Spring Framework

Spring DataBase Integration

❑ **About**

- Spring 에서 DataSource를 활용하는 방법에 대한 문서
- iBatis & MyBatis 등 DataMapper 에 대한 기본 지식을 갖추었음을 가정함

❑ **차례**

대덕인재개발원 : 최희연

❑ JDBC

- Java에서 JDBC는 중요한 부분을 차지한다. JDBC API를 직접 이용하여 데이터베이스에 접근하거나 또는 MyBatis, Hibernate, JPA 같은 ORM 프레임워크와 연동하여 데이터베이스에 접근할 수 있다.
- Spring 역시 개발자가 JDBC를 이용하거나 ORM과 연동하기 위한 방법을 제공해 준다.
- JDBC로 작업은 전통적으로 동일한 작업의 반복, 자원들의 반납등 노가다성 작업 ????
- Spring 역시 이러한 부분을 줄이기 위해 템플릿 클래스를 제공하고 있으며, 개발자는 템플릿 클래스를 사용하여 중복되는 코드를 줄일 수 있다. 또한 ORM연동에서도 마찬가지로 템플릿 클래스를 제공한다.
  - 템플릿 클래스를 통한 데이터 접근 지원
  - 의미있는 예외 클래스 제공
  - 트랜잭션 처리
- 대부분의 JDBC API 관련 클래스의 메서드는 항상 SQLException 예외를 처리해야 한다.
- Spring은 데이터베이스 처리과정에서 발생한 예외를 한단계 더 추상화하여 좀더 의미있는 예외를 throw 한다.
- Spring의 데이터베이스 관련 예외클래스는 모두 DataAccessException클래스를 상속받는다.
- DataAccessException은 RuntimeException이므로 필요한 경우에만 try~catch 로 처리하면 된다.

❑ DataSource

- 대부분의 어플리케이션은 데이터베이스와 연동을 위해 커넥션을 획득해야 하는데, JDBC 프로그래밍에서 커넥션을 확보하는 방법은 두가지다. Driver 이용한 커넥션 획득과 DataSource를 이용한 커넥션 획득 두가지 방식이 있는데, Spring은 DataSource 를 통해서 데이터베이스 연결을 위한 Connection을 제공한다.
- 스프링은 다양한 방식의 데이터베이스 연결을 제공하고,이에 대한 추상화된 계층을 제공함으로써, 비즈니스 로직과 데이터베이스 연결 방식 간의 종속성을 배제한다.
- Connection Provider별 Connection 객체를 얻기 위한 로직을 구현한 DataSource 구현체를 사용한다.

❑ JDBCDataSource

- JDBC driver를 이용하여 Database Connection을 생성한다.
- 커넥션 풀이나 JNDI를 사용할 수 없는 경우 DriverManager를 사용해서 커넥션을 구한다.
  - 설정 예시

```
<context:property-placeholder location="classpath:jdbc.properties"/>

<bean id="dataSource"
       class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="${maindb.driverClassName}" />
  <property name="url" value="${maindb.url}" />
  <property name="username" value="${maindb.username}" />
  <property name="password" value="${maindb.password}" />
</bean>
```

Properties	설 명
driverClassName	JDBC driver class name설정
url	DataBase에 접근하기 위한 JDBC URL
username	DataBase 접근하기 위한 사용자명
password	DataBase 접근하기 위한 암호

❑ DBCPDataSource

- Commons DBCP라 불리는 Jakarta의 Database Connection Pool의 pooling DataSource 를 이용해 풀링 커넥션을 사용하기도 한다.
  - 설정 예시 : Commons-dbcp 의존성 추가 필요.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <property name="driverClassName" value="${maindb.driverClassName}"/>
  <property name="url" value="${maindb.url}"/>
  <property name="username" value="${maindb.username}"/>
  <property name="password" value="${maindb.password}"/>
  <property name="defaultAutoCommit" value="false"/>
  <property name="poolPreparedStatements" value="true"/>
</bean>
```

Properties	설 명
driverClassName	jdbc driver의 class name 설정
url	DataBase url을 설정
username	DataBase 접근하기 위한 사용자명
password	DataBase 접근하기 위한 암호
defaultAutoCommit	datasource로부터 리턴된 connection에 대한 auto-commit 여부를 설정
poolPreparedStatements	PreparedStatement 사용여부
maxActive	동시에 할당할 수 있는 active connection 최대 갯수를 설정
maxIdle	pool에 남겨놓을 수 있는 idle connection 최대 갯수를 설정
maxWait	모든 Connection이 사용중일 경우 최대 대기 시간을 설정
defaultReadOnly	Connection Pool에 의해 생성된 Connection에 read-only 속성 부여
defaultTransactionIsolation	리턴된 connection에 대한 transaction isolation 속성 부여
defaultCatalog	Connection의 catlog 설정
minIdle	Connection pool의 최소한 idle connection 갯수 설정
initialSize	Connection pool에 생성될 초기 connection size 설정
testOnBorrow	Connection pool에서 객체를 가지고 오기 전에 그 객체의 유효성을 확인할 것인지 결정
testOnReturn	객체를 return하기 전에 객체의 유효성을 확인할 것인지 결정
validationQuery	validationQuery를 설정
loginTimeout	Database에 연결하기 위한 login timeout(in seconds)을 설정

❑ C3P0DataSource

- JDBC driver를 이용한 DataBase Connection를 생성하는 구현체. C3P0 Library에 관련 사항은 C3P0 Configuration 에서 확인할 수 있다.
  - 설정 예시

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
  destroy-method="close">
  <property name="driverClass" value="${driver}" />
  <property name="jdbcUrl" value="${dburl}" />
  <property name="user" value="${username}" />
  <property name="password" value="${password}" />
  <property name="initialPoolSize" value="3" />
  <property name="minPoolSize" value="3" />
  <property name="maxPoolSize" value="50" />
  <!-- <property name="timeout" value="0" /> -->   <!-- 0 means: no timeout -->
  <property name="idleConnectionTestPeriod" value="200" />
  <property name="acquireIncrement" value="1" />
  <property name="maxStatements" value="0" />  <!-- 0 means: statement caching is
turned off.  -->
  <property name="numHelperThreads" value="3" />  <!-- 3 is default -->
</bean>
```

Properties	설 명
driverClass	JDBC driver class name설정
jdbcUrl	DB URL
user	사용자명
password	패스워드
initialPoolSize	풀 초기값
minPoolSize	풀 최소값
maxPoolSize	풀 최대값
idleConnectionTestPeriod	idle상태 점검시간
acquireIncrement	증가값
maxStatements	캐쉬유지여부
numHelperThreads	HelperThread 개수

❑ JNDIDataSource

– JNDIDataSource는 JNDI Lookup을 이용하여 Database Connection을 생성한다. JNDIDataSource는 대부분 Enterprise application server에서 제공되는 JNDI tree로 부터 DataSource를 가져온다.

- 설정 예시
- 톰캣 예시 : 다음 페이지에서 톰캣을 이용한 DataSouce JNDI 설정을 정리한다.

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/mainDB"
       cache="true" lookup-on-startup="true"
       expected-type="javax.sql.DataSource" />
```

- Jeus 설정

```
<jee:jndi-lookup id="dataSource" jndi-name="${jndiName}" resource-ref="true">
   <jee:environment>
     java.naming.factory.initial=${jeus.java.naming.factory.initial}
     java.naming.provider.url=${jeus.java.naming.provider.url}
   </jee:environment>
</jee:jndi-lookup>
```

- Weblogic 설정

```
<util:properties id="jndiProperties"
                 location="classpath:/META-INF/spring/jndi.properties" />

<jee:jndi-lookup id="dataSource" jndi-name="${jndiName}"
                 resource-ref="true" environment-ref="jndiProperties" />
```

- JndiObjectFactoryBean을 직접 사용

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean"
       p:jndiName="jdbc/mainDB"
       p:lookupOnStartup="true" p:cache="true" p:resourceRef="true"
       p:expectedType="javax.sql.DataSource"
/>
```

Properties	설 명
jndiTemplate	JNDI 검색을 위해 JNDI 템플릿을 설정
jndiEnvironment	JNDI를 검색하기 위해 JNDI 환경을 설정
resourceRef	J2EE 컨테이너에서 검색할 수 있는지 설정
expectedType	JNDI 객체의 타입을 지정
jndiName	검색을 위해 JNDI 이름을 설정
proxyInterface	JNDI 객체를 사용하기 위해 proxy 인터페이스를 설정
lookupOnStartup	starup시에 JNDI object를 검색할 지 여부를 설정
cache	JNDI 객체를 캐싱할 것인지 설정
defaultObject	JNDI lookup에 실패하였을 경우 전달할 default object를 지정

❑ DataSource JNDI 설정 방법.

1,. GlobalResource 등록(naming) 방법 - dataSource 를 서버에서 일괄 관리하는 경우.

1) catalina.properties 에 dataSource 생성 프로퍼티 등록.

```
mainDB.driverClassName=oracle.jdbc.driver.OracleDriver
mainDB.url=jdbc:oracle:thin:@localhost:1521:xe
mainDB.username=sem
mainDB.password=java
```

2) servet.xml 에 GlobalResources 등록.

참고 : tomcat7 부터 DataSource 생성 Factory 로 commons-dbcp 를 사용하지 않고, toomcat-jdbc.jar 내에 있는 Factory 를 사용함.

```
<GlobalNamingResources>
        <Resource
                auth="Container"
                factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
                driverClassName="${mainDB.driverClassName}"
                name="jdbc/mainDB"
                password="${mainDB.password}"
                type="javax.sql.DataSource"
                url="${mainDB.url}"
                username="${mainDB.username}" />
</GlobalNamingResources>
```

3) context.xml 에 ResourceLink 등록.

```
<Context>
    <WatchedResource>WEB-INF/web.xml</WatchedResource>
    <ResourceLink name="jdbc/mainDB"
            global="jdbc/mainDB"
            type="javax.sql.DataSource" />
</Context>
```

2. 어플리케이션별로 별도 관리하는 dataSource 등록(naming) 방법.

web content 영역(context root 아래 META-INF/context.xml 생성 후 Resource 등록.

```
<Context>
        <Resource
                auth="Application"
                driverClassName="${mainDB.driverClassName}"
                 factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
                name="jdbc/mainDB"
                type="javax.sql.DataSource"
                url="${mainDB.url}"
                username="${mainDB.username}"
                password="${mainDB.password}"
        />
</Context>
```

- 참고사이트
  - http://tomcat.apache.org/tomcat-7.0-doc/config/globalresources.html
  - http://tomcat.apache.org/tomcat-7.0-doc/config/context.html#Resource_Definitions
  - http://tomcat.apache.org/tomcat-7.0-doc/jdbc-pool.html

❑ Database programming Template

- JDBC 프로그래밍을 위한 반복 작업을 줄여주고, 결과 집합을 객체 형태로 변환하는 작업을 해주는 템플릿 클래스

❑ JdbcTemplate

- dbcp pooling dataSource 등록

```
<context:component-scan base-package="kr.or.ddit" />

<context:property-placeholder
		location="classpath:/kr/or/ddit/database_info.properties" />

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
	p:driverClassName="${maindb.driverClassName}"
	p:url="${maindb.url}"
	p:username="${maindb.username}"
	p:password="${maindb.password}"
	p:initialSize="${maindb.initialSize}"
	p:maxWait="${maindb.maxWait}"
/>
```

- DataSource 를 주입받아 template으로 쿼리를 실행할 DAO 작성

```
@Repository
public class CommonDAO {
	JdbcTemplate template;
	@Inject
	public void setDataSource(DataSource dataSource) {
		this.template = new JdbcTemplate(dataSource);
	}

	public List<Map> retrieveProps(Object...params){
		StringBuffer sql = new StringBuffer("SELECT * FROM DATABASE_PROPERTIES");
		return template.query(sql.toString(), params, new RowMapper<Map>(){
			@Override
			public Map mapRow(ResultSet rs, int rowNum) throws SQLException {
				Map<String, Object> hashMap = new HashMap<>();
				ResultSetMetaData meta = rs.getMetaData();
				for(int i=1; i<=meta.getColumnCount(); i++){
					hashMap.put(meta.getColumnLabel(i), rs.getObject(i));
				}
				return hashMap;
			}
		});
	}
}
```

❑ JdbcTemplate(계속)

- SELECT

```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor", Integer.class);
```

```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(
        "select count(*) from t_actor where first_name = ?", Integer.class, "Joe");
```

```
String lastName = this.jdbcTemplate.queryForObject(
        "select last_name from t_actor where id = ?",
        new Object[]{1212L}, String.class);
```

```
Actor actor = this.jdbcTemplate.queryForObject(
        "select first_name, last_name from t_actor where id = ?",
        new Object[]{1212L},
        new RowMapper<Actor>() {
            public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
                Actor actor = new Actor();
                actor.setFirstName(rs.getString("first_name"));
                actor.setLastName(rs.getString("last_name"));
                return actor;
            }
        });
```

```
List<Actor> actors = this.jdbcTemplate.query(
        "select first_name, last_name from t_actor",
        new RowMapper<Actor>() {
            public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
                Actor actor = new Actor();
                actor.setFirstName(rs.getString("first_name"));
                actor.setLastName(rs.getString("last_name"));
                return actor;
            }
        });
```

- UPDATE, INSERT, DELETE

```
this.jdbcTemplate.update(
        "insert into t_actor (first_name, last_name) values (?, ?)",
        "Leonor", "Watling");
```

```
this.jdbcTemplate.update(
        "update t_actor set last_name = ? where id = ?",
        "Banjo", 5276L);
```

```
this.jdbcTemplate.update(
        "delete from actor where id = ?",
        Long.valueOf(actorId));
```

❑ NamedParameterJdbcTemplate

- JdbcTemplate 과 동일한 기능을 갖지만, 인덱스 기반의 파라미터가 아니라 이름을 기반으로 한 파라미터라서 잘못된 파라미터를 사용하는 오류를 줄여줄 수 있다.
- NamedParamterJdbcTemplate 는 맵으로 쿼리 파라미터를 넘기면서 바인딩 기호 ':'를 사용한다.

```
@Repository
public class CommonDAO {
	NamedParameterJdbcTemplate namedTemplate;

	@Inject
	public void setDataSource(DataSource dataSource) {
		this.namedTemplate = new NamedParameterJdbcTemplate(dataSource);
	}

	public Map retrieveMember(String mem_id){
		StringBuffer sql =
				new StringBuffer("SELECT * FROM MEMBER WHERE MEM_ID = :mem_id");
		return namedTemplate.queryForMap(sql.toString(),
					Collections.singletonMap("mem_id", mem_id));

	}
}
```

- 이름과 값의 쌍을 이룰수 있는 Map이나 자바빈 기반으로 파라미터를 전달하는 경우, SqlParameterSource 구현체를 사용하기도 한다.

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name = :first_name";

    SqlParameterSource namedParameters = new MapSqlParameterSource("first_name", firstName);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);
}
```

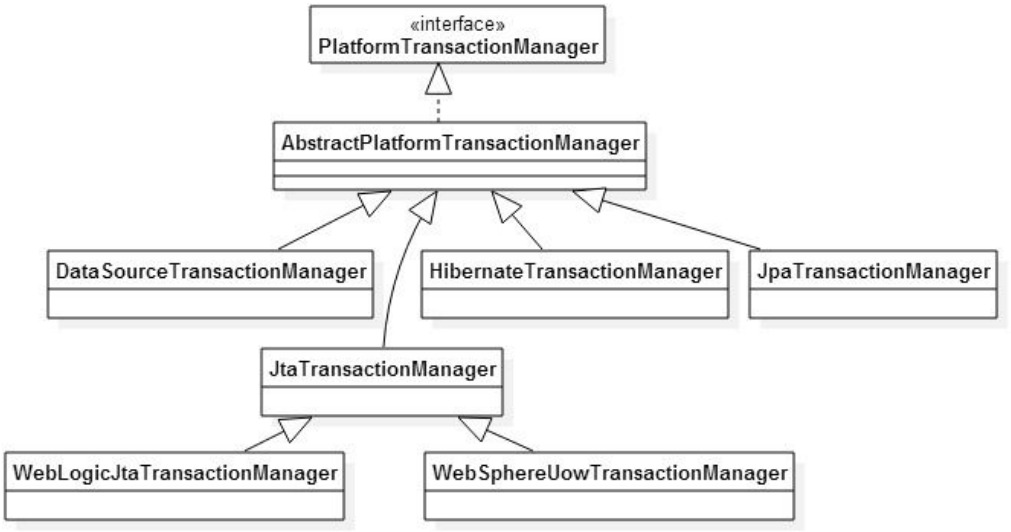
```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActors(Actor exampleActor) {

    // notice how the named parameters match the properties of the above Actor class
    String sql = "select count(*) from T_ACTOR where first_name = :firstName and last_name = :lastName";

    SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(exampleActor);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);
}
```

❑ Transaction

트랜잭션은 여러 단계의 과정을 하나의 작업 행위로 묶을 때 사용되는 일종의 작업에 대한 정의

- 원자성(Atomicity)
  트랜잭션은 한 개 이상의 동작을 논리적으로 한 개의 작업 단위(unit)로 묶는다. 원자성은 트랜잭션 범위에 있는 모든 동작이 모두 실행되거나 모두 실행 취소됨을 보장하여, 모든 동작이 성공적으로 실행되면 트랜잭션 성공으로 간주한다. 물론 하나라도 실패하면 트랜잭션은 실패하고 모든 과정은 롤백된다. - All or Nothing
- 일관성(Consistency)
  트랜잭션이 종료되면, 시스템은 비즈니스에서 기대하는 상태가 된다. 예를 들어, 서적 구매 트랜잭션이 성공적으로 실행되면 결제내역, 구매내역, 잔고정보가 비즈니스에 맞게 저장되고 변경된다.
- 고립성(Isolation)
  트랜잭션은 다른 트랜잭션과 독립적으로 실행되어야 하며, 서로 다른 트랜잭션이 동일한 데이터에 동시에 접근할 경우 알맞게 동시 접근을 제어해야 한다(동시 접근 제어는 설정한 격리 레벨에 따라 달라진다).
- 지속성(Durability)
  트랜잭션이 완료되면, 그 결과는 지속적으로 유지되어야 한다. 현재의 어플리케이션이 변경되거나 없어지더라도 데이터는 유지된다. 일반적으로 물리적인 저장소를 통해서 트랜잭션 결과가 저장된다.

❑ PlatformTransactionManager



- 트랜잭션 관리를 위해서는 적절한 PlatformTransactionManager 구현체를 등록해야 한다.

```
<jee:jndi-lookup jndi-name="jdbc/mainDB" id="dataSource" />

<bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource"
/>
```

```
@Bean
public DataSource dataSource(){
        JndiDataSourceLookup dsLookup = new JndiDataSourceLookup();
        dsLookup.setResourceRef(true);
        return dsLookup.getDataSource("jdbc/mainDB");
}
@Bean
public PlatformTransactionManager transactionManager() {
        return new DataSourceTransactionManager(dataSource());
}
```

❑ 트랜잭션 전파 속성

Propagation	설 명
REQUIRED	메소드를 수행하는 데 트랜잭션이 필요하다는 의미. 현재 진행중인 트랜잭션이 존재한다면, 이를 사용한다. 존재하지 않는 경우 새로운 트랜잭션을 생성한다.
MANDATORY	메소드를 수행하는 데 트랜잭션이 필요하지만, REQUIRED 와 다르게 진행중인 트랜잭션이 없다면 예외를 발생시킨다.
REQUIRES_NEW	항상 새로운 트랜잭션을 시작한다. 기존 트랜잭션이 존재하면 일시 중지하고, 새로운 트랜잭션을 시작되고, 종료된 뒤 기존 트랜잭션이 계속된다.
SUPPORTS	메소드가 트랜잭션을 필요로 하지는 않지만, 기존 트랜잭션이 있으면 사용한다. 진행중인 트랜잭션이 없어도 정상적으로 동작한다.
NOT_SUPPORTED	메소드가 트랜잭션을 필요로 하지 않으며, SUPPORT 와 달리 진행중인 트랜잭션이 있다면 메소드가 진행되는 동안 기존 트랜잭션은 일시 중지된다.
NEVER	메소드가 트랜잭션이 필요하지 않으며 기존 트랜잭션이 있다면 예외를 발생시킨다.
NESTED	기존 트랜잭션이 존재하면 거기에 중첩된 트랜잭션에서 메소드를 실행한다. 존재하지 않으면 새로운 트랜잭션을 시작하는데, JDBC 3.0 드라이버를 사용하는 경우에만 지원된다.

❑ 트랜잭션 격리 레벨

Isolation level	설 명
DEFAULT	기본 설정을 사용한다.
READ_UNCOMMITTED	다른 트랜잭션에서 커밋하지 않은 데이터를 읽을 수 있다.
READ_COMMITTED	다른 트랜잭션에 의해 커밋된 데이터를 읽을 수 있다.
REPEATABLE_READ	처음에 읽어온 데이터와 두 번째 읽어온 데이터가 동일한 값을 갖는다.
SERIALIZABLE	동일한 데이터에 대해 동시에 두 개 이상의 트랜잭션은 허용하지 않는다.

❑ 선언적 트랜잭션 처리(Declarative Transaction Managememt)

DTM 은 트랜잭션 처리를 코드에서 직접 수행하지 않고, 설정 파일이나 어노테이션을 이용해 트랜잭션의 전파 범위, 격리 레벨, 롤백 규칙 등을 정의한다.

- <tx:advice>태그를 이용한 설정

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
		<tx:attributes>
			<tx:method name="*" propagation="REQUIRED"/>
			<tx:method name="get*" read-only="true"/>
			<tx:method name="retrieve*" read-only="true"/>
			<tx:method name="select*" read-only="true"/>
		</tx:attributes>
</tx:advice>
<aop:config>
		<aop:advisor advice-ref="txAdvice" pointcut="within(kr.or.ddit..service..*)"/>
</aop:config>
```

- <tx:advice>태그를 이용한 설정 (계속)
  - <tx:method>태그의 속성

속성 이름	설명
name	트랜잭션이 적용될 메소드 이름을 명시한다. '*'를 사용한 설정이 가능한다.
propagation	트랜잭션 전파 규칙을 설정한다. 기본값 : REQUIRED
isolation	트랜잭션 격리 레벨을 설정한다.
read-only	읽기 전용 여부를 설정한다.
no-rollback-for	트랜잭션을 롤백하지 않을 예외의 종류를 지정한다.
rollback-for	트랜잭션을 롤백할 예외를 지정한다. 기본값 : RuntimeException 기본적으로 RuntimeException과 Error에 대해서만 롤백 처리를 하고, Throwable이나 Exception 타입의 예외는 발생하더라도 롤백되지 않고 예외 발생 전까지의 작업이 커밋되기 때문에 정교한 롤백 처리를 위해 rollback-for 속성을 사용한다.
timeout	트랜잭션 타임아웃 시간을 초 단위로 지정한다.

- @Transactional 어노테이션을 이용한 설정

```
public class BookOrderServiceImpl implements IBookOrderService{
	// inject ...
	@Override
	@Transactional(propagation=Propagation.REQUIRED)
	public OrderResult bookOrder(BookOrderVO orderVO){
		// 주문 정보 등록
		String orderId = bookOrderDao.insertOrder(orderVO.getOrderInfo());
		for(Entry<BookVO, Integer> entry : orderVO.getBookList()){
			// 주문아이디와 각 도서별 주문 수량을 가진 객체 생성
			EachBookOrder ebo =
				new EachBookOrder(orderId, entry.getKey(), entry.getValue());
			bookOrderDao.insertOrderedBook(ebo);
		}
		// 결제 정보 등록
		paymentDao.insert(orderVO.getPaymentInfo());
	}
}
```

  - @Transactional 어노테이션 속성

속성 이름	설명
propagation	트랜잭션 전파 규칙으로 기본값은 REQUIRED.
isolation	트랜잭션 격리 레벨
readOnly	읽기 전용 여부
rollbackFor	롤백한 예외 타입. ex) rollbackFor={Exception.class}
noRollbackFor	트랜잭션을 롤백하지 않을 예외 타입. ex) noRollbakcFor={BookNotFoundExcpetion.class}
timeout	초단위의 트랜잭션 타임아웃 시간.

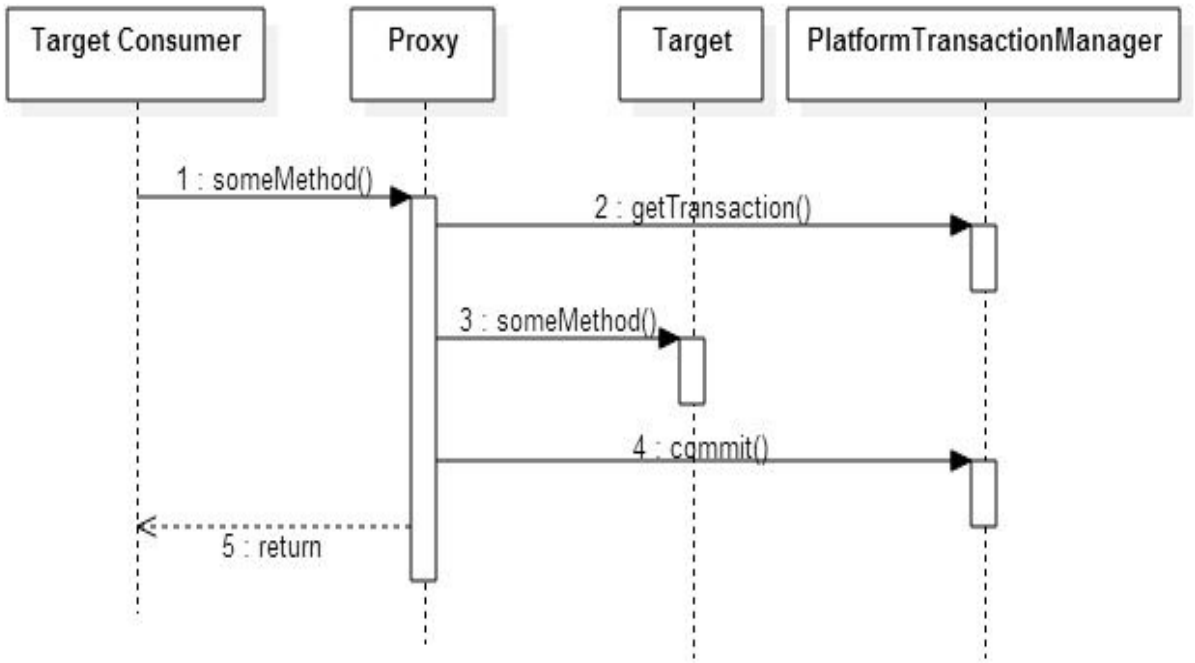
- @Transactional 지원 설정 (XML)

```
<tx:annotation-driven
	transaction-manager="transactionManager"
	proxy-target-class="false"
	mode="proxy"
/>
```

- @Transactional 지원 설정 (Java)

```
@Configuration
@EnableTransactionManagement(mode=AdviceMode.PROXY, proxyTargetClass=false)
public class DataSourceContext implements TransactionManagementConfigurer{

	@Bean
	public DataSource dataSource() {
		JndiDataSourceLookup dsLookup = new JndiDataSourceLookup();
		dsLookup.setResourceRef(true);
		return dsLookup.getDataSource("jdbc/mainDB");
	}

	@Override
	public PlatformTransactionManager annotationDrivenTransactionManager() {
		return new DataSourceTransactionManager(dataSource());
	}
}
```

- 선언적 트랜잭션 관리는 두가지 방식 모두 AOP 프록시를 기반으로 지원되고 있어서, @Transactional 이나 tx 네임스페이스를 이용하면, 트랜잭션을 처리할 메소드를 가진 빈 객체에 대한 프록시를 생성한다. 이 프록시 객체는 PlatformTransactionManager를 이용해서 트랜잭션을 시작한 뒤에, 실제 타겟 객체의 메소드를 호출하고, 그 다음에 PlatformTransactionManager를 이용해서 트랜잭션을 커밋한다.



❑ About

- iBATIS Data Mapper Framework는 DBMS에 접근할 때 필요한 자바코드를 현저하게 줄일 수 있습니다. 자바코드의 20%를 사용하여 JDBC기능의 80%를 제공하는 간단한 프레임워크라는 뜻이다.
- iBATIS는 다른 프레임워크, ORM(Object Relation Mapping)에 비해 가장 큰 장점은 심플함(Simplicity)이다.
- XML 기술을 사용해서 간단하게 JavaBean(또는 Map)객체를 PreparedStatement의 파라미터와 ResultSets으로 쉽게 mapping할 수 있다.
- iBATIS를 Persistant Framework, ORM Tool, SQL Mapper, Data Mapper 등 혼용되어 사용되나 여기서는 Data Mapper 라고 하겠습니다.
- 개발자 문서 및 전자정부 매뉴얼을 정리한 것이므로 개발자 문서를 꼭 참고 하시기 바랍니다.

❑ 참고

- 개발자인 Clinton Begin에 의해 java기반의 iBATIS 프로젝트 시작했으며 Apache 프로젝트로 있다가 2010년 6월 Google Code로 프로젝트 이전, 버전이 3.x 로 업그레이드 하면서 iBatis에서 myBatis로 명칭 변경
- MyBatis
  - http://www.mybatis.org
- iBATIS-SqlMaps-2 개발자 가이드(영문)
  - http://ibatis.apache.org/docs/java/pdf
- iBATIS-SqlMaps-2 개발자 가이드(한글, 이동국님 번역)
  - http://www.kldp.net/projects/kfwdp/download
- Spring Framework - Reference Documentation
  - http://static.springframework.org/spring/docs/2.5.6/reference/orm.html#orm-ibatis

❑ Architecture

- iBATIS Architecture 구성요소

Architecture 구성 요소	설 명
Parameter Object (Input)	파라미터 객체는 JavaBean, Map, Primitive 객체로서, update문 내에 입력값을 셋팅하기 위해 사용되거나 쿼리문의 where절을 셋팅하기 위해 사용된다.
SqlMapConfig.xml	Data Mapper에서 사용하는 설정을 담고 있는 파일로서, DataSource, Data Mapper 및 Thread Management등과 같은 상세 설정 정보를 담고있다.
SqlMap. xml	하나의 SqlMap.xml파일은 많은 CacheMdel, Parameter Maps, Result Maps, Statements 정보를 담고있다.
SQL Map	Data Mapper프레임워크는 PreparedStatement인스턴스를 생성하고 제공된 파라미터 객체를 사용해서 파라미터를 셋팅한다. 그리고 statement를 실행하고 ResultSet으로부터 결과객체를 생성한다.
Mapped Statement	Mapped Statement는 Data Mapper 프레임워크의 핵심으로서, Parameter Maps과 Result Maps를 이용하여 SQL statement로 치환된다.
Result Object (Output)	결과 객체는 JavaBean, Map, Primitive객체로서, 쿼리문의 결과값을 담고 있다.

❑ 단계

- library 설치 : iBATIS를 실행 하기 위한 파일 다운로드 및 설치
- SQL Map Config 파일 작성 : DB 연결정보, 설정정보등을 관리
- SQL Mapping 파일 작성 : 실제 쿼리문을 관리
- SqlMapClient 생성 파일 작성 : SqlMapClientBuilder를 이용 SqlMapClient인스턴스 생성
  (여기서는 Spring ORM 에서 제공하는 `SqlMapClientDaoSupport`를 이용)
- DAO 파일 구현 : dao객체에서 SqlMapClient 실행 API 를 통해 실행

❑ library 설치

- http://www.mybatis.org 또는 http://code.google.com/p/mybatis/
- 현업에서 현재 사용되는 버전은 2.x 이므로 ibatis-2.3.4.726.zip을 다운로드 한다.
- 지정된 CLASSPATH나 웹 어플리케이션의 /WEB-INF/lib 에 jar파일을 배치한다.
- iBATIS는 자체 jar 파일 하나만으로도 실행될 수 있게끔 다른 파일과의 의존성은 없다.
- 만약 성능향상을 위해서 추가적인 라이브러리를 설치할 수 있다.
- 참고로 Spring3.2에서 iBatis 와의 integration 모듈은 deprecated 되었고, spring4 부터는 ORM 모듈에서 완전히 제거되었다.

❑ SQL Map Config 파일

- SQL Map XML Configuration파일은 데이터소스, 데이터 매퍼에 대한 설정, 쓰레드 관리와 같은 SQL Maps와 다른 옵션에 대한 설정을 제공하는 중앙집중적인 XML설정 파일을 사용해서 설정된다. 다음은 SQL Maps설정파일을 작성한다. (sql-map-config-2.dtd)
  - 설정 파일 예시
    iBatis 만을 사용하는 경우, SqlMapConfig 설정에는 데이터소스를 비롯한 트랜잭션 매니저를 등록하여야 한다. 그러나 스프링과 연동하는 경우, 스프링의 플랫폼 기반 트랜잭션 매니저로 트랜잭션 관리 기능을 위임하기 때문에 트랜잭션 매니저나 데이터소스 자체를 iBatis 직접 설정하지 않는다.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
    PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">
<sqlMapConfig>
   <typeAlias alias="member" type="ddit.model.MemberBean" />
   <sqlMap resource="kr/or/ddit/ibatis/mappings/blank.xml" />
</sqlMapConfig>
```

    한가지 특이사항은 원래 sqlMap XML을 ibatis 에 직접 등록해야 사용했지만, 스프링의 Facotry bean을 사용하게 되면, 매핑파일에 대해 ant 스타일의 CoC 설정을 사용할수 있기 때문에 매핑 파일을 직접 등록할 필요도 없어진다.
    그러나, sqlMapConfig 라는 루트엘리먼트는 XML 스키마에 따라 반드시 하나 이상의 sqlMap 요소를 가져야 하기때문에, 매핑 파일을 직접 등록하지 않으려면, 아무런 sqlMap 설정도 갖지않는 가짜 XML을 하나 등록해서 XML 파싱 오류를 피해야만 한다.

- DB 연결정보를 별도의 파일로 작성할 수 있다.

```
# jdbc.properties
maindb.driverClassName=oracle.jdbc.driver.OracleDriver
maindb.url=jdbc:oracle:thin:@127.0.0.1:1521:xe
maindb.username=account
maindb.password=java
```

❑ SQL Mapping 파일

- 실제적인 SQL 구문을 관리하는 sql mapping 파일을 작성한다. (sql-map-2.dtd)
  - 매핑파일 예시

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
    PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="member">
  <!-- // resultClass에 완전한 클래스이름이나 별칭을 사용한다. -->
  <select id="getMemberList" resultClass="member">
    SELECT *
      FROM member
  </select>
  <select id="getMember" parameterClass="string"  resultClass="member">
     ....
  </select>
```

❑ Spring 에서 iBatis 설정

- Spring 프레임워크는 iBATIS SQL Mapper를 위해 template 스타일 프로그래밍이 가능토록 지원한다. 이러한 지원으로 Spring 의 특징인 IoC 의 장점과 Exception 계층 구조의 처리가 iBATIS 통합 환경에서도 쉽게 사용되고 있으며, iBATIS 단독 사용 시에 트랜잭션 관리 및 DataSource 에 대한 설정 및 관리가 별도로 필요했던 것에 비해 Spring-iBATIS 통합 환경에서는 Spring 의 유연한 트랜잭션 처리와 dataSource 를 그대로 사용할 수 있다.
- SqlMapClientFactoryBean 은 iBATIS 의 SqlMapClient 를 생성하는 FactoryBean 구현체로, Spring 의 context 에 iBATIS 의 SqlMapClient 를 설정하는 방식으로 사용되며, 여기서 얻어진 SqlMapClient 는 iBATIS 기반 DAO 에 DI 을 통해 넘겨지거나, POJO 인경우 SqlMapClientTemplate 을 주입하기도 한다.

```
<!-- // 환경변수 설정을 위해 외부 프러퍼티 파일 로드  -->
<context:property-placeholder location="classpath:jdbc.properties"/>

<!-- // DataSource 설정, 여기서는 일반 JDBC 방식으로    -->
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
    <property name="driverClassName" value="${maindb.driverClassName}" />
    <property name="url" value="${maindb.url}" />
    <property name="username" value="${maindb.username}" />
    <property name="password" value="${maindb.password}" />
</bean>
<!-- // iBatis의 SqlMapClient 생성 구현체 SqlMapClientFactoryBean 정의 -->
<bean id="sqlMapClient" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean"
       p:dataSource-ref="dataSource"
       p:configLocation="classpath:kr/or/ddit/ibatis/sqlMapConfig.xml"
       p:mappingLocations="classpath:kr/or/ddit/ibatis/mappings/*.xml"
/>
<bean id="sqlMapClientTemplate"
       class="org.springframework.orm.ibatis.SqlMapClientTemplate"
       p:sqlMapClient-ref="sqlMapClient"
/>
```

  - configLocation 속성에 iBatis 설정 파일의 위치
  - mappingLocations 속성으로 Sql 매핑 파일의 일괄 지정이 가능하다. (iBatis 2.3.2 이상에서)
  - dataSource속성에 커넥션을 얻기위한 빈 정의

- DAO의 구현
  - Spring 의 SqlMapClientDaoSupport 클래스는 iBATIS 의 SqlMapClient data access object 를 위한 편리한 수퍼 클래스로 이를 extends 하는 서브 클래스에 SqlMapClientTemplate 를 제공한다. SqlMapClientTemplate 는 iBATIS 를 통한 data access 를 단순화하는 헬퍼 클래스로 SQLException 을 Spring dao 의 exception Hierarchy 에 맞게 unchecked DataAccessException 으로 변환해 주며 Spring 의 JdbcTemplate 과 동일한 처리 구조의 SQLExceptionTranslator 를 사용할 수 있게 해준다. 또한 iBATIS 의 SqlMapExecutor 의 실행 메서드에 대한 편리한 mirror 메서드를 다양하게 제공하므로 일반적인 쿼리나 insert/update/delete 처리에 대해 편리하게 사용할 수 있도록 권고된다.
  - BATIS 연동 DAO 는 SqlMapClientDaoSupport 를 extends 하고 있으며, getSqlMapClientTemplate() 를 통해 SqlMapClientTemplate 를 얻어 실행하면 된다.

  - SqlMapClientDaoSupport 상속 예시

```
public class BoardDaoIBatis extends SqlMapClientDaoSupport implements IBoardDao{
  @Override
  public List<BoardBean> getBoardList(Map<String, Object> searchMap) {
    return getSqlMapClientTemplate().queryForList("board.getBoardList",searchMap);
  }

  @Override
  public BoardBean getBoard(int bo_no) {
    return (ProdBean)getSqlMapClientTemplate().queryForObject("board.getBoard",bo_no);
  }
}
```

  - POJO에서 SqlMapClientTemplate 을 주입받아 사용하는 예

```
public class BoardDaoIBatis {
  @Autowired
  SqlMapClientTemplate sqlMapClientTemplate;

  @Override
  public List<BoardBean> getBoardList(Map<String, Object> searchMap) {
    return sqlMapClientTemplate.queryForList("board.getBoardList",searchMap);
  }

  @Override
  public BoardBean getBoard(int bo_no) {
    return (ProdBean) sqlMapClientTemplate.queryForObject("board.getBoard",bo_no);
  }
}
```

- 구현체인 ProdDaoiBatis는 스프링에서 iBATIS를 위해 제공하는 SqlMapClientDaoSupport를 상속해서 작성할 것입니다. SqlMapClient를 구하는 것은 getSqlMapClientTemplate()를 통해서 작성하면 됩니다.
  - kr.or.ddit.prod.dao.ProdDaoiBatis.java

```
package kr.or.ddit.prod.dao;

import java.util.List;
import java.util.Map;
import kr.or.ddit.prod.model.ProdBean;
import org.springframework.orm.ibatis.support.SqlMapClientDaoSupport;

public class ProdDaoiBatis extends SqlMapClientDaoSupport implements IProdDao {
  @Override
  public List<ProdBean> getProdList(Map<String, Object> searchMap) {
    return getSqlMapClientTemplate().queryForList("prod.getProdList",searchMap);
  }

  @Override
  public ProdBean getProd(String prod_id) {
    return (ProdBean)getSqlMapClientTemplate().queryForObject("prod.getProd",prod_id);
  }
} // class
```

- iBATIS를 사용하므로 당연히 매핑파일도 작성해야겠군요. (왜 파일수가 자꾸 늘어나 머리 아프게)
  일반적으로 예약어는 대문자로 기술하고 객체는 소문자로 기술하지만 Oracle은 객체를 모두 대문자로 관리하는 관계로 바꾸도록 하겠습니다.
  - kr/or/ddit/ibatis/mapping/prod.xml

```
<sqlMap namespace="prod">
  <typeAlias alias="prodBean" type="kr.or.ddit.prod.model.ProdBean" />

  <select id="getProdList" parameterClass="java.util.Map" resultClass="prodBean">
    select PROD_ID,       PROD_NAME,     PROD_GUBUN,
           (select CODE_NAME from CODE_TB where CODE_ID = PROD_GUBUN) PROD_GUBUN_NAME,
           PROD_BUYER,    PROD_PRICE,    PROD_SALE,
           PROD_OUTLINE,  PROD_IMAGE,
           PROD_UNIT,     PROD_QTY,      PROD_MILEAGE
      from PROD
    <isNotEmpty property="gubun" prepend="where">
        PROD_GUBUN = #prod_gubun:VARCHAR2#
    </isNotEmpty>
  </select>

  <select id="getProd" parameterClass="string" resultClass="prodBean">
    select  PROD_ID,       PROD_NAME,     PROD_GUBUN,
            (select CODE_NAME from CODE_TB where CODE_ID = PROD_GUBUN) PROD_GUBUN_NAME,
            PROD_BUYER,    PROD_PRICE,    PROD_SALE,
            PROD_OUTLINE,  PROD_IMAGE,    PROD_DETAIL,
            PROD_UNIT,     PROD_QTY,      PROD_MILEAGE
      from  PROD
     where  PROD_ID = #prod_id:VARCHAR2#
  </select>
</sqlMap>
```

스프링에서 제공하는 SqlMapClientDaoSupport 또는 SqlMapClientTemplate 이용하지 않고 직접 iBATIS를 연동하면 Connection을 직접 제어해야 합니다. 즉, 비즈니스 로직을 처리할 때 트랜잭션 때문에 서비스 단에서 Connection을 Dao에게 넘겨주어야 하고 또한 상위 비즈니스 로직이 하위 비즈니스 로직을 포함하고 있을때는 아마 환장할겁니다.
(교재 323~327p)

DAO 객체에 SQLException이 안보여요?
SQLException이 발생했을때 어디서, 왜 발생했는지, 또는 catch 블록을 사용했지만 특별한 처리를 하지 않아 서비스단에서 난리가 발생하는등 예외처리하는 것이 까다로웠습니다.
스프링은 데이터베이스 처리과정에서 발생한 예외를 좀 더 구체적으로 만들어놨습니다. 모든 데이터베이스 관련 예외는 DataAccessException을 상속받아서 구현되었습니다. 그리고 RuntimeException입니다.
(교재 305p)

❑ Service 객체 생성

- 서비스객체도 인터페이스를 작성후 구현체를 생성하도록 합니다.
  - kr.ddit.prod.service.IProdService.java

```
package kr.ddit.prod.service;

import java.util.List;
import java.util.Map;
import kr.or.ddit.prod.model.ProdBean;

public interface IProdDao {
  // 전체 또는 분류검색된 상품 목록
  List<ProdBean> getProdList(Map<String, Object> searchMap);
  // 상품 정보
  ProdBean getProd(String prod_id);
}
```

  - kr.or.ddit.prod.service.ProdServiceImpl.java

```
1.  package kr.or.ddit.prod.service;

2.  import java.util.List;
3.  import java.util.Map;
4.  import kr.or.ddit.prod.dao.IProdDao;
5.  import kr.or.ddit.prod.model.ProdBean;

6.  public class ProdServiceImpl implements IProdService {
7.    private IProdDao prodDao;
8.    // 스프링 컨테이너로 부터 prodDao에 대한 의존 빈 주입을 받기 위해서 setter 메서드 생성
9.    public void setProdDao(IProdDao prodDao) {
10.     this.prodDao = prodDao;
11.   }

12.   @Override
13.   public List<ProdBean> getProdList(Map<String, Object> searchMap) throws Exception {
14.     return prodDao.getProdList(searchMap);
15.   }

16.   @Override
17.   public ProdBean getProd(String prod_id) throws Exception {
18.     return prodDao.getProd(prod_id);
19.   }
20. } // class
```

  - 7라인 : 의존하는 객체의 인터페이스 IProdDao 를 맴버변수로 선언한다.
  - 9라인 : 이전에 dao 와 service객체는 명시적으로 스프링 설정파일을 이용하자고 했습니다. 그래서 annotation을 이용하지 않고 설정파일에서 Setter Injection을 통해서 받도록 하기 위해 IProdDao를 받는 setter 메서드를 작성했습니다.
  - 13, 17라인 : dao 객체를 이용해서 비즈니스 로직을 처리합니다.

❑ View 파일 작성

- JSP 를 먼저 배우고 MVC 패턴이니, 프레임워크니 하는 것을 학습하면서 많이들 어려워 하리라 생각합니다. "어디 시 부터 시작해야 하는지", "어떤 파일이 필요한 건지" 등등
- JSP를 먼저 배워서 또는 너무 익숙해져서 "???.jsp" 파일을 바로 요청하면 바로 결과가 전송되어서 직관적이라고 해야 할까요. 그렇지만 jsp를 실행 한다는 것도 알고보면 웹 컨테이너(톰캣)가 내부적으로 "???.jsp"에 대한 매핑정보를 웹 컨테이너가 관리했기 때문이지 사실 "???.jsp" 라는 파일은 실행 되지 않는 다는것을 알잖아요.
- 클라이언트가 모르게 서버에서는 여러 단계를 거쳐 결과페이지(html, xml, 등)를 생성한다는 것을 알아야 합니다. 그것이 포워드를 통한 것이든, invoke를 통해 호출에 의한 것이든 모든 처리는 서버에서 일어나고 클라이언트는 서버의 누군가(뷰, jsp)에 의해 만들어낸 페이지를 본다는 것만 인지하도록 합시다.

❑ 상품 목록 및 상품 상세보기를 보여주는 뷰 작성

- 특별하게 어려운 것은 없을 것이라 생각합니다.
- 클라이언트가 "http://서버주소:포트/웹컨텍스트/product/prodList.do" 요청이 들어왔을 때 DispatcherServlet은 ProdAnnoController의 prodList() 메서드가 실행시키고 모델맵에 "prodList" 라는 이름으로 리스트가 저장됩니다. 반환된 문자열 "prod/prodList"는 ViewResolver에 의해(web-anno-servlet.xml 참고) "/WEB-INF/res/prod/prodList.jsp"를 내부 자원에서 호출하여 jsp 파일이 실행됩니다.
- 동일하게 "http://서버주소:포트/웹컨텍스트/product/prodDetail.do" 의 요청이 ProdAnnoController의 prodDetail() 메서드가 실행되고 "prodBean" 으로 모델맵에 저장됩니다.
  - /WEB-INF/res/prod/prodList.jsp
  - /WEB-INF/res/prod/prodDetail.jsp

❑ 스프링 설정 파일 작성

- /WEB-INF/classes/applicationContext.xml

```
<bean id="configProperties"
       class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
       p:location="classpath:org/ddit/ibatis/db.properties" />

<bean id="dataSource"
       class="org.springframework.jdbc.datasource.DriverManagerDataSource"
       p:driverClassName="${driverClassName}"
       p:url="${url}"
       p:username="${username}"
       p:password="${password}"  />

<bean id="sqlMapClient" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean"
       p:configLocation="classpath:org/ddit/ibatis/SqlMapSpringConfig.xml"
       p:mappingLocations="classpath:org/ddit/ibatis/mapping/*.xml"
       p:dataSource-ref="dataSource" />

<bean id="prodDao" class="kr.or.ddit.prod.dao.ProdDaoiBatis"
       p:sqlMapClient-ref="sqlMapClient"  />

<bean id="prodService" class="kr.or.ddit.prod.service.ProdServiceImpl"
       p:prodDao-ref="prodDao" />
```

- /WEB-INF/classes/web-anno-servlet.xml

```
<context:component-scan base-package="org.ddit.web.controller" >
  <context:include-filter type="annotation"
                          expression="org.springframework.stereotype.Controller" />
</context:component-scan>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
       p:prefix="/WEB-INF/res/"
       p:suffix=".jsp" />
```

❑ 참고 : https://mybatis.github.io/mybatis-3/ko, http://mybatis.github.io/spring/ko/

- 본 교재는 DataMapper 의 역할이나 Mybatis 에 대한 기본 지식을 갖추었음을 가정한다.

❑ 단계

- Mybatis 및 Mybatis-Spring 모듈 의존성 추가, (메이븐 사용을 가정한다)
- MapperConfiguration 파일 작성 : typeAlias 등을 등록하기위해 작성하기도 하나, 어노테이션을 활용하면 XML은 거의 사용할 필요가 없어진다.
- Mapper XML : 실행 쿼리 작성
- 매퍼 인터페이스 : type safety 를 보장하기 위해 매퍼 XML과 매퍼 인터페이스를 함께 사용
- SqlSession 생성 빈 등록
- Business logic object 에서 생성된 매퍼 프록시 객체를 interface type 기준으로 주입받아 사용

❑ 의존성 추가

```
<dependency>
	<groupId>org.mybatis</groupId>
	<artifactId>mybatis-spring</artifactId>
	<version>1.2.2</version>
</dependency>
<dependency>
	<groupId>org.mybatis</groupId>
	<artifactId>mybatis</artifactId>
	<version>3.2.8</version>
</dependency>
```

❑ Mapper XML 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="kr.or.ddit.member.dao.IMemberDao">
	<select id="selectMemberList" resultType="memberVO">
		SELECT MEM_ID, MEM_NAME,  MEM_ADD1,
		  MEM_HP,  MEM_MAIL,  MEM_MILEAGE
		FROM MEMBER
	</select>
</mapper>
```

❑ 매퍼 인터페이스 작성

```
package kr.or.ddit.member.dao;

import java.util.List;
import kr.or.ddit.vo.MemberVO;
import org.springframework.stereotype.Repository;

@Repository
public interface IMemberDao {
	public List<MemberVO> selectMemberList();
}
```

❑ SqlSession 생성 빈 등록

```
<context:property-placeholder
                location="classpath:/kr/or/ddit/database_info.properties" />
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        p:driverClassName="${maindb.driverClassName}"
        p:url="${maindb.url}"
        p:username="${maindb.username}"
        p:password="${maindb.password}"
        p:initialSize="${maindb.initialSize}"
        p:maxWait="${maindb.maxWait}"
/>
<bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource"
/>
<!-- myBatis 에서 SqlMapClient 대신 SqlSession 을 사용하며, -->
<!-- 설정 파일과 매퍼 파일 정보를 읽어 SqlSession을 생성해주는 빌더를 전략으로 등록함. -->
<!-- 설정파일의 경우, environment 를 비롯한 대부분의 것들을 SqlSessionFactoryBean 가
생성하기 때문에 별도의 설정 파일 없이 빈 등록시 직접 설정하는 것이 가능한다. -->
<!-- @Alias 어노테이션을 사용하는 특정 패키지 아래의 클래스들에 타입별칭은 일괄 설정 가능.-->
<!-- 혹은 CoC에 따라 클래스명을 기준으로 타입별칭이 등록되기도 한다. -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"
        p:dataSource-ref="dataSource"
        p:mapperLocations="classpath:kr/or/ddit/mybatis/mappers/*.xml"
        p:typeAliasesPackage="kr.or.ddit.vo"
/>
<!--
        특정 패키지에서 매퍼 인터페이스(DAO interface) 를 스캔해서 매퍼 프록시 객체를
        생성하기 위한 전략으로 DAO의 실구현체를 직접 작성할 필요가 없도록 한다.
        특정 패키지들에서 @Repository 어노테이션을 가진 매퍼들에 대해 프록시를 생성해줌.
        mybatis-spring integration 1.0.2 이후부터
        MapperScannerConfigurer 와 SqlSessionFactory 혹은 템플릿을 연결할때,
        레퍼런스(p:sqlSessionFactory-ref="sqlSessionFactory")로 의존관계를 형성하게 되면,
        BeanFactoryPostProcessor 들이 동작하지 않는 경우가 있기 때문에,
        대신 sqlSessionFactoryBeanName 을 설정하도록 권고하고 있음.
        http://mybatis.github.io/spring/mappers.html#MapperScannerConfigurer
-->
<!-- <bean id="mapperScannerConfigurer"
        class="org.mybatis.spring.mapper.MapperScannerConfigurer"
        p:basePackage="kr.or.ddit.*.dao"
        p:annotationClass="org.springframework.stereotype.Repository"
        p:sqlSessionFactoryBeanName="sqlSessionFactory"
/> -->
 <mybatis-spring:scan base-package="kr.or.ddit.*.dao"
        annotation="org.springframework.stereotype.Repository"
        factory-ref="sqlSessionFactory"
/>
```

❑ 매퍼 프록시 객체 주입

```
@Service
public class MemberServiceImpl implements IMemberService {
        @Inject
        IMemberDao memberDao;
}
```