

제 7 강 메이븐 프로젝트 : 스프링 JDBC 심화

1. 스프링에서의 데이터베이스 연동지원

① 템플릿 클래스를 통한 데이터 접근 지원

- ☐ 일반적인 JDBC 코딩은 Connection을 생성하고 PreparedStatement, ResultSet 그리고 자원의 반환하는 코드는 거의 모든 JDBC 코드에서 중복된다.
- ☐ 스프링 데이터베이스 연동을 위한 **템플릿 클래스를 제공함으로써 이러한 중복된 코드를 개발자가 입력해야 하는 작업을 줄일 수 있도록 하고 있다.**
(ex) JdbcTemplate, NamedParameterJdbcTemplate

② 의미 있는 예외 클래스 제공

- ☐ 스프링은 데이터베이스 처리과정에서 발생한 예외가 왜 발생했는지 좀 더 구체적으로 확인할 수 있도록 데이터베이스 처리와 관련된 예외 클래스를 제공한다.
- ☐ 예를 들어 JdbcTemplate 클래스는 처리과정에서 SQLException이 발생하면 스프링이 제공하는 예외 클래스 중 알맞은 예외 클래스로 변환해서 발생 시킨다.
- ☐ 스프링이 제공하는 데이터베이스 관련 예외 클래스는 모두 DataAccessException 클래스를 상속받는데 이는 RuntimeException 이다.

③ 트랜잭션 처리

- ☐ 스프링에서는 세 가지 방법으로 제공하고 자세한 내용은 다음 장에서 설명하도록 하겠다. 참고로 **스프링에서 제공하는 어떤 연동방식을 사용해도 내부적으로 동일하게 적용된다.**

2. DataAccessException

- ☐ 일관된 예외처리를 하기 위함.
 - ☐ SQLException이나 HibernateException 등과 같은 특정 기술에 의존적인 예외를 던지지 않는다.
 - ☐ 모든 DAO Exception은 org.springframework.dao.DataAccessException의 서브 클래스
 - ☐ 데이터 접근 인터페이스가 구현에 의존적인 예외가 아닌 스프링의 일반적인 예외를 던짐으로서, 특정한 퍼시스턴스 구현에 결합되는 일을 방지(loose coupling)
 - ☐ RuntimeException이기 때문에 비 검사 예외(unchecked exception)에 속한다.
 - DAO 계층에서 던진 예외를 코드에서 반드시 처리하지 않아도 된다는 의미
 - 검사 예외(checkedException)는 과도한 catch나 throw 절을 야기 시켜 코드를 난잡하게 만들 수 있다는 스프링의 일반적인 철학을 따르는 것
 - ☐ NestedRuntimeException의 서브클래스이다.
 - 항상 NestedRuntimeException의 getCause() 메소드를 통해 루트 예외인 Exception을 사용할 수 있다는 의미
 - ☐ Spring은 DAO지원에 관한 특정 기술에 대한 예외들을 인식하며, 각 데이터베이스 제품의 에러 코드까지 인식 가능
 - ☐ 자신의 예외 분류 체계에 있는 좀 더 특정한 예외중의 하나를 사용하여 다시 던질 수 있다.
- ① leanupFailureDataAccessException
 - ☐ 성공적으로 작동한 뒤, 데이터베이스 자원을 반환할 때 발생하는 경우(Connection의 반환 등)
 - ② DataAccessResourceFailureException
 - ☐ 데이터베이스로의 연결 실패 등 완전하게 자원 접근에 실패했을 경우

- ③ `DataIntegrityViolationException`
 - ☐ 유일키 제약(unique constraint) 위반과 같은 정합성 위반이 삽입(insert)나 갱신(update)의 결과로 발생한 경우
- ④ `DataRetrievalFailureException`
 - ☐ 기본키(primary key)로 레코드를 찾지 못하는 경우 등과 같이 어떤 데이터를 가져오지 못하는 경우
- ⑤ `DeadlockLoserDataAccessException`
 - ☐ 현재의 프로세스가 교착상태(deadlock)에 빠진 경우
- ⑥ `IncorrectUpdateSemanticsDataAccessException`
 - ☐ 원하는 수보다 많은 레코드를 갱신하는 등 갱신 작업에 있어서 의도하지 않은 어떤 일이 발생하는 경우. 트랜잭션이 롤백(rollback)되지는 않는다.
- ⑦ `InvalidDataAccessApiUsageException`
 - ☐ 실행되기 전에 컴파일 되어야 할 질의문의 컴파일에 실패하는 등 데이터 접근 자바 API를 잘못 사용하는 경우
- ⑧ `InvalidDataAccessResourceUsageException`
 - ☐ 관계형 데이터베이스 접근을 위해 잘못된 SQL 문법을 사용하는 등 자원에 대한 접근을 잘못하는 경우
- ⑨ `OptimisticLockingFailureException`
 - ☐ 낙관적 잠금(optimistic locking)에 실패한 경우. 이 예외는 ORM 툴이나 커스텀 DAO 구현체에 의해 발생된다.
 - ☐ 낙관적 잠금(optimistic locking)
 - ☒ 동일한 데이터에 정확히 동시에 접근하는 일이 발생할 확률이 매우 낮을 것이라고 낙관하여 배타적 잠금을 하지 않는, 즉 데이터에 대한 동시 접근을 허용하는 정책
- ⑩ `TypeMismatchDataAccessException`
 - ☐ String을 데이터베이스의 숫자 컬럼에 삽입하려고 하는 등 자바 타입과 데이터 타입이 일치하지 않는 경우
- ⑪ `UncategorizedDataAccessException`
 - ☐ 무엇인가 잘못됐지만 좀 더 구체적으로 판별할 수 없는 경우

3. DataSource 설정

① 컨텍스트 풀을 이용한 DataSource 설정

- ☐ DBCP가 제공하는 `BasicDataSource` 클래스를 이용해서 DataSource를 설정하는 예

```
<bean id="dataSource" class="org.apache.common.dbcp2.BasicDataSource"
      destroy-method="close" p:driverClassName="oracle.jdbc.driver.OracleDriver"
      p:url="jdbc:oracle:thin:@localhost:1521/XEPDB1" p:username="mytest" p:password="mytest" />
```

- ☐ `BasicDataSource` 클래스의 주요 프로퍼티

프로퍼티	설명
initialSize	초기에 풀에 생성되는 커넥션 수
maxTotal	커넥션 풀이 제공할 최대 커넥션 수
maxIdle	사용되지 않고 풀에 저장될 수 있는 최대 커넥션 수 음수일 경우 제한이 없다는 의미

minIdle	사용되지 않고 풀에 저장될 수 있는 최소 커넥션 수
maxWaitMillis	풀이 커넥션이 존재하지 않을 때 커넥션이 다시 풀에 리턴 될 때 까지 대기하는 시간. 단위 1/1000초
minEvictableIdleTimeMillis	사용되지 않는 커넥션을 추출할 때 이 속성에서 지정한 시간이상 비활성화 상태인 커넥션만 추출한다.
timeBetweenEvictionRunsMills	사용되지 않는 커넥션을 추출하는 스레드의 실행 주기를 지정한다.
numTestsPerEvictionRun	사용되지 않는 커넥션을 몇 개 검사할지를 지정
testOnBorrow	true일 경우 커넥션 풀에서 커넥션을 가져올 때 커넥션이 유효한지 여부를 검사한다.
testOnReturn	true일 경우 커넥션 풀에 커넥션을 반환할 때 커넥션이 유효한지 여부를 검사한다.
testWhileIdle	true일 경우 비활성화 커넥션을 추출할 때 커넥션이 유효한지 여부를 검사해서 유효하지 않은 커넥션은 풀에서 제거한다.

- ☐ DBCP API는 <http://commons.apache.org/dbcp/configuration.html> 참조
- ☐ DBCP와 함께 널리 사용되는 c3p0 라이브러리를 이용해서도 DataSource를 설정할 수 있다.
<http://www.mchage.com/projects/c3p0> 사이트 참조

② JNDI를 이용한 DataSource 설정

- ☐ WebLogic이나 JBoss와 같은 J2EE 어플리케이션 서버를 사용할 경우에 JNDI를 이용해서 DataSource를 구할 수 있다.
- ☐ JNDI를 이용하려면 jee 네임스페이스를 추가하고 <jee:jndi-lookup> 태그를 이용하면 된다.

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/myOracle" resource-ref="true" />
```

- ☐ <jee:jndi-lookup> 태그의 jndi-name 속성은 JNDI에서 객체를 검색할 때 사용할 이름을 입력하고, resource-ref 속성의 값이 true일 경우 검색할 때 이름 앞에 "java:comp/env"가 붙는다.
- ☐ <jee:jndi-lookup> 태그를 이용하지 않고 아래와 같이 JndiObjectFactoryBean 클래스를 이용해서 JNDI로부터 DataSource를 구할 수도 있다.

```
1 <bean id="dataSource" class ="org.springframework.jndi.JndiObjectFactoryBean">
2     <property name="jndiName" value="jdbc/myOracle"/>
3     <property name="resourceRef" value="true" />
4 </bean>
```

- ☐ <jee:jndi-lookup> 태그는 내부적으로 JndiObjectFactoryBean 클래스를 사용한다.

③ DriverManager를 이용한 DataSource 설정

- ☐ DriverManager를 이용해서 커넥션을 제공하는 DriverManagerDataSource 클래스를 사용할 수 있다.

```
1 <bean id="dataSource" class ="org.springframework.jdbc.datasource.DriverManagerDataSource"
2     p:driverClass="oracle.jdbc.driver.OracleDriver" p:url="jdbc:oracle:thin:@localhost:1521/XEPDB1"
3     p:username="mytest" p:password="mytest" />
```

④ DataSource로부터 Connection 구하기

- 스프링은 JDBC를 위한 템플릿 클래스와 DAO지원 클래스를 제공함으로 개발자가 직접 Connection을 이용해서 프로그래밍 하는 경우는 드물다.
- 하지만 DataSource로부터 Connection을 직접 구해야 하는 경우 DataSource의 getConnection() 메서드를 사용해서 Connection을 구할 수 있다.

```
1 public class MyDAO implements SuperDAO{
2     private DataSource dataSource;
3     public void setDataSource(DataSource dataSource){
4         this.dataSource = dataSource;
5     }
6     public void myMethod(){
7         Connction conn = null;
8         try{
9             conn = dataSource.getConnection();
10            ...
11        }
12    }
```

4. 스프링 JDBC 지원

- 스프링에서는 JDBC 지원을 위해 다음과 같은 클래스를 제공하고 있다.

① JdbcTemplate

: 기본적인 JDBC 템플릿 클래스로서 JDBC를 이용해서 데이터에 대한 접근을 제공한다.

② NamedParameterJdbcTemplate

: PreparedStatement에서 인덱스 기반의 파라미터가 아닌 이름을 가진 파라미터를 사용할 수 있도록 지원하는 템플릿 클래스

③ SimpleJdbcTemplate

: 자바5의 가변 인자를 이용해서 쿼리를 실행할 때 사용되는 데이터를 전달할 수 있는 템플릿 클래스. Deprecated 됨.

④ SimpleJdbcInsert

: 데이터 삽입을 위한 인터페이스를 제공해 주는 클래스

⑤ SimpleJdbcCall

: 프로시저 호출을 위한 인터페이스를 제공해 주는 클래스

※ 참고 : 스프링 3.0 버전에서는 SimpleJdbcTemplate를 사용하라고 추천하였으나 3.1 버전부터는 SimpleJdbcTemplate를 Deprecated 하고 대신에 NamedParameterJdbcTemplate를 사용하라고 추천하고 있다.

5. JdbcTemplate 클래스를 이용한 JDBC 프로그래밍

- JdbcTemplate 클래스를 이용하려면 객체를 생성할 때 DataSource를 전달해 주면 된다.

1	public class MyJdbcTemplateDao implements MyDao{
2	private JdbcTemplate jdbcTemplate;
3	public MyJdbcTemplateDao(DataSource dataSource){
4	jdbcTemplate = new JdbcTemplate(dataSource);
5	}
6	}

☐ 위 클래스에 대한 설정파일은 아래와 같을 것이다.

1	<bean id="dataSource"
	class="org.apache.common.dbcp.BasicDataSource"
	destroy-method="close" p:driverClassName="oracle.jdbc.driver.OracleDriver"
	p:url="jdbc:oracle:thin:@localhost:1521/XEPDB1"
	p:username="mytest" p:password="mytest" />
2	<bean id="myDao" class="tommy.spring.dao.MyJdbcTemplateDao">
3	<constructor-arg><ref bean="dataSource"/></constructor-arg>
4	</bean>

☐ 또는 아래와 같이 JdbcTemplate 클래스를 전달받도록 구현할 수도 있다.

1	public class MyJdbcTemplateDao implements MyDao{
2	private JdbcTemplate jdbcTemplate;
3	public void setJdbcTemplate(JdbcTemplate jdbcTemplate){
4	this.jdbcTemplate = jdbcTemplate;
5	}
6	}

☐ 위 클래스에 대한 설정파일은 아래와 같을 것이다.

1	<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" ... />
2	<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"
	p:dataSource-ref="dataSource" />
3	<bean id="myDao" class="com.lsj.spring.dao.MyJdbcTemplateDao"
	p:jdbcTemplate-ref="jdbcTemplate"/>

① 조회를 위한 메서드

☐ 쿼리 실행 결과를 객체 목록으로 가져올 때는 RowMapper를 이용하는 query() 메서드를 이용하면 된다.

- ▣ List<T> query(String sql, Rowmapper<T> rowMapper)
- ▣ List<T> query(String sql, Object[] args, RowMapper<T> rowMapper)
- ▣ List<T> query(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper)

☐ sql 파라미터는 실행할 쿼리, RowMapper는 실행 결과를 자바 객체로 변환해주는 매퍼를 args 파라미터는 PreparedStatement를 실행할 때 사용할 바인딩 값 목록, argTypes는 바인딩할 때 사용할

SQL 타입 목록을 의미한다.

☐ argTypes에 사용되는 값은 java.sql.Types에 정의된 값을 사용한다.

☐ RowMapper의 mapRow() 메서드는 ResultSet에서 읽어온 값을 이용해서 원하는 타입의 객체를 생성한 뒤 리턴 한다. 이때 rowNum은 행 번호를 의미하며 0부터 시작.

```
public interface RowMapper<T>{
    T mapRow(ResultSet rs, int rowNum) throws SQLException;
}
```

☐ query() 메서드의 사용 예

```
1 public List<BoardVO> select(int begin, int end){
2     int startRowNum = begin - 1;
3     int count = end - begin + 1;
4     return jdbcTemplate.query(
5         "select * from (select rownum rnum, num, writer, email, subject, pass," +
6         " regdate, readcount, ref, step, depth, content, ip " +
7         "from (select * from board order by ref desc, step asc))" +
8         " where rnum>=? and rnum<=?",
9         new Object[] {startRowNum, count},
10        new RowMapper<BoardVO>(){
11            @Override
12            public BoardVO mapRow(ResultSet rs, int rowNum) throws SQLException{
13                BoardVO article = new BoardVO();
14                article.setId(rs.getInt("id"));
15                article.setWriter(rs.getString("writer"));
16                ...
17                return article;
18            }
19        });
20 }
```

☐ 만약 쿼리 실행 결과로 읽어온 컬럼의 개수가 한 개라면 queryForObject() 메서드를 이용해서 데이터를 조회할 수 있다.

☒ List<T> queryForList(String sql, Class<T> elementType)

☒ List<T> queryForList(String sql, Object[] args, Class<T> elementType)

☒ List<T> queryForList(String sql, Object[] args, int[] argTypes, Class<T> elementType)

☐ queryForObject() 메서드의 사용 예

```
1 List<String> names = jdbcTemplate.queryForList(
    "select name from member where id=?", new Object[]{"javaline"}, String.class);
```

☐ 쿼리 실행 결과 행의 개수가 한 개인 경우에는 queryForObject() 메서드를 이용해서 실행결과를 가져올 수 있다.

☒ T queryForObject(String sql, RowMapper<T> rowMapper)

- T queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)
- T queryForObject(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper)
- T queryForObject(String sql, Class<T> requiredTypes)
- T queryForObject(String sql, Object[] args, Class<T> requiredTypes)
- T queryForObject(String sql, Object[] args, int[] argTypes, Class<T> requiredTypes)

□ 주의 : 만약 실행결과 행의 개수가 한 개가 아니면 즉 행의 개수가 0이거나 2이상이라면 `IncorrectResultSizeDataAccessException`을 발생 시킨다.

□ queryForObject() 메서드 사용 예

```

1 public int count(){
2     return jdbcTemplate.queryForObject("select count(*) from board", Integer.class);
3 }

```

② 삽입 수정을 위한 메서드 : update()

□ insert, update, delete를 수행할 때 update() 메서드를 이용하면 된다.

- int update(String sql)
- int update(String sql, Object ... args)
- int update(String sql, Object[] args, int[] argTypes)

□ update() 메서드는 쿼리 실행 결과 변경된 행의 개수를 리턴 한다.

□ update() 메서드의 사용 예

```

1 public int insert(MemberVO vo){
2     int count = jdbcTemplate.update(
3         "insert into member(id, pass, name) values(?, ?, ?)",
4         vo.getId(), vo.getPass(), vo.getName());
5     ...
6     return count;
7 }

```

③ ConnectionCallback을 이용한 Connection 사용

□ Connection을 직접 사용해야 한다면 execute() 메서드를 이용하면 된다. : 자주 사용안함.

```

1 public int count(){
2     return jdbcTemplate.execute(new ConnectionCallback<Integer>(){
3         @Override
4         public Integer doInConnection(Connection con)
5             throws SQLException, DataAccessException{
6             Statement stmt = null;
7             ResultSet rs = null;
8             try{
9                 stmt = con.createStatement();
10                rs = stmt.executeQuery("select count(*) from board");

```

```

10         rs.next();
11         return rs.getInt(1);
12     }finally{
13         JdbcUtils.closeResultSet(rs);
14         JdbcUtils.closeStatement(stmt);
15     }
16 }
17 });
18 }

```

6. JdbcTemplate 클래스를 사용한 실습

- ① SpringJDBC 라는 메이븐 프로젝트를 생성한다.
- ② pom.xml 에 메이븐 설정을 추가한다.

```

<!-- 상단 부분 생략 -->
1      <build>
2          <plugins>
3              <plugin>
4                  <groupId>org.apache.maven.plugins</groupId>
5                  <artifactId>maven-compiler-plugin</artifactId>
6                  <version>3.6.2</version>
7                  <configuration>
8                      <source>1.8</source>
9                      <target>1.8</target>
10                     <encoding>UTF-8</encoding>
11                 </configuration>
12             </plugin>
13         </plugins>
14     </build>
15 </project>

```

- ③ 기본적인 스프링 의존관계를 추가한다.

```

<!-- 상단 부분 생략 -->
1      <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
2      <dependency>
3          <groupId>org.springframework</groupId>
4          <artifactId>spring-context</artifactId>
5          <version>5.1.9.RELEASE</version>
6      </dependency>
7      <!-- https://mvnrepository.com/artifact/commons-logging/commons-logging -->
8      <dependency>
9          <groupId>commons-logging</groupId>
10         <artifactId>commons-logging</artifactId>
11         <version>1.2</version>
12     </dependency>
<!-- 하단 부분 생략 -->

```


④ Spring 데이터베이스 관련 의존관계를 추가한다.

```

1      <!-- 상단 부분 생략 -->
2      <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
3      <dependency>
4          <groupId>org.springframework</groupId>
5          <artifactId>spring-jdbc</artifactId>
6          <version>5.1.9.RELEASE</version>
7      </dependency>
8      <!-- https://mvnrepository.com/artifact/org.apache.commons/commons-dbcp2 -->
9      <dependency>
10         <groupId>org.apache.commons</groupId>
11         <artifactId>commons-dbcp2</artifactId>
12         <version>2.7.0</version>
13     </dependency>
14     <!-- 오라클 드라이버 -->
15     <dependency>
16         <groupId>com.oracle</groupId>
17         <artifactId>ojdbc8</artifactId>
18         <version>18.3</version>
19     </dependency>
20 <!-- 하단 부분 생략 -->

```

⑤ AOP 관련 의존관계를 추가한다.

```

1      <!-- 상단 부분 생략 -->
2      <!-- https://mvnrepository.com/artifact/aspectj/aspectjrt -->
3      <dependency>
4          <groupId>aspectj</groupId>
5          <artifactId>aspectjrt</artifactId>
6          <version>1.9.4</version>
7      </dependency>
8      <!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
9      <dependency>
10         <groupId>org.aspectj</groupId>
11         <artifactId>aspectjweaver</artifactId>
12         <version>1.9.4</version>
13     </dependency>
14 <!-- 하단 부분 생략 -->

```

⑥ 실습용 데이터베이스 테이블을 작성한다.

```

1 CREATE TABLE "GUESTBOOK" (
2     "MESSAGE_ID" NUMBER(5,0),
3     "GUEST_NAME" VARCHAR2(20) DEFAULT null,
4     "MESSAGE" VARCHAR2(4000),
5     "REGISTRY_DATE" DATE DEFAULT null,
6     CONSTRAINT "GUESTBOOK_PK" PRIMARY KEY ("MESSAGE_ID")
7 );

```

- ☐ GuestBook 테이블에서 사용할 sequence를 작성한다.

1	CREATE SEQUENCE "GUEST_SEQ" MINVALUE 1 MAXVALUE 99999
2	INCREMENT BY 1 START WITH 1 NOCACHE NOORDER NOCYCLE;

⑦ Model 클래스 작성

- ☐ V0에 해당하는 GuestMessage 클래스 작성

1	package tommy.spring.guestbook.vo;
2	import java.util.Date;
3	public class GuestMessage {
4	private Integer id;
5	private String guestName;
6	private String message;
7	private Date registryDate;
	//getter, setter 추가
8	}

- ☐ V0의 목록을 나타내는 GuestMessageList 클래스 작성

1	package tommy.spring.guestbook.vo;
2	import java.util.List;
3	public class GuestMessageList {
4	private int totalCount;
5	private int pageNum;
6	private int begin;
7	private int end;
8	private List<GuestMessage> messages;
9	public GuestMessageList(int totalCount, int pageNum, int begin, int end,
	List<GuestMessage> messages) {
10	this.totalCount = totalCount;
11	this.pageNum = pageNum;
12	this.begin = begin;
13	this.end = end;
14	this.messages = messages;
15	}
	//getter, setter 추가
16	}

⑧ DAO 관련 클래스 및 인터페이스 작성

- ☐ DAO 기능을 추상화한 GuestMessageDAO 인터페이스 작성

1	package tommy.spring.guestbook.dao;
2	import java.util.List;
3	import tommy.spring.guestbook.vo.GuestMessage;
4	public interface GuestMessageDao {
5	public int count();

6	public List<GuestMessage> select(int begin, int end);
7	public int insert(GuestMessage message);
8	public int delete(int id);
9	public int update(GuestMessage message);
10	}

- 목록 조회 시 사용할 RowMapper 클래스 작성.

```

1 package tommy.spring.guestbook.dao;
2 import java.sql.ResultSet;
3 import java.sql.SQLException;
4 import org.springframework.jdbc.core.RowMapper;
5 import tommy.spring.guestbook.vo.GuestMessage;
6 public class GuestMessageRowMapper implements RowMapper<GuestMessage> {
7     @Override
8     public GuestMessage mapRow(ResultSet rs, int rowNum) throws SQLException {
9         GuestMessage message = new GuestMessage();
10        message.setId(rs.getInt("MESSAGE_ID"));
11        message.setGuestName(rs.getString("GUEST_NAME"));
12        message.setMessage(rs.getString("MESSAGE"));
13        message.setRegistryDate(rs.getDate("REGISTRY_DATE"));
14        return message;
15    }
16 }

```

☐ GuestMessageDAO를 상속받아 실제적인 데이터베이스 처리를 담당할 구현 클래스 작성

```

1 package tommy.spring.guestbook.dao;
2 import java.sql.Types;
3 import java.util.List;
4 import org.springframework.jdbc.core.JdbcTemplate;
5 import tommy.spring.guestbook.vo.GuestMessage;
6 public class JdbcTemplateGuestMessageDao implements GuestMessageDao {
7     private JdbcTemplate jdbcTemplate;
8     public JdbcTemplateGuestMessageDao(JdbcTemplate jdbcTemplate) {
9         this.jdbcTemplate = jdbcTemplate;
10    }
11    public int count() {
12        return jdbcTemplate.queryForObject("select count(*) from GUESTBOOK",
13                                           Integer.class);
14    }
15    public int delete(int id) {
16        return jdbcTemplate.update(
17            "delete from GUESTBOOK where MESSAGE_ID = ?", id);
18    }
19    public int insert(final GuestMessage message) {
20        int insertedCount = jdbcTemplate.update(
21            "insert into GUESTBOOK(MESSAGE_ID, GUEST_NAME, " +
22            "MESSAGE, REGISTRY_DATE) values (guest_seq.nextval, ?, ?, ?)",

```

```

21         message.getGuestName(), message.getMessage(),
22         message.getRegistryDate());
23     return insertedCount;
24 }
25 public List<GuestMessage> select(int begin, int end) {
26     return jdbcTemplate.query(
27         "select * from (select ROWNUM rnum, MESSAGE_ID, GUEST_NAME,"
28         + "MESSAGE, REGISTRY_DATE "
29         + "from (select * from GUESTBOOK order by MESSAGE_ID desc)) "
30         + "where rnum>=? and rnum<=?",
31         new Object[] { begin, end },
32         new GuestMessageRowMapper());
33 }
34 public int update(GuestMessage message) {
35     return jdbcTemplate.update("update GUESTBOOK "
36         + "set MESSAGE = ? where MESSAGE_ID = ?",
37         new Object[] { message.getMessage(), message.getId() },
38         new int[] { Types.VARCHAR, Types.INTEGER });
39 }

```

- ☐ 원래는 Service 관련 인터페이스와 클래스를 작성해야 하나 JDBC 처리 과정을 살펴보는 것이 목적
이므로 생략 하도록 하겠다.

⑨ 스프링 설정 파일 작성하기

- ☐ **src/main/resources**라는 폴더를 생성하고 **applicationContext.xml** 파일을 작성하자.
(bean, aop, p, jdbc 네임스페이스 체크)

```

<!-- 상단 부분 생략 -->
1     <bean id="dataSource"
2         class="org.apache.commons.dbcp2.BasicDataSource"
3         destroy-method="close"
4         p:driverClassName="oracle.jdbc.driver.OracleDriver"
5         p:url="jdbc:oracle:thin:@localhost:1521/XEPDB1" p:username="mytest"
6         p:password="mytest" />
7     <bean id="jdbcTemplate"
8         class="org.springframework.jdbc.core.JdbcTemplate"
9         p:dataSource-ref="dataSource" />
10    <bean id="jdbcTemplateGuestMessageDao"
11        class="tommy.spring.guestbook.dao.JdbcTemplateGuestMessageDao">
12        <constructor-arg>
13            <ref bean="jdbcTemplate" />
14        </constructor-arg>
15    </bean>
<!-- 하단 부분 생략 -->

```

⑩ 테스트를 위한 메인클래스 작성하기.

1	package tommy.spring.guestbook.controller;
2	import java.util.Date;
3	import java.util.List;
4	import org.springframework.context.support.AbstractApplicationContext;
5	import org.springframework.context.support.ClassPathXmlApplicationContext;
6	import tommy.spring.guestbook.dao.JdbcTemplateGuestMessageDao;
7	import tommy.spring.guestbook.vo.GuestMessage;
8	public class JdbcTemplateMain {
9	private GuestMessage makeGuestMessage(String guestName, String message) {
10	GuestMessage msg = new GuestMessage();
11	msg.setGuestName(guestName);
12	msg.setMessage(message);
13	msg.setRegistryDate(new Date());
14	return msg;
15	}
16	public static void main(String[] args) {
17	String[] configLocations = new String[] { "applicationContext.xml" };
18	AbstractApplicationContext context =
	new ClassPathXmlApplicationContext(configLocations);
19	JdbcTemplateGuestMessageDao dao = (JdbcTemplateGuestMessageDao)
	context.getBean("jdbcTemplateGuestMessageDao");
20	JdbcTemplateMain myTest = new JdbcTemplateMain();
21	dao.insert(myTest.makeGuestMessage("이승재", "하이방가"));
22	dao.insert(myTest.makeGuestMessage("javaline", "Hi, Hello"));
23	dao.insert(myTest.makeGuestMessage("Spring", "안녕하세요?"));
24	int count = dao.count();
25	System.out.println("전체글수 : " + count);
26	List<GuestMessage> list = dao.select(1, count);
27	for (GuestMessage msg : list) {
28	System.out.println(msg.getId() + " : " + msg.getGuestName() + " : " +
	msg.getMessage() + " : " + msg.getRegistryDate());
29	}
30	context.close();
31	}
32	}

⑪ 실행 결과 확인하기.

```

<terminated> JdbcTemplateMain [Java Application]
전체글수 : 3
3 : Spring : 안녕하세요? : 2019-09-15
2 : javaline : Hi, Hello : 2019-09-15
1 : 이승재 : 하이방가 : 2019-09-15

```

7. NamedParameterJdbcTemplate 클래스를 활용한 JDBC 프로그래밍

- ☐ JdbcTemplate 클래스와 비슷한 기능을 제공하는데 차이점은 인덱스 기반의 파라미터가 아니라 이름 기반의 파라미터를 설정할 수 있도록 해준다는 점이다.
- ☐ 마찬가지로 JdbcTemplate 클래스와 동일한 이름의 메서드를 제공하는데 단지 이름 기반의 파라미터 값을 설정하기 위하여 Map이나 SqlParameterSource를 전달받는다.

```
1 public int delete(int id){
2     Map<String, Object> paramMap = new HashMap<String, Object>();
3     paramMap.put("id", id);
4     return template.update("delete from board where board_id = :id", paramMap);
5 }
```

① Map을 이용한 파라미터 값 설정 메서드

- ☐ List<T> query(String sql, Map<String, ?> paramMap, RowMapper<T> rowMapper)
- ☐ List<T> queryForList(String sql, Map<String, ?> paramMap, Class<T> elementType)
- ☐ T queryForObject(String sql, Map<String, ?> paramMap, RowMapper<T> rowMapper)
- ☐ T queryForObject(String sql, Map<String, ?> paramMap, Class<T> requiredType)
- ☐ int update(String sql, Map<String, ?> paramMap)

☐ query 메서드의 사용 예

```
1 public List<GuestMessage> select(int begin, int end){
2     Map<String, Object> paramMap = new HashMap<String, Object>();
3     paramMap.put("startRowNum", begin);
4     paramMap.put("count", end-begin+1);
5     return template.query(
6         "select * from (select ROWNUM rnum, MESSAGE_ID, GUEST_NAME, " +
7         "MESSAGE, REGISTRY_DATE from (select * from GUESTBOOK" +
8         "order by MESSAGE_ID desc)) where rnum>=:startRowNum and rnum<=:count",
9         paramMap, new RowMapper<GuestMessage>(){
10             @Override
11             public GuestMessage mapRow<ResultSet rs, int rowNum>throws SQLException{
12                 GuestMessage message = new GuestMessage();
13                 message.setId(rs.getInt("MESSAGE_ID"));
14                 message.setGuestName(rs.getString("GUEST_NAME"));
15                 message.setMessage(rs.getString("MESSAGE"));
16                 message.setRegistryDate(rs.getDate("REGISTRY_DATE"));
17                 return message;
18             }
19         });
20 }
```

- ☐ 만약 이름 기반의 파라미터를 갖지 않는 쿼리를 실행할 경우에는 아무 값도 갖지 않는 Map 객체를 사용하면 된다.

```
int id = template.queryForInt("select count(*) from member",
    Collections.<String, Object> emptyMap());
```

② SqlParameterSource를 이용한 파라미터 값 설정 메서드

- ☐ List<T> query(String sql, SqlParameterSource paramSource, RowMapper<T> rowMapper);
 - ☐ List<T> query(String sql, SqlParameterSource paramSource, Class<T> elementType);
 - ☐ T queryForObject(String sql, SqlParameterSource paramSource, RowMapper<T> rowMapper);
 - ☐ T queryForObject(String sql, SqlParameterSource paramSource, Class<T> requiredType);
 - ☐ int update(String sql, SqlParameterSource paramSource);
- ☐ SqlParameterSource는 인터페이스이기 때문에 실제로 사용할 때는 SqlParameterSource 인터페이스를 구현한 클래스를 사용해서 파라미터 값을 전달해 주어야 한다.
- ▣ org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource
 - ▣ org.springframework.jdbc.core.namedparam.MapSqlParameterSource

```
1 public int insert(GuestMessage message){
2     BeanPropertySqlParameterSource paramSource =
3         new BeanPropertySqlParameterSource(message);
4     int insertCount = template.update("insert into GUESTBOOK " +
5         "(GUEST_NAME, MESSAGE, REGISTRY_DATE) " +
6         "values(:guestName, :message, :registryDate)", paramSource);
7     ...
8     return insertCount;
9 }

10 public int insert(GuestMessage message){
11     MapSqlParameterSource paramSource = new MapSqlParameterSource();
12     paramSource.addValue("message", message.getMessage());
13     paramSource.addValue("id", message.getId(), Types.INTEGER);
14     return template.update("update GUESTBOOK set MESSAGE = :message "
15         + "where MESSAGE_ID = :id", paramSource);
16 }
```

8. NamedParameterJdbcTemplate을 이용한 프로그래밍 실습

- ☐ 앞에서 실습했던 JdbcTemplate을 이용한 예제를 그대로 바꾸어 보겠다.
- ☐ 따라서 기본적인 모델과 서비스는 그대로 이용하고 DAO만 변경해 보도록 하자.

① NamedParameterJdbcTemplate을 이용한 DAO 클래스작성

```
1 package tommy.spring.guestbook.dao;
2 import java.sql.ResultSet;
3 import java.sql.SQLException;
4 import java.sql.Types;
5 import java.util.Collections;
6 import java.util.HashMap;
```

```

7 import java.util.List;
8 import java.util.Map;
9 import org.springframework.jdbc.core.RowMapper;
10 import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
11 import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
12 import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
13 import tommy.spring.guestbook.vo.GuestMessage;
14 public class NamedParamGuestMessageDao implements GuestMessageDao {
15     private NamedParameterJdbcTemplate template;
16     public void setTemplate(NamedParameterJdbcTemplate template) {
17         this.template = template;
18     }
19     @Override
20     public int count() {
21         Map<String, Object> paramMap = new HashMap<String, Object>();
22         return template.queryForObject("select count(*) from GUESTBOOK",
23                                         paramMap, Integer.class);
24     }
25     @Override
26     public int delete(int id) {
27         Map<String, Object> paramMap = new HashMap<String, Object>();
28         paramMap.put("id", id);
29         return template.update("delete from GUESTBOOK where MESSAGE_ID = :id",
30                               paramMap);
31     }
32     @Override
33     public int insert(GuestMessage message) {
34         BeanPropertySqlParameterSource paramSource =
35             new BeanPropertySqlParameterSource(message);
36         int insertedCount = template.update("insert into GUESTBOOK "
37             + "(MESSAGE_ID, GUEST_NAME, MESSAGE, REGISTRY_DATE)"
38             + " values(guest_seq.nextval, :guestName, :message, :registryDate)",
39             paramSource);
40         if (insertedCount > 0) {
41             int id = template.queryForObject("select count(*) from GUESTBOOK",
42                 Collections.<String, Object>emptyMap(),
43                 Integer.class);
44             message.setId(id);
45         }
46         return insertedCount;
47     }
48     @Override
49     public List<GuestMessage> select(int begin, int end) {
50         Map<String, Object> paramMap = new HashMap<String, Object>();
51         paramMap.put("startRowNum", begin);
52         paramMap.put("count", end - begin + 1);
53         return template.query(
54             "select * from (select ROWNUM rnum, MESSAGE_ID, "
55             + "GUEST_NAME, MESSAGE, REGISTRY_DATE from "

```



```

52         + "(select * from GUESTBOOK order by MESSAGE_ID desc)) "
53         + "where num>=:startRowNum and num<=:count",
54         paramMap, new RowMapper<GuestMessage>() {
55             @Override
56             public GuestMessage mapRow(ResultSet rs, int rowNum)
57                                     throws SQLException {
58                 GuestMessage message = new GuestMessage();
59                 message.setMsg(rs.getInt("MESSAGE_ID"));
60                 message.setGuestName(rs.getString("GUEST_NAME"));
61                 message.setMessage(rs.getString("MESSAGE"));
62                 message.setRegistryDate(rs.getDate("REGISTRY_DATE"));
63                 return message;
64             }
65         });
66     @Override
67     public int update(GuestMessage message) {
68         MapSqlParameterSource paramSource = new MapSqlParameterSource();
69         paramSource.addValue("message", message.getMessage());
70         paramSource.addValue("id", message.getId(), Types.INTEGER);
71         return template.update("update GUESTBOOK set MESSAGE = :message"
72                               + " where MESSAGE_ID = :id", paramSource);
73     }
74 }

```

② 스프링 설정파일에 빈을 등록하자.

```

1 <!-- 상단 부분 생략 -->
2 <bean id="namedParameterJdbcTemplate"
3     class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
4     <constructor-arg>
5         <ref bean="dataSource"/>
6     </constructor-arg>
7 </bean>
8 <bean id="namedParamGuestMessageDao"
9     class="tommy.spring.guestbook.dao.NamedParamGuestMessageDao">
10     <property name="template" ref="namedParameterJdbcTemplate" />
11 </bean>

```

③ 테스트용 메인 클래스를 작성하자.

```

1 package tommy.spring.guestbook.controller;
2 import java.util.Date;
3 import java.util.List;
4 import org.springframework.context.support.AbstractApplicationContext;
5 import org.springframework.context.support.ClassPathXmlApplicationContext;
6 import tommy.spring.guestbook.dao.NamedParamGuestMessageDao;
7 import tommy.spring.guestbook.vo.GuestMessage;
8 public class NamedParameterMain {

```

```

9      private GuestMessage makeGuestMessage(String guestName, String message) {
10          GuestMessage msg = new GuestMessage();
11          msg.setGuestName(guestName);
12          msg.setMessage(message);
13          msg.setRegistryDate(new Date());
14          return msg;
15      }
16      public static void main(String[] args) {
17          String[] configLocations = new String[] { "applicationContext.xml" };
18          AbstractApplicationContext context =
19              new ClassPathXmlApplicationContext(configLocations);
20          NamedParamGuestMessageDao dao = (NamedParamGuestMessageDao)
21              context.getBean("namedParamGuestMessageDao");
22          NamedParameterMain myTest = new NamedParameterMain();
23          dao.insert(myTest.makeGuestMessage("이승재", "방가방가"));
24          dao.insert(myTest.makeGuestMessage("javaline", "Hi, Hello"));
25          dao.insert(myTest.makeGuestMessage("Spring", "안녕하세요^^"));
26          int count = dao.count();
27          System.out.println("전체글수 : " + count);
28          List<GuestMessage> list = dao.select(1, count);
29          for (GuestMessage msg : list) {
30              System.out.println(msg.getId() + " : " + msg.getGuestName() + " : " +
31                  msg.getMessage() + " : " + msg.getRegistryDate());
32          }
33          context.close();
34      }
35  }

```

④ 실행 및 결과 확인

□ 참고 : 원래 글이 6개만 들어가야 하나 실수로 두 번 실행하여 9개가 되었다.



```

<terminated> NamedParameterMain [Java Applicat
전체글수 : 9
9 : Spring : 안녕하세요^^ : 2019-09-15
8 : javaline : Hi, Hello : 2019-09-15
7 : 이승재 : 방가방가 : 2019-09-15
6 : Spring : 안녕하세요^^ : 2019-09-15
5 : javaline : Hi, Hello : 2019-09-15
4 : 이승재 : 방가방가 : 2019-09-15
3 : Spring : 안녕하세요? : 2019-09-15
2 : javaline : Hi, Hello : 2019-09-15
1 : 이승재 : 하이방가 : 2019-09-15

```

9. SimpleJdbcInsert 클래스를 이용한 데이터 삽입

□ SimpleJdbcInsert 클래스는 SQL 문을 사용하지 않고 데이터를 삽입할 수 있도록 해 주는 클래스이다.

```

1 public class NamedParameterGuestMessageDao implements GuestMessageDao{
2     private SimpleJdbcInsert insertMessage;
3     public NamedParameterGuestMessageDao (DataSource dataSource){
4         insertMessage = new SimpleJdbcInsert(dataSource);
5         insertMessage.withTableName("GUESTBOOK");
6         insertMessage.usingColumns("GUEST_NAME", "MESSAGE", "REGISTRY_DATE");
7     }
8     @Override
9     public int insert(GuestMessage message){
10         Map<String, Object> paramValue = new HashMap<String, Object>();
11         paramValue.put("GUEST_NAME", message.getGuestName());
12         paramValue.put("MESSAGE", message.getMessage());
13         paramValue.put("REGISTRY_DATE", message.getRegistryDate());
14         return insertMessage.execute(paramValue);
15     }
16 }
17 }

```

- ☐ withTableName() 메서드는 데이터를 삽입할 테이블의 이름을 지정한다.
- ☐ execute(Map<String, Object>) 메서드는 Map의 키를 컬럼으로 사용하고, 값을 컬럼에 삽입할 데이터로 사용하는 SQL 쿼리를 실행한다.
- ☐ withColumns() 메서드를 사용하면 쿼리를 생성할 때 사용할 컬럼을 직접 지정할 수 있다. 이 때 **주의할 점은 지정한 컬럼에 대해서만 값이 삽입된다는 점이다.**
- ☐ SimpleJdbcInsert 클래스가 제공하는 설정메서드는 체이닝을 지원한다.

① execute() 메서드를 이용한 데이터 삽입

- ☐ int execute(Map<String, Object> args)
- ☐ int execute(SqlParameterSource parameterSource)
- ☐ 컬럼명의 일치여부 검사
 - ☒ 지정한 컬럼명과 동일한 이름을 갖는 파라미터 값이 설정되어 있는지 검사한다
 - ☒ “_” 이 포함된 경우 “_” 를 제외한 나머지 문자열과 일치하는 파라미터 값이 설정되어 있는지 검사한다.
- ☐ 참고 : 실제로 Map, SqlParameterSource와 컬럼명 사이의 매핑처리는 TableMetaDataContext 클래스를 통해서 처리한다. (API 참조)

② executeAndReturnKey() 메서드를 이용한 데이터 삽입 및 자동 생성키 조회

- ☐ Number executeAndReturnKey(Map<String, Object> args)
- ☐ Number executeAndReturnKey(SqlParameterSource paramSource)
- ☐ KeyHolder executeAndReturnKeyHolder(Map<String, Object> args)
- ☐ KeyHolder executeAndReturnKeyHolder(SqlParameterSource paramSource)
- ☐ executeAndReturnKey() 메서드를 사용하려면 usingGeneratedKeyColumns() 메서드를 이용해서 자동 생성되는 키 컬럼을 지정해 주면 된다.

```

insertMessage.withTableName("GUESTBOOK_MESSAGE").usingGeneratedKeyColumns("MESSAGE_ID")
...

```

- 자동생성 키 칼럼을 지정했다면 아래와 같이 조회할 수 있다.

```
Number keyValue = insertMessage.executeAndReturnKey(parameterSource);
message.setIdx(keyValue.intValue());

KeyHolder keyHolder = insertMessage.executeAndReturnKeyHolder(paramSource);
message.setIdx(keyHolder.getKey().intValue());
```

③ NamedParameterGuestMessageDao 클래스 수정

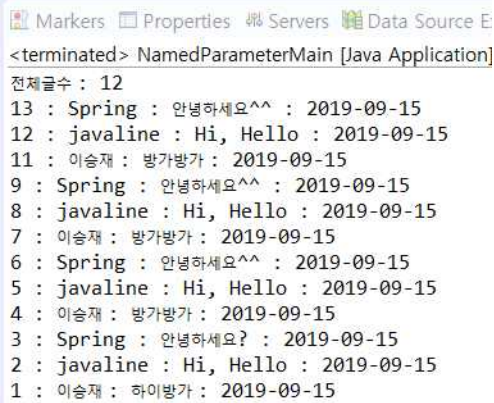
```
1 public class NamedParameterGuestMessageDao implements GuestMessageDao{
2     private SimpleJdbcInsert insertMessage;
3     public NamedParameterGuestMessageDao(SimpleJdbcInsert insertMessage){
4         this.insertMessage = insertMessage;
5         insertMessage.withTableName("GUESTBOOK");
6         insertMessage.usingColumns(
7             "MESSAGE_ID", "GUEST_NAME", "MESSAGE", "REGISTRY_DATE");
8     }
9     public int nextVal() {
10         Map<String, Object> paramMap = new HashMap<String, Object>();
11         return template.queryForObject(
12             "select guest_seq.nextval from dual", paramMap, Integer.class);
13     }
14     <!-- 나머지 생략 -->
15     @Override
16     public int insert(GuestMessage message){
17         MapSqlParameterSource paramSource = new MapSqlParameterSource();
18         message.setIdx(nextVal());
19         paramSource.addValue("MESSAGE_ID", message.getId());
20         paramSource.addValue("GUEST_NAME", message.getGuestName());
21         paramSource.addValue("MESSAGE", message.getMessage());
22         paramSource.addValue("REGISTRY_DATE", message.getRegistryDate());
23         return insertMessage.execute(paramSource);
24     }
```

④ 스프링 설정파일에 SimpleJdbcInsert를 빈으로 등록하자.

```
<!-- 상단 부분 생략 -->
1 <bean id="simpleJdbcInsert" class="org.springframework.jdbc.core.simple.SimpleJdbcInsert">
2     <constructor-arg>
3         <ref bean="dataSource"/>
4     </constructor-arg>
5 </bean>
6 <bean id="namedParamGuestMessageDao"
7     class="tommy.spring.guestbook.dao.NamedParamGuestMessageDao">
8     <constructor-arg>
9         <ref bean="simpleJdbcInsert"/>
10    </constructor-arg>
```

10	<property name="template" ref="namedParameterJdbcTemplate" />
11	</bean>
	<!-- 하단 부분 생략 -->

⑤ NamedParameterMain 클래스를 실행하여 결과를 확인하자.



```

<terminated> NamedParameterMain [Java Application]
전제글수 : 12
13 : Spring : 안녕하세요^^ : 2019-09-15
12 : javaline : Hi, Hello : 2019-09-15
11 : 이승재 : 방가방가 : 2019-09-15
9 : Spring : 안녕하세요^^ : 2019-09-15
8 : javaline : Hi, Hello : 2019-09-15
7 : 이승재 : 방가방가 : 2019-09-15
6 : Spring : 안녕하세요^^ : 2019-09-15
5 : javaline : Hi, Hello : 2019-09-15
4 : 이승재 : 방가방가 : 2019-09-15
3 : Spring : 안녕하세요? : 2019-09-15
2 : javaline : Hi, Hello : 2019-09-15
1 : 이승재 : 하이방가 : 2019-09-15

```