibernateService.class);
            goodService.insertGood();
            context.close();
    }
}
```

Hibernate

- Maria DB 사용
  - ✓ GoodHibernateHibernate 클래스에 전체 데이터를 읽어오는 메서드를 구현

```
public List<Good> listGood() {
    CriteriaQuery<Good> criteriaQuery =
    sessionFactory.getCurrentSession().getCriteriaBuilder().createQuery(Good.class);
            criteriaQuery.from(Good.class);
    List<Good> list =
    sessionFactory.getCurrentSession().createQuery(criteriaQuery).getResultList();


    //List<Good> list = sessionFactory.getCurrentSession().createSQLQuery("select *
    from goods").addEntity(Good.class).list();

    return list;
}
```

Hibernate

- Maria DB 사용
  - ✓ GoodHibernateService 인터페이스에 전체 데이터를 읽어오는 메서드를 선언

```
public void listGood();
```

Hibernate

- Maria DB 사용
  - ✓ GoodHibernateServiceImpl 클래스에 전체 데이터를 읽어오는 메서드를 구현

```
@Transactional
public void listGood() {
    List<Good> list = goodHibernateRepository.listGood();
    for(Good good : list){
            System.out.println(good);
    }
}
```

Hibernate

- Maria DB 사용
  - ✓ main 메서드를 수정

```
public static void main(String[] args) {
    GenericXmlApplicationContext context =
            new GenericXmlApplicationContext(
                    "classpath:applicationContext.xml");
    GoodHibernateService goodService = context.getBean(GoodHibernateService.class);
    goodService.listGood();
    context.close();
}
```

Hibernate

- Maria DB 사용
  - ✓ code를 가지고 하나의 데이터를 검색해 올 메서드를 GoodHibernateRepository 클래스에 구현

```
public Good getGood(int code) {
    Good item = (Good)sessionFactory.getCurrentSession().get(Good.class, code);
    return item;


//          List<Good> list = sessionFactory.getCurrentSession().createSQLQuery("select
    * from goods where code=:code").addEntity(Good.class)
//                                      .setParameter("code", code).list();
//          if(list == null || list.size() < 1)
//                  return null;
//          return list.get(0);

}
```

Hibernate

- Maria DB 사용
  - ✓ code를 가지고 하나의 데이터를 검색해 올 메서드를 GoodHibernateService 인터페이스에 선언

```
public void getGood(int code);
```

Hibernate

- Maria DB 사용
  - ✓ code를 가지고 하나의 데이터를 검색해 올 메서드를 GoodHibernateServiceImpl 클래스에 구현

```
@Override
@Transactional
public void getGood(int code) {
    Good good = goodHibernateRepository.getGood(code);
    System.out.println(good);
}
```

Hibernate

- Maria DB 사용
  - ✓ main 메서드를 수정

```
public static void main(String[] args) {
    GenericXmlApplicationContext context =
            new GenericXmlApplicationContext(
                    "classpath:applicationContext.xml");
     GoodHibernateService goodService = context.getBean(GoodHibernateService.class);
    goodService.getGood(101);
    context.close();
}
```

Hibernate

- Maria DB 사용
  - ✓ 가장 큰 Code를 가져오는 메서드를 GoodHibernateRepository 클래스에 구현

```
public int maxCode() {
    int maxCode = (Integer)sessionFactory.getCurrentSession().createSQLQuery("select
    max(code) as maxcode from goods").addScalar("maxcode", new
    IntegerType()).uniqueResult();
    return maxCode;
}
```

Hibernate

- Maria DB 사용
  - ✓ 가장 큰 Code를 출력하는 메서드를 GoodHibernateService 인터페이스에 선언

```
public void maxCode();
```

Hibernate

- Maria DB 사용
  - ✓ 가장 큰 Code를 출력하는 메서드를 GoodHibernateServiceImpl 클래스에 구현

```
@Override
@Transactional
public void maxCode() {
    int maxCode = goodDao.maxCode();
    System.out.println(maxCode);
}
```

Hibernate

- Maria DB 사용
  - ✓ 프로젝트를 실행 시킬 main 메서드를 수정

```
public static void main(String[] args) {
    GenericXmlApplicationContext context =
            new GenericXmlApplicationContext(
                    "classpath:applicationContext.xml");
    GoodHibernateService goodService = context.getBean(GoodHibernateService.class);
    goodService.maxCode();
    context.close();
}
```

Hibernate

- Maria DB 사용
  - ✓ 데이터를 수정하는 메서드를 GoodHibernateRepository 클래스에 구현

```
public void updateGood(Good good) {
        Session session = sessionFactory.getCurrentSession();
        session.update(good);
}
```

Hibernate

- Maria DB 사용
  - ✓ 데이터를 수정하는 메서드를 GoodHibernateService 인터페이스에 선언

```
public void updateGood();
```

Hibernate

- Maria DB 사용
  - ✓ 데이터를 수정하는 메서드를 GoodHibernateServiceImpl 클래스에 구현

```
@Override
@Transactional
public void updateGood() {
    Good good = new Good();
    good.setCode(101);
    good.setName("낙지");
    good.setManufacture("영암");
    good.setPrice(10000);
    goodHibernateRepository.updateGood(good);
}
```

Hibernate

- Maria DB 사용
  - ✓ 프로젝트를 실행 시킬 main 메서드를 수정

```
public static void main(String[] args) {
    GenericXmlApplicationContext context =
            new GenericXmlApplicationContext(
                    "classpath:applicationContext.xml");
    GoodHibernateService goodService = context.getBean(GoodHibernateService.class);
    goodService.updateGood();
    context.close();
}
```

Hibernate

- Maria DB 사용
  - ✓ 데이터를 삭제하는 메서드를 GoodHibernateRepository 클래스에 구현

```
public void deleteGood(int code) {
    Session session = sessionFactory.getCurrentSession();
    Good vo = new Good();
    vo.setCode(code);
    session.delete(vo);
}
```

Hibernate

- Maria DB 사용
  - ✓ 데이터를 삭제하는 메서드를 GoodHibernateService 인터페이스에 선언

```
public void deleteGood();
```

Hibernate

- Maria DB 사용
  - ✓ 데이터를 삭제하는 메서드를 GoodHibernateServiceImpl 클래스에 구현

```
@Override
@Transactional
public void deleteGood() {
    goodHibernateRepository.deleteGood(101);
}
```

Hibernate

- Maria DB 사용
  - ✓ 프로젝트를 실행 시킬 main 메서드를 수정

```
public static void main(String[] args) {
    GenericXmlApplicationContext context =
            new GenericXmlApplicationContext(
                    "classpath:applicationContext.xml");
    GoodHibernateService goodService = context.getBean(GoodHibernateService.class);
    goodService.deleteGood();
    context.close();
}
```

Hibernate

- Oracle 사용
  - ✓ 샘플 데이터 생성

```
DROP TABLE goods;

create table goods (
    code      number(5) not null,
    name      varchar2(50) not null,
    manufacture    varchar2(20),
    price     number(10) not null,
    shelflife DATE,
    primary key(code)
);


INSERT INTO goods VALUES(1, '사과', '대구', '3000', sysdate);
INSERT INTO goods VALUES(2, '배', '나주', '5000', sysdate);

COMMIT;

select * from goods;
```

Hibernate

- Oracle 사용
  - ✓ applicationContext.xml 파일에 tx Namespace를 추가하고 Hibernate 사용을 위한 설정과 트랜잭션 사용을 위한 bean 설정 태그 추가

```
<bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
value="net.sf.log4jdbc.sql.jdbcapi.DriverSpy"
                />
        <property name="url"
value="jdbc:log4jdbc:oracle:thin:@localhost:1521:xe"/>
        <property name="username" value="system" />
        <property name="password" value="oracle" />
</bean>
```

Hibernate

- Oracle 사용
  - ✓ applicationContext.xml 파일에 tx Namespace를 추가하고 Hibernate 사용을 위한 설정과 트랜잭션 사용을 위한 bean 설정 태그 추가

```
<bean id="sessionFactory"
        class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="annotatedClasses">
                <list>
                        <value>springdb.hibernate.domain.Good</value>
                </list>
        </property>
        <property name="hibernateProperties">
                <value>

        hibernate.dialect=org.hibernate.dialect.Oracle10gDialect
                </value>
        </property>
</bean>
```

Hibernate

- 자바 환경 설정 파일 이용
  - ✓ HibernateConfig.java 를 생성하고 작성

```
@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages = { "springdb.hibernate" })
public class HibernateConfig {
    @Bean
    public DataSource dataSource() {
            DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("net.sf.log4jdbc.sql.jdbcapi.DriverSpy");
        dataSource.setUrl("jdbc:log4jdbc:oracle:thin:@localhost:1521:xe");
        dataSource.setUsername("system");
        dataSource.setPassword("oracle");
            return dataSource;
    }
```

Hibernate

- 자바 환경 설정 파일 이용
  - ✓ HibernateConfig.java 를 생성하고 작성

```
@Bean
public LocalSessionFactoryBean sessionFactory() throws Exception{
        LocalSessionFactoryBean localSessionFactoryBean =  new
LocalSessionFactoryBean();
        localSessionFactoryBean.setDataSource(dataSource());
        localSessionFactoryBean.setAnnotatedClasses(Good.class);
        Properties props = new Properties();
    props.setProperty("hibernate.dialect", "org.hibernate.dialect.Oracle10gDialect");
    localSessionFactoryBean.setHibernateProperties(props);

        return localSessionFactoryBean;
}

@Bean
public HibernateTransactionManager transactionManager() throws Exception{
        HibernateTransactionManager hibernateTransactionManager = new
HibernateTransactionManager();
        hibernateTransactionManager.setDataSource(dataSource());
        hibernateTransactionManager.setSessionFactory(sessionFactory().getObject());
        return hibernateTransactionManager;
}
}
```

Hibernate

- 자바 환경 설정 파일 이용
  - ✓ main 메서드를 수정하고 실행

```
public class HibernateMain {
    public static void main(String[] args) {
            AnnotationConfigApplicationContext context =
                                        new
    AnnotationConfigApplicationContext(HibernateConfig.class);
            GoodHibernateService goodService =
    context.getBean(GoodHibernateService.class);
            goodService.listGood();
            context.close();
    }
}
```