

Anyframe Query Plugin



Version 1.1.2

저작권 © 2007-2011 삼성SDS

본 문서의 저작권은 삼성SDS에 있으며 Anyframe 오픈소스 커뮤니티 활동의 목적하에서 자유로운 이용이 가능합니다. 본 문서를 복제, 배포할 경우에는 저작권자를 명시하여 주시기 바라며 본 문서를 변경하실 경우에는 원문과 변경된 내용을 표시하여 주시기 바랍니다. 원문과 변경된 문서에 대한 상업적 용도의 활용은 허용되지 않습니다. 본 문서에 오류가 있다고 판단될 경우 이슈로 등록해 주시면 적절한 조치를 취하도록 하겠습니다.

I. Introduction	1
II. Query	2
1. Configuration	3
1.1. jdbcTemplate	3
1.2. sqlRepository	4
1.3. pagingSQLGenerator	5
1.4. lobHandler	6
1.5. Samples	7
1.6. TestCase	8
1.6.1. INSERT	8
1.6.2. SELECT	8
1.6.3. UPDATE	9
1.6.4. DELETE	10
1.7. Configuration Simplification	10
2. Mapping XML Files	13
2.1. Mapping XML 스키마	13
2.2. table-mapping 정의 방법	13
2.3. queries 정의 방법	14
3. Usecases	17
3.1. Result Mapping	17
3.1.1. 조회 결과 매핑이 별도로 정의되어 있지 않은 경우	17
3.1.2. <result-mapping> 없이 <table-mapping>을 이용할 경우	17
3.1.3. <table-mapping>,<result-mapping>없이 <result>만을 이용할 경우	18
3.1.4. <result-mapping>을 이용할 경우	18
3.1.5. 테스트 코드 Sample	20
3.2. Embedded SQL	21
3.2.1. 속성 정의 파일 Sample	21
3.2.2. 테스트 코드 Sample	21
3.3. OR Mapping	22
3.3.1. 속성 정의 파일 Sample	22
3.3.2. 매핑 XML 파일 Sample	23
3.3.3. OR Mapping시 사용할 매핑 클래스 Sample	23
3.3.4. 테스트 코드 Sample	24
3.4. Dynamic Query	25
3.4.1. 속성 정의 파일 Sample	25
3.4.2. 매핑 XML 파일 Sample	26
3.4.3. 테스트 코드 Sample	26
3.5. Pagination	30
3.5.1. 속성 정의 파일 Sample	30
3.5.2. 매핑 XML 파일 Sample	31
3.5.3. 테스트 코드 Sample	31
3.6. Batch Update	32
3.6.1. 속성 정의 파일 Sample	32
3.6.2. 매핑 XML 파일 Sample	32
3.6.3. 테스트 코드 Sample	33
3.7. Callable Statement	35
3.7.1. 속성 정의 파일 Sample	35
3.7.2. 매핑 XML 파일 Sample	36
3.7.3. 테스트 코드 Sample	36
3.8. CLOB, BLOB	37
3.8.1. Oracle 9i 이상일 경우	37
3.8.2. Oracle 8i일 경우	40
3.9. Named Parameter 'vo' 활용	41
3.9.1. 속성 정의 파일 Sample	41
3.9.2. 매핑 XML 파일 Sample	41

3.9.3. 테스트 코드 Sample	42
3.10. extends AbstractDao	44
3.10.1. 매핑 XML 파일 Sample	45
3.10.2. Dao 클래스 코드 Sample	46
3.10.3. Dao 클래스 속성 정의 파일 Sample	47
3.10.4. Dao 클래스 테스트 코드 Sample	47
3.11. implements ResultSetMapper	48
3.11.1. 속성 정의 파일 Sample	48
3.11.2. 매핑 XML 파일 Sample	48
3.11.3. ResultSetMapper 코드 Sample	49
3.11.4. 테스트 코드 Sample	49

I.Introduction

query-plugin은 Spring JdbcTemplate를 확장/활용하여 XML 기반 쿼리문 실행, 조회 결과를 특정 객체로 매핑, 조회 결과에 대한 DB별 페이징 처리, OR 매핑, Dynamic Query 실행, Callable Statement 실행 등과 같은 다양한 기능을 간편하게 수행할 수 있도록 지원하는 Query Service의 기본 활용 방법을 가이드하기 위한 샘플 코드와 이 오픈소스를 활용하는데 필요한 참조 라이브러리들로 구성되어 있다.

Installation

Command 창에서 다음과 같이 명령어를 입력하여 query-plugin을 설치한다.

```
mvn anyframe:install -Dname=query
```

installed(mvn anyframe:installed) 혹은 jetty:run(mvn clean jetty:run) command를 이용하여 설치 결과를 확인해볼 수 있다.

Dependent Plugins

Plugin Name	Version Range
Core [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.0.3/reference/htmlsingle/core.html]	2.0.0 > *

II. Query

Query 서비스는 쿼리문이나 객체의 입력만으로 DB 데이터 조작을 가능하게 하는 서비스이다. Query 서비스는 JDBC(Java Database Connectivity)를 이용한 데이터 액세스 수행 부분을 추상화함으로써 간편한 데이터 액세스 방법을 제공하고, JDBC 사용 시 발생할 수 있는 공통적인 에러를 줄여준다. Query 서비스는 내부적으로 DataSource 서비스를 이용하고 있으므로, DataSource 서비스와 같이 배포되어야 함에 유의하도록 한다.

Query 서비스 매뉴얼에서 제공하는 모든 테스트 코드는 HSQL DB를 기반으로 실행된다. (단, ※ CallableStatement, LOB의 경우는 Oracle 9i, 10g를 기반으로 함.)

1. Configuration

Query 서비스를 활용하기 위해서는 다음과 같은 속성들이 정의되어 있어야 한다. 다음에서 각 속성이 가지는 의미에 대해 알아보기로 하자.

Property	Description	Required	Default Value
jdbcTemplate	DataSource 서비스를 이용하여 해당하는 DB로부터 java.sql.Connection을 얻어 정의된 쿼리문을 실행시킬 수 있도록 한다. PagingJdbcTemplate의 Bean Id를 값으로 정의한다.	Y	N/A
sqlRepository	테이블 매핑 정보 및 쿼리문을 정의한 매핑 XML 파일들을 처리하는 역할을 수행하는 SQLLoader의 Bean Id를 정의한다.	Y	N/A
pagingSQLGenerator	DB에 특화된 형태의 페이징 처리를 위한 SQL을 정의하지 않더라도, 해당 DB에 따라 페이징 처리를 위해 알맞은 SQL을 생성할 수 있도록 도와주는 PagingSQLGenerator의 Bean Id를 정의한다.	Y	N/A
lobHandler	LOB 유형의 데이터를 다루어야 하는 경우 해당하는 DB에 적합한 LobHandler의 Bean Id를 정의한다.	N	N/A
velocityPropsFilename	Dynamic SQL 문을 다루어야 하는 경우 Velocity에 의해 남겨지는 Log 파일의 경로를 변경하고자 할 때 정의한다. Velocity Log 파일의 경로는 다음과 같이 절대/상대 경로(file:...)나 클래스패스(classpath:...)를 이용하여 정의 가능하다. <ul style="list-style-type: none"> file:./query/log/velocity.log classpath:/anyframe/core/query/log/velocity.log 값을 정의하지 않았을 경우에는 Velocity Log는 남겨지지 않는다. Log를 남기고자 하는 경우에는 정의된 경로에 지정된 로그 파일이 생성되어 있어야 함에 유의하도록 한다.	N	Log를 남기지 않음.

위에서 언급한 Query 서비스가 필요로 하는 설정 정보 중, jdbcTemplate, sqlRepository, pagingSQLGenerator, lobHandler에 대해 좀 더 자세히 짚어보기로 하자.

1.1.jdbcTemplate

Query 서비스에서는 DataSource 서비스를 이용하여 해당하는 DB에 접근하고, java.sql.Connection 객체를 얻어내어 정의된 쿼리문을 실행시키기 위해 org.anyframe.query.impl.jdbc 패키지 하위에 Spring의 JdbcTemplate을 확장한 PagingJdbcTemplate, PagingNamedParamJdbcTemplate, OraclePagingJdbcTemplate를 제공하고 있다.

- PagingJdbcTemplate : 내부 ResultSetExtractor를 이용하여 조회 결과에 대한 매핑 처리 및 페이징 처리를 수행한다.
- OraclePagingJdbcTemplate : DBMS가 Oracle일 경우, batch 처리에 의해 변경된 데이터의 전체 건수를 알아야 하는 경우에 사용할 수 있다. Oracle에서는 Insert, Update, Delete에 대한 batch 처리를 위해 2가지 방법을 제공한다. 그 중 하나가 **JDBC 2.0 Spec**을 준수하여 처리하는 방법이며, 다른 하나는 **Oracle 특화된 batch 처리 방법**이다. Query 서비스는 이 중 첫번째 방법으로 batch 작업을 수

행하는데, 정상 처리되었을 경우 수행 결과로 전달되는 int 배열 내에는 -2 값이 셋팅되어 있게 된다. JDBC 2.0 Spec.에 의하면 결과값 -2가 의미하는 바는 "정상 처리되었으나 변경된 데이터의 건수를 알 수 없음."이다. 따라서, batch 처리 후, 변경된 데이터의 전체 건수를 알기 위해서는 두번째 방법으로 batch 작업을 수행해야 하며 이를 위해서 제공된 구현체가 OraclePagingJdbcTemplate이다.

- PagingNamedParamJdbcTemplate : 별도 설정없이 QueryServiceImpl 클래스에서 내부적으로 사용하는 클래스로 Named Parameter를 가진 Dynamic SQL 처리를 수행한다.

다음은 PagingJdbcTemplate을 위해 필요한 주요 설정 정보들이다. (이외 설정 정보에 대해서는 org.springframework.jdbc.core.JdbcTemplate의 설정 정보를 참고하도록 한다.)

Property	Description	Required	Default Value
dataSource	참조할 dataSource의 Bean Id를 정의한다.	Y	N/A
exceptionTranslator	ExceptionTranslator의 Bean Id를 정의한다. ExceptionTranslator는 DB 데이터 조작시 SQLException이 발생한 경우 별도 Exception 객체에 해당하는 SQL Error Code와 Error Message 정보를 셋팅하여 throw하도록 하는 역할을 수행하며, Query 서비스에서는 org.anyframe.query.impl.util.RawSQLExceptionTranslator를 제공하고 있다. 따라서, ExceptionTranslator를 별도 셋팅하면, 데이터 조작으로 인한 오류가 발생한 경우 Query 서비스를 통해 throw된 QueryServiceException으로부터 SQL Error Code와 Error Message 정보를 추출할 수 있게 된다.	N	N/A
maxFetchSize	다량의 데이터 전체 조회시 발생할 수 있는 성능 저하를 방지하기 위해, maxFetchSize를 활용할 수 있다. 조회된 결과 데이터의 건수가 정의된 maxFetchSize보다 큰 경우 Exception 처리된다. 이 속성은 Query Service를 통해 조회되는 모든 쿼리문에 대해 적용되는 속성이며 특정 쿼리문에 대해서만 결과 데이터의 건수를 제한하고자 하는 경우에는 Mapping XML 파일 내에 쿼리문 정의시 쿼리문별로 maxFetchSize를 정의하여 허용되는 최대 조회 건수를 제한할 수 있다. maxFetchSize의 우선순위는 특정 쿼리문에 대해 부여한 maxFetchSize > jdbcTemplate의 maxFetchSize의 순으로 처리되며, maxFetchSize가 정의되지 않은 경우에는 조회된 결과 데이터의 건수를 제한하지 않는다.	N	N/A
nativeJdbcExtractor	OraclePagingJdbcTemplate을 사용하는 경우에만 정의할 수 있다. 사용중인 Connection Pool에 맞게 Wrapping되어 있는 Connection 객체로부터 본래의 JDBC Connection 객체를 추출하는 역할을 수행하는 NativeJdbcExtractor의 Bean Id를 정의한다.	N	N/A

1.2.sqlRepository

Query 서비스에서는 테이블 매핑 정보 및 쿼리문을 정의한 매핑 XML 파일들을 처리하는 역할을 수행하기 위해 org.anyframe.query.impl.config.loader.SQLLoader를 제공하고 있다. 다음은 SQLLoader를 위해 필요한 주요 설정 정보들이다.

Property	Description	Required	Default Value
mappingFiles		Y	N/A

Property	Description	Required	Default Value
	테이블 매핑 정보와 사용할 쿼리문을 정의하고 있는 매핑 XML 파일명을 지정하는 요소로 ','를 구분자로 하여 복수 정의 가능하다. filename 요소에 대한 지정은 Spring Configuration 파일 경로 지정 방식과 동일하므로, 절대/상대적인 파일 경로 지정(file:...)과 클래스패스를 이용한 지정(classpath:...)이 가능하다. *를 활용한 Pattern Matching 역시 적용 가능하다.		
nullchecks	해당 DB Column의 값이 없어서 null value가 리턴되었을 때, 지정한 값으로 변환시켜준다. 현재, CHAR, VARCHAR, LONGVARCHAR 타입의 칼럼에 대해서만 지원된다. Collection 타입 중, <map>을 이용하여 정의한다.	N	N/A
dynamicReload	매핑 XML 파일에 대한 동적 Reload 주기를 세팅한다. (milliseconds 단위) 10미만 입력시 10으로 인식하며, 10 이상 입력시 입력값으로 인식한다. 입력하지 않을 경우 동적 Reload는 수행되지 않는다.	N	
skipError	매핑 XML 파일을 읽어들이면서, error가 발생한 경우 skip 여부를 셋팅한다.	N	

1.3.pagingSQLGenerator

Query 서비스에서는 페이징 처리를 위해 DB에 특화된 형태의 SQL을 구성하지 않더라도, 해당 DB에 적합한 페이징 처리 SQL을 구성할 수 있도록 도와주는 역할을 수행하기 위해 다음과 같은 PagingSQLGenerator를 제공한다.

DB 종류	PagingSQLGenerator Class
Altibase	org.anyframe.query.impl.jdbc.generator.AltibasePagingSQLGenerator
DB2	org.anyframe.query.impl.jdbc.generator.DB2PagingSQLGenerator
HSQLDB	org.anyframe.query.impl.jdbc.generator.HSQLPagingSQLGenerator
MySQL	org.anyframe.query.impl.jdbc.generator.MySQLPagingSQLGenerator
Oracle	org.anyframe.query.impl.jdbc.generator.OraclePagingSQLGenerator

이 외, PagingSQLGenerator가 필요한 경우에는 org.anyframe.query.impl.jdbc.generator.AbstractPagingSQLGenerator를 확장하여 신규 PagingSQLGenerator를 생성하고, getPaginationSQL() 메소드를 구현해주면 된다. getPaginationSQL() 메소드에는 입력받은 SQL을 기반으로 페이징 처리를 위해 변경된 SQL을 전달하는 로직을 정의하면 된다. 다음은 OraclePagingSQLGenerator 클래스의 일부 내용이다.

```
public class OraclePagingSQLGenerator extends AbstractPagingSQLGenerator {
    public String getPaginationSQL(String originalSql, Object[] originalArgs,
        int[] originalArgTypes, int pageIndex, int pageSize) {
        // 정의된 기본 쿼리문을 ROWNUM을 이용한 형태로 변경하기 위해 앞,뒤로 문자열 추가
        StringBuilder sql = new StringBuilder(
            " SELECT * FROM ( SELECT INNER_TABLE.* , ROWNUM AS ROW_SEQ FROM ( \n");
        sql.append(originalSql);
        sql.append(" ) INNER_TABLE WHERE ROWNUM <= ? ) "
            + " WHERE ROW_SEQ BETWEEN ? AND ?");

        // 변경된 쿼리문 전달
        return sql.toString();
    }
}
```

```

// 쿼리문에 입력되어야 할 기본 입력 인자 외에
// 페이징 처리를 위한 pageIndex, pageSize 인자값 셋팅
public Object[] setQueryArgs(Object[] originalArgs, int pageIndex,
    int pageSize) {
    Object[] args = new Object[originalArgs.length + 3];

    for (int i = 0; i < originalArgs.length; i++) {
        args[i] = originalArgs[i];
    }

    args[originalArgs.length] = String.valueOf(new Long(pageIndex
        * pageSize));
    args[originalArgs.length + 1] = String.valueOf(new Long((pageIndex - 1)
        * pageSize + 1));
    args[originalArgs.length + 2] = String.valueOf(new Long(pageIndex
        * pageSize));

    return args;
}

// 쿼리문에 입력되어야 할 기본 입력 인자 외에
// 페이징 처리를 위한 pageIndex, pageSize 인자 타입 셋팅
public int[] setQueryArgTypes(int[] originalArgTypes) {
    int[] argTypes = new int[originalArgTypes.length + 3];

    for (int i = 0; i < originalArgTypes.length; i++) {
        argTypes[i] = originalArgTypes[i];
    }

    argTypes[originalArgTypes.length] = Types.VARCHAR;
    argTypes[originalArgTypes.length + 1] = Types.VARCHAR;
    argTypes[originalArgTypes.length + 2] = Types.VARCHAR;

    return argTypes;
}
}
}

```

또한, 조회 조건에 해당하는 전체 데이터의 건수를 조회하기 위한 쿼리문은 상위 클래스인 `AbstractPagingSQLGenerator` 내에 정의되어 있으며 해당 DBMS가 `count(*)`을 지원하지 않는 경우에는 해당하는 `PagingSQLGenerator`에서 오버라이드해 주도록 한다.

사용중인 DBMS에 적합한 `pagingSQLGenerator`가 없어서, `DefaultPagingSQLGenerator`를 사용하는 경우, 특정 페이지에 속한 목록을 조회할 때 Query 서비스에서는 일단 해당되는 전체 목록을 모두 조회한다. 그리고 `ScrollableResultSet`의 `Cursor`를 이용하여 해당 페이지에 속한 데이터들을 추출하게 된다. 따라서, 해당 페이지에 속한 데이터만을 조회하는 `PagingSQLGenerator`에 비해 성능이 저하된다는 점에 유의하도록 한다.

1.4.lobHandler

Query 서비스에서는 Spring에서 제공하는 `LobHandler`를 사용하여 LOB 유형의 데이터를 다루도록 권장한다. 다음은 Spring에서 제공하는 `LobHandler` 목록이다.

- Oracle(9i이상) : `org.springframework.jdbc.support.lob.OracleLobHandler`
- the Others : `org.springframework.jdbc.support.lob.DefaultLobHandler`

단, Spring에서 제공하는 `OracleLobHandler`의 경우 Oracle 9i 이상에서만 사용 가능하므로 Oracle 8i 사용자를 위해 `org.anyframe.query.impl.jdbc.lob.Oracle8iLobHandler`를 추가로 제공하고 있다. `OracleLobHandler`나 `Oracle8iLobHandler`의 경우 다음과 같은 설정 정보가 필요하다.

Property	Description	Required	Default Value
nativeJdbcExtractor	<p>사용중인 Connection Pool에 맞게 Wrapping되어 있는 Connection 객체로부터 본래의 JDBC Connection 객체를 추출하는 역할을 수행하는 NativeJdbcExtractor의 Bean Id를 정의한다. (해당 lobHandler에서 nativeJdbcExtractor를 필요로 하는 경우에만 정의)</p> <p>다음은 Spring에서 제공하는 주요 JdbcExtractor 클래스들이다.</p> <ul style="list-style-type: none"> Common DBCP : org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor C3PO : org.springframework.jdbc.support.nativejdbc.C3PONativeJdbcExtractor WebLogic : org.springframework.jdbc.support.nativejdbc.WebLogicNativeJdbcExtractor WebSphere : org.springframework.jdbc.support.nativejdbc.WebSphereNativeJdbcExtractor <p>즉, 오픈소스 프로젝트인 Commons DBCP를 Connection Pool로 채택한 경우 CommonsDbcpNativeJdbcExtractor를 사용할 수 있다.</p>	N	N/A

1.5.Samples

다음은 위에서 언급한 Query 서비스 속성 정의를 포함하고 있는 context-query.xml의 일부이다.

```

<bean id="queryService" class="org.anyframe.query.impl.QueryServiceImpl">
  <property name="jdbcTemplate" ref="jdbcTemplate"/>
  <property name="sqlRepository" ref="sqlLoader"/>
  <property name="pagingSQLGenerator" ref="pagingSQLGenerator"/>
  <property name="lobHandler" ref="lobHandler"/>
  <!-- if you don't define velocityPropsFilename,
       queryservice doesn't make a velocity log file. -->
  <property name="velocityPropsFilename" value="file:./testvelocity/velocity.log"/>
</bean>

<bean id="jdbcTemplate" class="org.anyframe.query.impl.jdbc.PagingJdbcTemplate">
  <property name="dataSource" ref="dataSource" />
</bean>

<bean id="pagingSQLGenerator"
      class="org.anyframe.query.impl.jdbc.generator.OraclePagingSQLGenerator"/>

<bean id="nativeJdbcExtractor"
      class="org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor"
      lazy-init="true"/>

<bean id="lobHandler" class="org.springframework.jdbc.support.lob.OracleLobHandler"
      lazy-init="true">
  <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/>

```

```
</bean>
```

또한, sqlLoader의 속성은 다음 context-query-sqlloader.xml 파일에서와 같이 정의할 수 있다.

```
<bean name="sqlLoader" class="org.anyframe.query.impl.config.loader.SQLLoader">
<property name="mappingFiles">
<value>
<!-- xml files in folder -->
file:./src/test/resources/foldermappings/**/testcase-*.xml,
file:./src/test/resources/dynamicreload/testcase-queries-dynamicreload.xml,
<!-- xml files in classpath -->
classpath*/mappings/testcase-queries-dynamic.xml,
classpath*/mappings/testcase-queries-general.xml,
classpath*/mappings/testcase-queries-resultmapping.xml,
classpath*/mappings/testcase-queries-resultsetmapper.xml,
classpath*/mappings/testcase-table-mappings.xml,
classpath*/mappings/testcase-queries-extended.xml,
classpath*/mappings/testcase-queries-mappingstyle.xml,
<!-- xml files in jar -->
classpath*/jarmappings/testcase-queries-lob.xml
</value>
</property>
<property name="nullChecks">
<map>
<!-- key: type, value: default-value -->
<entry key="VARCHAR" value=""/>
</map>
</property>
<property name="dynamicReload" value="5" />
<property name="skipError" value="true" />
</bean>
```

1.6.TestCase

다음은 Query 서비스를 사용하여 해당하는 DB에 샘플 데이터를 INSERT, SELECT, UPDATE, DELETE하는 테스트 코드의 일부이다.

1.6.1.INSERT

다음은 INSERT 예제이다.

```
public void insertQuery() throws Exception{
    IQueryService queryService = (IQueryService) context.getBean("queryService");
    //create() : XML mapping 파일에 정의되어 있는 SQL query를 이용하여 INSERT를 실행한다.
    int rs = queryService.create
        ("create", new Object[] { "1234567890123", "AAAAA" , "seoul"});
    if ( rs == -1 ){
        throw new Exception("Insert query failed");
    }
}
```

1.6.2.SELECT

다음은 SELECT 예제이다.

```
public void selectQuery() throws Exception{
```

```

IQueryService queryService = (IQueryService) context.getBean("queryService");
//find() : XML mapping파일에 정의되어 있는 SQL query를 이용하여 SELECT를 실행한다.

//일반적인 경우(table과 class를 mapping하지 않은 경우)
ArrayList rsquery = (ArrayList) queryService
    .find("selectGeneral", new Object[] { "%12345%" });
Map hsRsquery = new HashMap();
for( int i = 0 ; i < rsquery.size() ; i ++ ){
    hsRsquery = (Map) rsquery.get(i);
    String name = (String) hsRsquery.get("name");
}

/*매핑 XML에 해당 클래스와 매핑되는 테이블이 존재하지 않을 경우,
* 쿼리 수행 결과에 대해 하나의 Row별도 칼럼명,
* 해당값을 쌍으로 org.apache.commons.collections.map.ListOrderedMap에 put하고
* ListOrderedMap들을 ArrayList에 담은 형태로 결과값을 리턴하게 된다.
*/

//Table - Class mapping을 사용한 경우
Collection rsqueryNotUsingResultMapping = queryService
    .find("selectNotUsingResultMapping", new Object[] { "%12345%" });
Iterator rsqueryItr = rsqueryNotUsingResultMapping.iterator();
while (rsqueryItr.hasNext()) {
    Customer customer = (Customer) rsqueryItr.next();
    String name = customer.getNm();
}

//result-mapping을 사용한 경우
Collection rsqueryUsingResultMapping = queryService
    .find("selectUsingResultMapping", new Object[] { "%12345%" });
Iterator rsqueryItr_01 = rsqueryUsingResultMapping.iterator();
while (rsqueryItr_01.hasNext()) {
    CompositionCustomer compositionCustomer
        = (CompositionCustomer) rsqueryItr_01.next();
    String name = compositionCustomer.getCompositionName();
}

System.out.println("rsquery.size() : "
    + rsquery.size());
System.out.println("rsqueryNotUsingResultMapping.size() : "
    + rsqueryNotUsingResultMapping.size());
System.out.println("rsqueryUsingResultMapping.size( : "
    + rsqueryUsingResultMapping.size());
}

```

1.6.3.UPDATE

다음은 UPDATE 예제이다.

```

public void updateQuery() throws Exception {
    IQueryService queryService = (IQueryService) context
        .getBean("queryService");
    //update() : XML mapping파일에 정의되어 있는 SQL query를 이용하여 UPDATE를 실행한다.
    int rs = queryService.update("update"
        , new Object[] { "999999999999", "AAAAA", "busan", "1234567890123"});
    if ( rs == -1 ){
        throw new Exception("Update query failed");
    }
}
}

```

1.6.4.DELETE

다음은 DELETE 예제이다.

```
public void deleteQuery() throws Exception {
    IQueryService queryService = (IQueryService) context
        .getBean("queryService");
    //remove() : XML mapping 파일에 정의되어 있는 SQL query를 이용하여 DELETE를 실행한다.
    int rs = queryService.remove("delete", new Object[] { "9999999999999999" });
    if ( rs == -1 ){
        throw new Exception("Delete query failed");
    }
}
```

1.7.Configuration Simplification

QueryService의 속성을 정의하기 위해서는 앞서 살펴본 바와 같이 jdbcTemplate, pagingSQLGenerator, sqlRepository 등과 같은 다양한 Bean에 대한 속성 정의가 수반되어야 하므로 QueryService의 속성 정의가 다소 복잡하다고 느낄 수 있다. 또한 QueryService는 특정 DB를 기반으로 동작하므로 여러 DB를 이용하는 어플리케이션을 구축하는 경우에는 DB 속성 뿐만 아니라 이와 매핑되는 QueryService 속성 정의도 함께 추가하면서 속성 정의가 더욱 복잡해질 수 있다. QueryService에서는 속성 정의의 복잡성을 제거하고 간편하게 QueryService의 속성을 정의할 수 있도록 지원하기 위해 query namespace를 제공한다. (버전 1.1.0 이후)

다음에서는 query namespace를 활용하는 방법에 대해서 차례대로 살펴보도록 하자.

query namespace는 auto-config라는 태그를 제공하고 있으며 auto-config는 다음과 같은 속성을 가진다.

Property	Description	Required	Default Value
dbType	<p>QueryService를 통해 연결한 DBMS의 타입을 정의한다. altibase,db2,hsqldb,mysql,oracle,default 중에서 선택 가능하며, 정의된 dbType에 의해 pagingSQLGenerator와 lobHandler Bean의 구현체가 결정된다. 만약 Anyframe에서 제공하지 않는 PagingSQLGenerator 구현체를 사용하는 경우에는 구현체 클래스명을 반드시 xxxxPagingSQLGenerator와 같은 형태로 정의하고 dbType의 값으로 구현 클래스명에서 PagingSQLGenerator를 뺀 나머지 문자열을 대소문자 구분하여 정의하면 된다.</p> <p>다음은 각 dbType별로 셋팅되는 pagingSQLGenerator Bean의 구현체이다. (다음에서 제시되는 구현체들은 모두 org.anyframe.query.impl.jdbc.generator 패키지 하위에 속한다.)</p> <ul style="list-style-type: none"> altibase : AltibasePagingSQLGenerator db2 : DB2PagingSQLGenerator hsqldb : HSQLPagingSQLGenerator mysql : MySQLPagingSQLGenerator oracle : OraclePagingSQLGenerator 	Y	N/A

Property	Description	Required	Default Value
	<ul style="list-style-type: none"> default : DefaultPagingSQLGenerator <p>적절한 PagingSQLGenerator 구현체를 찾지 못하였을 경우에도 DefaultPagingSQLGenerator로 셋팅된다.</p> <p>다음은 각 dbType별로 셋팅되는 lobHandler Bean의 구현체이다. (다음에서 제시되는 구현체들은 모두 org.springframework.jdbc.support.lob 패키지 하위에 속한다.)</p> <ul style="list-style-type: none"> oracle : OracleLobHandler 그 외 : DefaultLobHandler 		
id	<query:auto-config>를 이용하여 속성을 정의한 QueryService Bean에 대해 별도의 식별자를 부여하고자 하는 경우에 사용한다.	N	queryService
dataSource-ref	<query:auto-config>를 이용하여 속성을 정의한 QueryService Bean은 기본적으로 'dataSource'라는 이름의 Bean을 참조하여 실행된다. 다른 DataSource Bean을 참조하여 QueryService를 실행하고자 하는 경우에 사용한다.	N	dataSource
jdbcTemplate-ref	<query:auto-config>를 이용하여 속성을 정의한 QueryService Bean은 기본적으로 'jdbcTemplate'이라는 이름의 Bean을 참조하여 실행된다. 이때 기본 적용되는 'jdbcTemplate' Bean의 구현체는 org.anyframe.query.impl.jdbc.PagingJdbcTemplate이다. 따라서 QueryService를 통해 제공되는 org.anyframe.query.impl.jdbc.OraclePagingJdbcTemplate과 같은 다른 JdbcTemplate 구현체를 활용하고자 하는 경우 사용한다.	N	jdbcTemplate
sqlLoader-ref	<query:auto-config>를 이용하여 속성을 정의한 QueryService Bean은 기본적으로 'sqlLoader'라는 이름의 Bean을 참조하여 실행된다. 이 때 기본 적용되는 'sqlLoader' Bean은 기본적으로 클래스패스 상의 sql/query/*.xml 파일을 로드하며 매핑 XML 파일에 대해 Dynamic Reload 속성을 적용하지 않게 된다. (이 외 skipError=true, nullchecks={VARCHAR,""} 속성 적용) 따라서 매핑 XML 파일의 위치 및 기타 속성을 달리 적용하고자 하는 경우 사용한다.	N	sqlLoader

query namespace를 활용하기 위해서는 다음과 같은 namespace 정의가 선행되어야 함에 주의하도록 한다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:query="http://www.anyframejava.org/schema/query"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.anyframejava.org/schema/query
    http://www.anyframejava.org/schema/query/anyframe-query-1.0.xsd">
  <query:auto-config dbType="oracle"/>
</beans>
```

```
</beans>
```

위에서 보여진 query namespace를 이용한 속성 정의는 다음에서 보이는 속성 정의와 동일한 의미를 가진다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean name="queryService" class="org.anyframe.query.impl.QueryServiceImpl">
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
    <property name="pagingSQLGenerator" ref="pagingSQLGenerator"/>
    <property name="sqlRepository" ref="sqlLoader"/>
    <property name="lobHandler" ref="lobHandler"/>
  </bean>

  <bean name="sqlLoader"
    class="org.anyframe.query.impl.config.loader.SQLLoader">
    <property name="mappingFiles">
      <value>classpath:sql/query/*.xml</value>
    </property>
    <property name="nullChecks">
      <map>
        <entry key="VARCHAR" value="" />
      </map>
    </property>
    <property name="skipError" value="true" />
  </bean>

  <bean id="jdbcTemplate"
    class="org.anyframe.query.impl.jdbc.PagingJdbcTemplate">
    <property name="dataSource" ref="dataSource" />
    <property name="exceptionTranslator" ref="exceptionTranslator" />
  </bean>

  <bean id="exceptionTranslator"
    class="org.anyframe.query.impl.util.RawSQLExceptionTranslator"/>

  <bean id="pagingSQLGenerator"
    class="org.anyframe.query.impl.jdbc.generator.OraclePagingSQLGenerator"/>

  <bean id="lobHandler"
    class="org.springframework.jdbc.support.lob.OracleLobHandler" lazy-init="true">
    <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/>
  </bean>

  <bean id="nativeJdbcExtractor"
    class="org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor"
    lazy-init="true"/>
</beans>
```

단, auto-config에 의해 정의되는 jdbcTemplate, pagingSQLGenerator, lobHandler, sqlLoader Bean의 식별자는 랜덤 문자열로 생성됨에 유의해야 한다. 다양한 DB를 이용하여 DB 접근 로직을 수행하려는 경우 QueryService가 여러개 정의되어야 하고, QueryService에 의해 참조되는 jdbcTemplate, pagingSQLGenerator, lobHandler, sqlLoader Bean들은 각각 유일한 식별자를 가져야 하기 때문이다.

2.Mapping XML Files

Query 서비스 초기화 시, Query 서비스는 속성 정의 파일에 정의되어 있는 매핑 xml 파일들을 로드한다. 그리고 사용자 요청 시 매핑 정보를 기반으로 query id를 이용해 실행하고자 하는 쿼리문을 찾아 실행한다. 매핑 XML 파일은 <queryservice>와 </queryservice>내에 크게 <table-mapping>과 <queries>로 구성된다. <queries>는 필수 요소이므로 빠뜨리지 않도록 주의해야 한다.

2.1.Mapping XML 스키마

Query 서비스를 통해 로드되는 매핑 XML 파일의 스키마는 XSD 기반(버전 1.1.0 이후)으로 정의해야 한다. (1.0.0 이전은 DTD 기반) 따라서 매핑 XML 파일을 정의하기 위해서는 먼저 다음과 같은 Namespace 정의가 선행되어야 한다.

```
<queryservice xmlns="http://www.anyframejava.org/schema/query/mapping"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.anyframejava.org/schema/query/mapping
  http://www.anyframejava.org/schema/query/mapping/anyframe-query-mapping-1.0.xsd">
  ...
</queryservice>
```

2.2.table-mapping 정의 방법

<table-mapping> 내에 <table>을 이용하여, 테이블과 특정 클래스간의 매핑 정보들을 정의할 수 있다. <table-mapping> 내에는 여러 개의 <table>을 정의할 수 있다.

Tag Name	Description	Child Tag
table	테이블과 클래스간의 매핑 정보를 정의한다. * attribute 설명 name: 해당 테이블명 class : 매핑 클래스명	field-mapping(필수), primary-key(필수)
field-mapping	테이블의 칼럼과 이에 매핑되는 클래스의 attribute를 정의한다.	dbms-column(필수), class-attribute(필수)
primary-key	해당 테이블의 Primary Key를 정의한다.	dbms-column(필수)
dbms-column	해당 테이블의 칼럼명을 정의한다.	
class-attribute	dbms-column에서 정의한 칼럼과 매핑되는 해당 클래스의 attribute명을 정의한다.	

다음은 위에서 나열한 설정 정보들을 이용한 <table-mapping>의 설정 예제로, 테이블 TBL_CUSTOMER와 클래스 Customer간의 매핑 정보를 담고 있다.

```
<queryservice>
  <table-mapping>
    <table name="TBL_CUSTOMER"
      class="anyframe.sample.domain.Customer">
      <field-mapping>
        <dbms-column>ssno</dbms-column>
```

```

        <class-attribute>ssno</class-attribute>
    </field-mapping>
</field-mapping>
    <dbms-column>name</dbms-column>
    <class-attribute>nm</class-attribute>
</field-mapping>
</field-mapping>
    <dbms-column>address</dbms-column>
    <class-attribute>addr</class-attribute>
</field-mapping>
<primary-key>
    <dbms-column>ssno</dbms-column>
</primary-key>
</table>
</table-mapping>
</queryservice>
    
```

2.3.queries 정의 방법

<queries> 내에 <query>를 이용하여, Query 서비스들을 통해 실행할 쿼리문들을 정의할 수 있다. <queries> 내에는 여러 개의 <query>를 정의할 수 있다.

Tag Name	Description	Child Tag
query	<p>쿼리문을 정의 한다.</p> <p>* attribute설명</p> <p>id : 해당하는 쿼리문을 식별하기 위한 식별자</p> <p>isDynamic : 동적 쿼리인지 아닌지 식별 (Default=true)</p> <p>isCamelCase : 조회 결과 매핑시 조회 칼럼명에 대해 CamelCase 적용할 것인지 정의 (Default=true)</p> <p>maxFetchSize : 다량의 데이터 전체 조회시 발생할 수 있는 성능 저하를 방지하기 위해, 쿼리문별로 maxFetchSize를 정의하여 허용되는 최대 조회 건수를 제한할 수 있다. 조회된 결과 데이터의 건수가 정의된 maxFetchSize보다 큰 경우 Exception 처리된다. 특정 쿼리문에 대해 maxFetchSize가 정의되지 않은 경우 jdbcTemplate의 maxFetchSize가 적용된다. jdbcTemplate에 대해서도 maxFetchSize가 정의되지 않은 경우에는 조회된 결과 데이터의 건수를 제한하지 않는다.</p> <p>mappingStyle : 조회 결과 매핑시 조회 칼럼명에 대해 어떤 유형으로 변경할 것인지 정의 (camel,lower, upper, none 중 택일, Default=camel)</p> <p>camel - 조회 칼럼명에 대해 CamelCase 적용</p> <p>lower - 조회 칼럼명에 대해 LowerCase 적용</p> <p>upper - 조회 칼럼명에 대해 UpperCase 적용</p> <p>none - 조회 칼럼명을 DBMS가 전달한 그대로 적용</p>	statement(필수), lobStatement(선택), param(선택), result(선택)

Tag Name	Description	Child Tag																																								
	<p>*isCamelCase로는 다양한 스타일의 컬럼 매핑 지원에 제한이 있어 신규 속성을 추가함. 따라서 isCamelCase는 Deprecated 예정임 (Anyframe 3.2.1 이후) mappingStyle이 적용된 경우 isCamelCase는 무시되며, 이전 버전과의 호환을 위해 isCamelCase가 true인 경우 mappingStyle은 'camel', false인 경우 'lower'로 처리됨.</p>																																									
statement	<p>실행할 쿼리문을 정의한다. Joined 쿼리에서는 동일한 조회 컬럼명이 있을 경우, Alias를 부여하도록 한다.</p>																																									
param	<p>Query 서비스는 해당 쿼리문을 미리 컴파일하여 PreparedStatement 형태로 저장하여 처리하고 있다. 따라서 입력값 셋팅을 위해 PreparedStatement에 setXXX를 수행하려면, 입력 Parameter에 해당하는 SQL Type을 java.sql.Types에 정의된 값을 참조하여 param tag의 attribute인 type의 값으로 정의한다. param tag는 입력 Parameter의 개수와 순서에 맞게 추가한다. 입력 Parameter의 데이터를 가져오기 위해서는 다음과 같은 기준에 따라 데이터 타입을 정의해야만 한다</p> <table border="1" data-bbox="432 792 1179 1704"> <thead> <tr> <th data-bbox="432 792 807 842">Java Type</th> <th data-bbox="807 792 1179 842">DBMS Type</th> </tr> </thead> <tbody> <tr><td data-bbox="432 842 807 884">String</td><td data-bbox="807 842 1179 884">CHAR</td></tr> <tr><td data-bbox="432 884 807 927">String</td><td data-bbox="807 884 1179 927">VARCHAR</td></tr> <tr><td data-bbox="432 927 807 969">String</td><td data-bbox="807 927 1179 969">LONGVARCHAR</td></tr> <tr><td data-bbox="432 969 807 1012">java.math.BigDecimal</td><td data-bbox="807 969 1179 1012">NUMERIC</td></tr> <tr><td data-bbox="432 1012 807 1055">java.math.BigDecimal</td><td data-bbox="807 1012 1179 1055">DECIMAL</td></tr> <tr><td data-bbox="432 1055 807 1097">boolean</td><td data-bbox="807 1055 1179 1097">BIT</td></tr> <tr><td data-bbox="432 1097 807 1140">byte</td><td data-bbox="807 1097 1179 1140">TINYINT</td></tr> <tr><td data-bbox="432 1140 807 1182">short</td><td data-bbox="807 1140 1179 1182">SMALLINT</td></tr> <tr><td data-bbox="432 1182 807 1225">int</td><td data-bbox="807 1182 1179 1225">INTEGER</td></tr> <tr><td data-bbox="432 1225 807 1267">long</td><td data-bbox="807 1225 1179 1267">BIGINT</td></tr> <tr><td data-bbox="432 1267 807 1310">float</td><td data-bbox="807 1267 1179 1310">REAL</td></tr> <tr><td data-bbox="432 1310 807 1352">double</td><td data-bbox="807 1310 1179 1352">FLOAT</td></tr> <tr><td data-bbox="432 1352 807 1395">double</td><td data-bbox="807 1352 1179 1395">DOUBLE</td></tr> <tr><td data-bbox="432 1395 807 1438">byte[]</td><td data-bbox="807 1395 1179 1438">BINARY</td></tr> <tr><td data-bbox="432 1438 807 1480">byte[]</td><td data-bbox="807 1438 1179 1480">VARBINARY</td></tr> <tr><td data-bbox="432 1480 807 1523">byte[]</td><td data-bbox="807 1480 1179 1523">LONGVARBINARY</td></tr> <tr><td data-bbox="432 1523 807 1565">java.sql.Date</td><td data-bbox="807 1523 1179 1565">DATE</td></tr> <tr><td data-bbox="432 1565 807 1608">java.sql.Time</td><td data-bbox="807 1565 1179 1608">TIME</td></tr> <tr><td data-bbox="432 1608 807 1650">java.sql.Timestamp</td><td data-bbox="807 1608 1179 1650">TIMESTAMP</td></tr> </tbody> </table> <p>* attribute 설정</p> <p>type : parameter의 DBMS type</p> <p>binding : CallableStatement경우 'IN','OUT','INOUT' 중 선택</p> <p>name : CallableStatement경우 Stored Procedure 내에 정의된 변수 이름 정의</p>	Java Type	DBMS Type	String	CHAR	String	VARCHAR	String	LONGVARCHAR	java.math.BigDecimal	NUMERIC	java.math.BigDecimal	DECIMAL	boolean	BIT	byte	TINYINT	short	SMALLINT	int	INTEGER	long	BIGINT	float	REAL	double	FLOAT	double	DOUBLE	byte[]	BINARY	byte[]	VARBINARY	byte[]	LONGVARBINARY	java.sql.Date	DATE	java.sql.Time	TIME	java.sql.Timestamp	TIMESTAMP	<p>생략 시 VARCHAR로 인식됨</p>
Java Type	DBMS Type																																									
String	CHAR																																									
String	VARCHAR																																									
String	LONGVARCHAR																																									
java.math.BigDecimal	NUMERIC																																									
java.math.BigDecimal	DECIMAL																																									
boolean	BIT																																									
byte	TINYINT																																									
short	SMALLINT																																									
int	INTEGER																																									
long	BIGINT																																									
float	REAL																																									
double	FLOAT																																									
double	DOUBLE																																									
byte[]	BINARY																																									
byte[]	VARBINARY																																									
byte[]	LONGVARBINARY																																									
java.sql.Date	DATE																																									
java.sql.Time	TIME																																									
java.sql.Timestamp	TIMESTAMP																																									

Tag Name	Description	Child Tag
result	<p>해당 쿼리가 조회를 위한 SELECT문일 경우에 사용할 수 있으며, 쿼리 수행 결과를 매핑할 클래스명을 정의한다. <result>가 지정되지 않았을 경우 쿼리 수행 결과의 각 Row에 대해 칼럼명, 칼럼값을 쌍으로 Map에 put하고 각 Row별 Map들을 ArrayList에 담은 형태로 결과값을 리턴하게 된다. isCamelCase, mappingStyle 속성값에 따라 Map에 저장되는 키값이 달라짐에 유의하도록 한다. (예를 들어, mappingStyle의 속성값이 'camel'이고 조회 칼럼명이 USER_NAME인 경우 Map의 키값은 userName이 된다.)</p> <p>* attribute 설명</p> <p>length : 한 페이지에 보여질 데이터의 건수</p> <p>class : 수행 결과를 저장할 클래스명</p>	result-mapping(선택)
result-mapping	<p>해당 쿼리가 조회를 위한 SELECT문일 경우에 사용할 수 있으며, 수행 결과를 저장할 클래스와 매핑되는 테이블이 정의되지 않았을 경우 또는 수행 결과를 저장할 클래스의 속성명이 조회 칼럼명에 mappingStyle을 적용한 이름과 일치하지 않는 경우에 한해 해당하는 칼럼에 대해 매핑되는 해당 클래스의 attribute 명을 정의한다.</p> <p>* attribute 설명</p> <p>column : 조회된 칼럼명</p> <p>attribute : 정의한 칼럼에 매핑되는 클래스의 attribute명</p>	

3.Usecases

이 페이지를 통하여 다양한 Query Service 사용 방법에 대해 소개하고자 한다. 상세한 내용을 알고자 한다면, 아래 나열된 각각의 Use Case를 참고하도록 한다.

3.1.Result Mapping

다음은 Query Service를 통해 목록 조회를 수행한 이후 조회 결과를 특정 객체에 매핑하기 위한 방법들이다.

3.1.1.조회 결과 매핑이 별도로 정의되어 있지 않은 경우

조회 결과 매핑을 위해 <table-mapping/>, <result-mapping/>을 별도로 정의하지 않은 경우, 쿼리문 수행 결과는 각 결과 Row 별로 Map에 담아 ArrayList 형태로 리턴된다. 조회 결과값을 추출하기 위해서는 각 Map으로부터 get("칼럼명")을 통해 해당 칼럼의 값을 얻어낼 수 있다.

```
<queryservice>
  <queries>
    <query id="selectGeneral" isDynamic="false">
      <statement>
        SELECT * FROM TBL_CUSTOMER WHERE SSNO like ?
      </statement>
      <param type="VARCHAR" />
    </query>
  </queries>
</queryservice>
```

3.1.2.<result-mapping> 없이 <table-mapping>을 이 용할 경우

특정 클래스와 단일 테이블 사이의 매핑 정보를 정의할 때 사용하며, 테이블과 특정 클래스 간의 매핑 정보를 정의해 두면 특정 조회문의 조회 결과를 매핑할 때 별도의 <result-mapping>없이 해당 클래스 명만 <result>에 정의해 두면 되므로 XML 정의가 보다 간단해질 수 있다. 또한 <table-mapping>을 이용하면 별도 쿼리문 정의없이 객체만으로도 단건 데이터 생성/수정/삭제/조회가 가능해진다. <result class=""anyframe.sample.domain.Customer"/>와 같이 table mapping시 정의한 클래스를 이용하면 쿼리문 수행 결과는 해당 클래스의 setter 호출을 통해 저장되고, getter를 호출함으로써 결과값을 얻을 수 있게 된다.

```
<queryservice>
  <table-mapping>
    <table name="TBL_CUSTOMER"
      class="anyframe.sample.domain.Customer">
      <field-mapping>
        <dbms-column>ssno</dbms-column>
        <class-attribute>ssno</class-attribute>
      </field-mapping>
      <field-mapping>
        <dbms-column>name</dbms-column>
        <class-attribute>nm</class-attribute>
      </field-mapping>
      <field-mapping>
        <dbms-column>address</dbms-column>
```

```

        <class-attribute>addr</class-attribute>
    </field-mapping>
    <primary-key>
        <dbms-column>ssno</dbms-column>
    </primary-key>
</table>
</table-mapping>
<queries>
    <query id="select" isDynamic="false">
        <statement>
            SELECT * FROM TBL_CUSTOMER WHERE SSNO like ?
        </statement>
        <param type="VARCHAR" />
        <result class="anyframe.sample.domain.Customer"/>
    </query>
</queries>
</queryservice>

```

3.1.3.<table-mapping>,<result-mapping>없이 <result>만을 이용할 경우

매핑 대상 클래스의 속성명이 조회 칼럼명과 동일하거나 CamelCase된 형태이어서, <table-mapping>나 <result-mapping>를 통해 별도 매핑을 수행하지 않아도 되는 경우에 사용할 수 있다. 즉, isCamelCase, mappingStyle 속성값에 따라 조회된 칼럼명과 매핑되는 클래스의 속성명을 찾는다. (예를 들어, mappingStyle의 속성값이 'camel'이고 조회 칼럼명이 USER_NAME인 경우 매핑되는 속성명은 userName이 된다.)

```

<queryservice>
    <queries>
        <query id="getUser" isDynamic="false">
            <statement>
                SELECT USER_ID, USER_NAME, PASSWORD, SSN, SL_YN,
                    BIRTH_DAY, AGE, CELL_PHONE, ADDR, EMAIL, EMAIL_YN,
                    IMAGE_FILE, REG_DATE
                FROM USERS
                WHERE USER_ID = ?
            </statement>
            <param type="VARCHAR" />
            <result class="anyframe.sample.domain.User"/>
        </query>
    </queries>
</queryservice>

```

* Query 서비스는 내부적으로 쿼리문 수행으로 얻어진 조회 결과를 매핑할 때 다음과 같은 순서로 매핑 기준을 찾는다.

1. 정의된 <result-mapping> 정보가 있으면 이를 기반으로 매핑
2. 정의된 <result> 클래스에 대한 <table-mapping> 정보가 있으면 이를 기반으로 매핑
3. 정의된 <result> 클래스에 대한 정보가 있으면 이를 기반으로 매핑
4. HashMap에 mappingStyle, isCamelCase 속성값을 기반으로 매핑

3.1.4.<result-mapping>을 이용할 경우

<result-mapping>은 <table-mapping>에 정의되지 않은 클래스이면서, 조회된 칼럼명과 매핑 클래스의 속성명이 연관성이 없어 별도 매핑이 필요한 경우 사용한다. <result-mapping>에 해당 column과

attribute를 일대일로 매핑한다. <result>에 정의된 클래스의 setter 호출을 통해 저장되고, getter 호출을 통해 결과값을 얻을 수 있다.

```
<queryservice>
  <queries>
    <query id="selectUsingResultMapping" isDynamic="false">
      <statement>SELECT * FROM TBL_CUSTOMER WHERE SSNO like ?</statement>
      <param type="VARCHAR"/>
      <result class="anyframe.sample.domain.CompositionCustomer">
        <result-mapping column="NAME" attribute="nm"/>
        <result-mapping column="ADDRESS" attribute="addr"/>
      </result>
    </query>
  </queries>
</queryservice>
```



Result Mapping시 유의사항

- SELECT 문을 통해 조회된 칼럼 중 <result-mapping>을 통해 매핑하지 않은 경우에는 기본적으로 mappingStyle을 이용하여, 매핑 처리를 수행한다. 만약 정의된 Result 클래스 내에 mappingStyle에 맞는 속성명이 존재하지 않으면 매핑은 수행되지 않을 것이다.
- Result 클래스가 내부에 다른 User Defined Object를 포함하고 있는 Composite Object인 경우에는 column/attribute 사이의 매핑 정보를 정의할 때, {} 내에 ,를 구분자로 하여 정의할 수 있다. 단, attribute 정의시 반드시 내부 Object의 변수명을 그대로 사용해야 한다.

쿼리 수행 결과를 성공적으로 Result 클래스에 매핑하기 위해 매핑되는 테이블은 ID라는 이름의 칼럼명을 포함하지 않아야 함을 반드시 기억하도록 한다.

```
<query id="findwithCompositeResultMapping">
  <statement>
    select grp.GROUP_ID, grp.GROUP_NAME, cd.CODE_ID, cd.CODE_NAME,
           cd.CODE_DESC, cd.CODE_USE_YN
    from TB_CODE_GROUP grp, TB_CODE cd
    where grp.GROUP_ID = cd.GROUP_ID and cd.GROUP_ID = :groupId
  </statement>
  <result class="anyframe.core.query.vo.LocalResultMappingVO">
    <result-mapping column="{ GROUP_ID, GROUP_NAME }"
      attribute="{ group.groupId, group.codeNm }" />
    <result-mapping attribute="codeID" column="CODE_ID" />
    <result-mapping attribute="codeDescription" column="CODE_DESC" />
  </result>
</query>
```

위에서 정의한 쿼리 정의에 의해 'findWithCompositeResultMapping'의 수행 결과는 LocalResultMappingVO 객체에 담겨져 전달될 것이다. 이 때, GROUP_ID, GROUP_NAME 칼럼값은 LocalResultMappingVO 내의 group이라는 이름의 변수에 해당하는 객체의 groupId, codeNm에 각각 매핑된다. 따라서 전달된 LocalResultMappingVO로부터 GROUP_ID, GROUP_NAME 값을 추출하기 위해서는 LocalResultMappingVO.getGroup().getGroupId(), LocalResultMappingVO.getGroup().getCodeNm()을 호출하면 된다.

다음은 CodeVO라는 객체를 포함하고 있는 LocalResultMappingVO라는 객체의 일부이다.

```
public class LocalResultMappingVO implements Serializable {
  private String groupId;
  private String groupName;
```

```

private String codeID;
private String codeName;
private String codeDescription;
private String codeUseYn;
private CodeVO group;

// getter, setter...
}

```

3.1.5. 테스트 코드 Sample

다음은 앞서 제시한 매핑 XML 파일에 정의된 쿼리문을 실행하는 테스트 코드의 일부이다.

```

/**
 * Query 서비스를 통해 DB에 입력된 데이터를 조회하는 테스트 코드
 */
public void selectCustomer() throws Exception{
    QueryService queryService = (QueryService) context.getBean("queryService");

    //find() : 매핑 XML 파일에 정의되어 있는 query id를 이용하여 SELECT를 실행한다.

    //별도 Result Mapping을 정의하지 않은 경우
    ArrayList rsquery = (ArrayList) queryService
        .find("selectGeneral", new Object[] { "%12345%" });
    Map hsRsquery = new HashMap();
    for( int i = 0 ; i < rsquery.size() ; i ++ ){
        hsRsquery = (Map) rsquery.get(i);
        String name = (String) hsRsquery.get("name");
    }

    // table-mapping을 정의한 경우
    Collection rsqueryUsingTableMapping = queryService.find(
        "selectUsingTableMapping", new Object[] { "%12345%" });
    Iterator rsqueryItr = rsqueryUsingTableMapping.iterator();
    while (rsqueryItr.hasNext()) {
        Customer customer = (Customer) rsqueryItr.next();
        String name = customer.getNm();
    }

    // result-mapping을 정의한 경우
    Collection rsqueryUsingResultMapping = queryService.find("selectUsingResultMapping"
        , new Object[] { "%12345%" });
    Iterator rsqueryItr_01 = rsqueryUsingResultMapping.iterator();
    while (rsqueryItr_01.hasNext()) {
        CompositionCustomer compositionCustomer
            = (CompositionCustomer) rsqueryItr_01.next();
        String name = compositionCustomer.getCompositionName();
    }

    // result class만 정의한 경우
    Collection rsqueryUsingOnlyResultClass = queryService.find(
        "selectUsingOnlyResultClass", new Object[] { "%12345%" });
    Iterator rsqueryItr_02 = rsqueryUsingOnlyResultClass.iterator();
    while (rsqueryItr_02.hasNext()) {
        CamelCasedCustomer camelCasedCustomer
            = (CamelCasedCustomer) rsqueryItr_02.next();
        String name = camelCasedCustomer.getName();
    }
}

```

3.2.Embedded SQL

다음은 Embedded SQL을 사용하는 경우로, 매핑 XML 파일에 별도로 쿼리문을 정의해 두지 않고도 특정 쿼리문을 소스 코드 내에 직접 입력하여 실행할 수 있다.

3.2.1.속성 정의 파일 Sample

다음은 Query 서비스를 정의한 context-query.xml 파일의 일부이다.

```
<bean id="queryService" class="org.anyframe.query.impl.QueryServiceImpl">
  <property name="jdbcTemplate" ref="jdbcTemplate"/>
  <property name="sqlRepository" ref="sqlLoader"/>
  종략...
</bean>
<bean id="jdbcTemplate" class="org.anyframe.query.impl.jdbc.PagingJdbcTemplate">
  <property name="dataSource" ref="dataSource" />
</bean>
```

3.2.2.테스트 코드 Sample

다음은 매핑 XML 파일에 별도 쿼리문을 정의하지 않고 쿼리문을 직접 입력하여 실행하는 테스트 코드의 일부이다. 다음에서 볼 수 있듯이 Query 서비스는 매핑 XML 파일없이도 INSERT, SELECT, UPDATE, DELETE를 실행할 수 있도록 지원한다.

```
/**
 * 테스트 코드를 실행하기 위한 data table 생성
 */
public void insertCustomer() throws Exception{
    QueryService queryService = (QueryService) context.getBean("queryService");

    //createBySQL() : 매핑 XML 파일에 쿼리문이 정의되어 있지 않을 때 INSERT를 실행한다.
    int result = queryService.createBySQL(
        "insert into TBL_CUSTOMER values (?, ?, ?)",
        new String[] { "VARCHAR", "VARCHAR", "VARCHAR" },
        new Object[] { "1234567890123", "GilDongHong", "Seoul" });
    if ( result == -1 ){
        throw new Exception("Insert Query failed");
    }
}

/**
 * Query 서비스를 통해 DB에 입력된 데이터를 조회하는 테스트 코드
 */
public void selectCustomer() throws Exception{
    QueryService queryService = (QueryService) context.getBean("queryService");

    //findBySQL() : 매핑 XML 파일에 쿼리문이 정의되어 있지 않을 때 SELECT를 실행한다.
    Collection result = queryService.findBySQL(
        "select NAME, ADDRESS from TBL_CUSTOMER where SSNO like ?",
        new String[] { "VARCHAR" },
        new Object[] { "%4567890123" });
    Iterator resultItr = result.iterator();
    while( resultItr.hasNext() ){
        Map resultMap = (Map) resultItr.next();
        resultMap.get("name");
    }
}
```

```

        System.out.println("result.size() : " + result.size());
    }

    /**
     * Query 서비스를 통해 DB에 입력된 데이터를 수정하는 테스트 코드
     */
    public void updateCustomer() throws Exception{
        QueryService queryService = (QueryService) context.getBean("queryService");

        //updateBySQL() : 매핑 XML 파일에 쿼리문이 정의되어 있지 않을 때 UPDATE를 실행한다.
        int result = queryService.updateBySQL(
            "update TBL_CUSTOMER set NAME=? where SSNO=?",
            new String[] { "VARCHAR", "VARCHAR" },
            new Object[] { "Anonymous", "1234567890123" });

        System.out.println("result : " + result);
    }

    /**
     * Query 서비스를 통해 DB에 입력된 데이터를 삭제하는 테스트 코드
     */
    public void deleteCustomer() throws Exception{
        QueryService queryService = (QueryService) context.getBean("queryService");

        //removeBySQL() : 매핑 XML 파일에 쿼리문이 정의되어 있지 않을 때 DELETE를 실행한다.
        int result = queryService.removeBySQL(
            "delete from TBL_CUSTOMER where SSNO=?",
            new String[] { "VARCHAR" },
            new Object[] { "1234567890123" });

        System.out.println("result : " + result);
    }
}

```

3.3.OR Mapping

다음은 OR mapping을 사용하는 예로, query 정의없이 object만을 사용해서 기본적인 INSERT, UPDATE, DELETE, SELECT를 실행한다.

3.3.1.속성 정의 파일 Sample

다음은 Query 서비스를 정의한 context-query.xml과 Query 서비스에서 읽어들이 매핑 XML 파일의 위치를 정의한 context-query-sqlloader.xml 파일의 일부이다.

```

<bean id="queryService" class="org.anyframe.query.impl.QueryServiceImpl">
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
    <property name="sqlRepository" ref="sqlLoader"/>
    종략...
</bean>
<bean id="jdbcTemplate" class="org.anyframe.query.impl.jdbc.PagingJdbcTemplate">
    <property name="dataSource" ref="dataSource" />
</bean>

```

```

<bean name="sqlLoader" class="org.anyframe.query.impl.config.loader.SQLLoader">
    <property name="mappingFiles">
        <value>
            <!-- xml files in folder -->
            file:./src/test/resources/foldermappings/**/testcase-*.xml,
            file:./src/test/resources/dynamicreload/testcase-queries-dynamicreload.xml,

```

```

<!-- xml files in classpath -->
  classpath*/mappings/testcase-queries-dynamic.xml,
  classpath*/mappings/testcase-queries-general.xml,
  classpath*/mappings/testcase-queries-resultmapping.xml,
  classpath*/mappings/testcase-queries-resultsetmapper.xml,
  classpath*/mappings/testcase-table-mappings.xml,
  classpath*/mappings/testcase-queries-extended.xml,
  classpath*/mappings/testcase-queries-mappingstyle.xml,
<!-- xml files in jar -->
  classpath*/jarmappings/testcase-queries-lob.xml
</value>
</property>
중략 ...
</bean>

```

3.3.2.매핑 XML 파일 Sample

다음은 앞서 정의한 Query 서비스를 통해 로드된 mapping-query-ormapping.xml 파일의 일부로, 테이블 TBL_ORMapping와 ORMapping 클래스 사이의 매핑 정보를 포함하고 있다.

```

<queryservice>
  <table-mapping>
    <table name="TBL_ORMapping"
      class="anyframe.sample.domain.ORMapping">
      <field-mapping>
        <dbms-column>id</dbms-column>
        <class-attribute>id</class-attribute>
      </field-mapping>
      <field-mapping>
        <dbms-column>name</dbms-column>
        <class-attribute>nm</class-attribute>
      </field-mapping>
      <field-mapping>
        <dbms-column>address</dbms-column>
        <class-attribute>addr</class-attribute>
      </field-mapping>
      <primary-key>
        <dbms-column>id</dbms-column>
      </primary-key>
    </table>
  </table-mapping>
</queryservice>

```

3.3.3.OR Mapping시 사용할 매핑 클래스 Sample

다음은 앞서 언급한 매핑 XML 파일에 정의된 OR Mapping 정보를 기반으로 특정 테이블에 데이터를 INSERT, UPDATE, SELECT할 때 사용되는 클래스 ORMapping.java 내용의 일부이다.

```

public class ORMapping implements Serializable {

  public String id;
  public String nm;
  public String addr;

  public ORMapping() {
  }

  public ORMapping(String i, String n, String a) {
    id = i;
  }
}

```

```

        nm = n;
        addr = a;
    }

    // getter, setter ...
}

```

3.3.4.테스트 코드 Sample

다음은 매핑 XML 파일에 별도 쿼리문을 정의하지 않고 객체만을 이용하여 INSERT, SELECT, UPDATE, DELETE를 실행하는 테스트 코드의 일부이다. 매핑 클래스에 필요한 값을 셋팅하여 Query 서비스에 전달함으로써 해당하는 쿼리문을 실행시킬 수 있다.

```

/**
 * Query 서비스를 통해 DB에 신규 데이터를 입력하는 테스트 코드
 */
public void insertORMapping() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    ORMapping ormapping = new ORMapping("1234567890123", "HonggilDong", "Ansan");

    // ORMapping 객체에 초기값을 셋팅하고 이 객체를 통해 INSERT를 실행한다.
    // 다음 코드에 의해 실행되는 쿼리문은
    // INSERT INTO TBL_ORMapping (address ,name ,id )
    // values ( '1234567890123', 'HonggilDong', 'Ansan' )
    int result = queryService.create(ormapping);
    System.out.println("result : " + result);
}

/**
 * Query 서비스를 통해 DB에 입력된 데이터를 조회하는 테스트 코드
 */
public void selectORMapping() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    ORMapping ormapping = new ORMapping();
    ormapping.setId("1234567890123");

    // primary-key로 지정한 칼럼에 매핑되는 attribute에 값을 입력하고
    // find를 실행하면 primary-key를 조회 조건으로 하는 SELECT를 실행한다.
    // 다음 코드에 의해 실행되는 쿼리문은
    // SELECT NAME, ADDRESS FROM TBL_ORMAPPING WHERE ID = '1234567890123'
    Collection result = queryService.find(ormapping);
    if ( result.size() != 1 ){
        throw new Exception("Select failed");
    }
}

/**
 * Query 서비스를 통해 DB에 입력된 데이터를 수정하는 테스트 코드
 */
public void updateORMapping() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    ORMapping ormapping = new ORMapping("1234567890123", "HonggilDong", "Seoul");

    // 다음 코드에 의해 실행되는 쿼리문은
    // Update TBL_ORMapping set address = 'Seoul' ,name = 'HonggilDong'
    // where id = '1234567890123'
    int result = queryService.update(ormapping);
}

```

```

    if ( result == -1 ){
        throw new Exception("Update failed");
    }
}

/**
 * Query 서비스를 통해 DB에 입력된 데이터를 삭제하는 테스트 코드
 */
public void deleteORMapping() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    ORMapping ormapping = new ORMapping();
    ormapping.setId("1234567890123");

    // 다음 코드에 의해 실행되는 쿼리문은
    // DELETE FROM TBL_ORMapping where id = 1234567890123
    int result = queryService.remove(ormapping);
    if ( result == -1 ){
        throw new Exception("Update failed");
    }
}
}

```

3.4.Dynamic Query

Query 서비스는 text 치환, named parameter 형태 등을 통해 운영 시 입력된 조건 값에 따라 동적으로 변경되는 쿼리문 정의를 지원한다. 이를 위해서는 다음과 같은 syntax를 사용할 수 있다.

- **:ParameterName** : 특정 쿼리문에 입력되어야 할 변수를 지정할 때 사용한다.
- **{{치환 문자열 키}}** : 치환 문자열 키에 해당하는 값으로 치환되어야 하는 부분에 정의한다.
- **#if ~ (#elseif) ~ #end** : 조건 분기가 필요한 부분에 정의한다.
- **# foreach ~ #end** : Loop가 필요한 부분에 정의한다.
- **\$velocityCount** : foreach 구문내의 Loop index를 체크하고자 하는 부분에 정의한다.

3.4.1.속성 정의 파일 Sample

다음은 Query 서비스를 정의한 context-query.xml 과 Query 서비스에서 읽어들이 매핑 XML 파일의 위치를 정의한 context-query-sqlloader.xml 파일의 일부이다.

```

<bean id="queryService" class="org.anyframe.query.impl.QueryServiceImpl">
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
    <property name="sqlRepository" ref="sqlLoader"/>
    중략...
</bean>
<bean id="jdbcTemplate" class="org.anyframe.query.impl.jdbc.PagingJdbcTemplate">
    <property name="dataSource" ref="dataSource" />
</bean>

```

```

<bean name="sqlLoader" class="org.anyframe.query.impl.config.loader.SQLLoader">
    <config:configuration>
        <filename>classpath:/query/mapping-query-dynamic.xml</filename>
        중략...
    </config:configuration>
</bean>

```

3.4.2.매핑 XML 파일 Sample

다음은 앞서 정의한 Query 서비스를 통해 로드된 mapping-query-dynaminc.xml의 일부로, 여러 유형의 dynamic 쿼리문을 포함하고 있다.

```
<queries>
  <query id="getLogonIdByNamedParam" isDynamic="true">
    <statement>
      SELECT LOGON_ID
      FROM TBL_USER
      WHERE LOGON_ID BETWEEN :lowId AND :highId
    </statement>
    <param name="lowId" type="VARCHAR"/>
    <param name="highId" type="VARCHAR"/>
  </query>
  <query id="getEmployeesByTextReplacement" isDynamic="true">
    <statement>
      SELECT LOGON_ID
      FROM {{schema}}
      ORDER BY {{sortColumn}}
    </statement>
  </query>
  <query id="getLogonIdByIf" isDynamic="true">
    <statement>
      SELECT LOGON_ID
      FROM TBL_USER
      #if ($id && !$id.equals(""))
      WHERE LOGON_ID like '% ' || :logonId || '%'
      #end
      ORDER BY {{sortColumn}}
    </statement>
    <param name="logonId" type="VARCHAR"/>
  </query>
  <query id="getLogonIdByForeach" isDynamic="true">
    <statement>
      SELECT LOGON_ID, NAME
      FROM TBL_USER
      WHERE LOGON_ID IN (
        #foreach ($logonId in $logonIdList)
          #if ($velocityCount == 1 )
            '$logonId'
          #else
            , '$logonId'
          #end
        #end
      )
      ORDER BY NAME
    </statement>
  </query>
</queries>
```

3.4.3.테스트 코드 Sample

다음은 앞서 언급한 매핑 XML 파일에 정의된 parameter 값에 따라 동적으로 변경 가능한 쿼리문들을 실행하는 테스트 코드의 일부이다.

```
/**
 * Named Parameter를 이용해서 query id가 'getLogonIdByNamedParam'인 Dynamic 쿼리문을
 * 실행한다. getLogonIdByNamedParam : key가 lowId, highId인 Named Parameter의 값을
```

```

* key=value 형태로 Query 서비스에 전달하면 Query 서비스는 해당 value를 PreparedStatement
* 에 셋팅하고 해당 쿼리를 실행한다. 이 메소드에서는 "lowId = a", "highId = z"라는
* parameter를 Object[] 형태로 Query 서비스에 전달하고 있으며, 이 때 실행되는 쿼리는
* 다음과 같다.
* SELECT LOGON_ID FROM TBL_USER WHERE LOGON_ID BETWEEN 'a' AND 'z'
*/
public void dynamicQueryUsingNamedParameter() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    Object[] iVal = new Object[2];
    iVal[0] = new Object[]{"lowId", "a"};
    iVal[1] = new Object[]{"highId", "z"};

    ArrayList rtList
        = (ArrayList)(queryService.find("getLogonIdByNamedParam", iVal));
    System.out.println("rtList.size() : " + rtList.size());
}

/**
 * {{치환문자열키}}를 이용해서 query id가 'getEmployeesByTextReplacement'인 Dynamic
 * 쿼리를 실행한다. getEmployeesByTextReplacement : key가 schema, sortColumn인
 * 치환문자열의 값을 key=value 형태로 Query 서비스에 전달하면 Query 서비스는 해당 value를
 * 문자열로 그대로 치환해서 사용한다. 이 메소드에서는 "schema=TBL_USER", "sortColumn=NAME"
 * 라는 parameter를 Object[] 형태로 Query 서비스에 전달하고 있으며, 이 때 실행되는 쿼리는
 * 다음과 같다.
 * SELECT * FROM TBL_USER ORDER BY NAME
 */
public void dynamicQueryUsingTextreplace() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    Object[] iVal = new Object[2];
    iVal[0] = new Object[]{"schema", "TBL_USER"};
    iVal[1] = new Object[]{"sortColumn", "NAME"};

    ArrayList rtList
        = (ArrayList)(queryService.find("getEmployeesByTextReplacement", iVal));
    System.out.println("rtList.size() : " + rtList.size());
}

/**
 * 조건분기를 위한 #if를 이용해서 query id가 'getLogonIdByIf'인 Dynamic 쿼리를 실행한다.
 * getLogonIdByIf : key가 id인 parameter의 값이 Null이 아니고, 빈 문자열도 아니라면, #if문
 * 내에 포함된 WHERE절이 실행될 쿼리에 포함된다. (#if문이 끝나는 지점에는 반드시 #end를
 * 정의해주어야 함에 유의하자.)
 * 또한, WHERE절 내에서는 '%' || :logonId || '%'와 같은 형태의 문장을 사용하고 있는데
 * 이것은 Named Parameter의 값에 앞뒤로 %를 붙인 형태의 문자열을 만들어 내기 위함이다.
 * (이 때, '%', ||, :logonId 사이에는 빈 칸을 두어야 WHERE절이 정상적으로 동작한다.)
 * 이 메소드에서는 "id=test", "sortColumn=NAME" 라는 parameter를 Object[] 형태로 Query
 * 서비스에 전달하고 있으며 실행되는 쿼리는 다음과 같다.
 * SELECT LOGON_ID
 * FROM TBL_USER
 * WHERE LOGON_ID like '%test%'
 * ORDER BY NAME
 */
public void dynamicQueryUsingCondition() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    Object[] iVal = new Object[3];
    iVal[0] = new Object[]{"id", "yes"};
    iVal[1] = new Object[]{"logonId", "test"};
    iVal[2] = new Object[]{"sortColumn", "NAME"};

```

```

ArrayList rtList
    = (ArrayList)(queryService.find("getLogonIdByIf", ival));
System.out.println("rtList.size() : " + rtList.size());
}

/**
 * Loop를 위한 #foreach를 이용해서 query id가 'getLogonIdByForeach'인 Dynamic 쿼리문을
 * 실행한다. getLogonIdByForeach : key가 logonIdList인 parameter의 값은 List 형태이며,
 * List에서 순서대로 추출된 값들은 내부적으로 logonId라는 변수에 셋팅된다. logonId는
 * Query 서비스에 전달되어야 하는 입력 parameter가 아니다.
 * velocityCount가 1인 경우 즉, 첫번째 Loop일 경우 logonIdList의 첫번째 값이 그대로
 * 추가되고 그 이후부터는 ,를 붙인 값이 추가되게 된다. (#foreach, #if문이 끝나는 지점에는
 * 반드시 #end를 정의해주어야 함에 유의하자.) 이 메소드에서는 logonIdList의 값이 "admin",
 * "test"라는 두 개의 문자열로 구성된 ArrayList를 Object[] 형태로 Query 서비스에 전달하고
 * 있으며 실행되는 쿼리문은 다음과 같다.
 * SELECT LOGON_ID, NAME
 * FROM TBL_USER
 * WHERE LOGON_ID IN ('admin', 'test')
 * ORDER BY NAME
 */
public void dynamicQueryUsingLoop() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    List logonIdList = new ArrayList();
    logonIdList.add("admin");
    logonIdList.add("test");

    Object[] ival = new Object[]{"logonIdList", logonIdList};

    ArrayList rtList
        = (ArrayList) (queryService.find("getLogonIdByForeach", new Object[]{" ival }));

    if (rtList.size() != 1) {
        throw new Exception("Dynamic Query Using Condition failed");
    }
}

```



Dynamic Query 정의시 유의사항

Query Service를 이용하여 목록 조회 결과를 특정 객체에 매핑하기 위해 쿼리문 실행시마다 매핑 정보 셋팅을 위한 로직이 반복 수행된다. Query Service의 성능을 개선하기 위해 Anyframe 4.0.0 이후부터 이 로직을 변경하여 한 번 처리한 Result Mapping 정보를 SQLLoader에서 queryId를 기준으로 관리하도록 하였다. 따라서 조회되는 칼럼 정보가 쿼리문 실행시마다 변경될 수 있는 SELECT, FROM 절이 가변적으로 변경되는 쿼리문인 경우에는 Result Mapping 정보가 쿼리문 실행시마다 변경될 수 있으므로 별도 Mapper 클래스를 통해 처리될 수 있도록 해야 한다. Anyframe에서 org.anyframe.query.impl.jdbc.mapper 패키지에 하위에 제공되는 Mapper 클래스는 다음과 같은 역할을 수행한다.

표 3.1. Mapper Class

Class Name	Description
DefaultCallbackResultSetMapper	[기본 적용] 특정 쿼리문에 대해 별도 mapper가 정의되어 있지 않고, 조회 결과를 Map 형태로 전달받는 경우에 Query Service에서 기본적으로 적용하는 Mapper. 처음으로 실행하는 쿼리문의 조회 칼럼 정보는 저장되며, 다음 실행시에는 저장된 정보를 기반으로 조회 칼럼 매핑 수행.
CallbackResultSetMapper	조회 결과를 Map 형태로 전달받는 경우 쿼리문 실행 시점마다 조회 칼럼 매핑을 처리하는 Mapper. SELECT/FROM 절이 동적으로 구성되어 있어 쿼리문 실행시마다 조회 결과가 달라지는 쿼리문인 경우 활용 가능하며 해당하는 쿼리문의 mapper 속성값으로 정의.
DefaultReflectionResultSetMapper	[기본 적용] 특정 쿼리문에 대해 별도 mapper가 정의되어 있지 않고, 조회 결과를 특정 VO 형태로 전달받는 경우 Query Service에서 기본적으로 적용하는 Mapper. 처음으로 실행하는 쿼리문의 조회 칼럼 정보는 저장되며, 다음 실행시에는 저장된 정보를 기반으로 조회 칼럼 매핑 수행.
ReflectionResultSetMapper	조회 결과를 특정 VO 형태로 전달받는 경우 쿼리문 실행 시점마다 칼럼 매핑을 처리하는 Mapper. SELECT/FROM 절이 동적으로 구성되어 있어 쿼리문 실행시마다 조회 결과가 달라지는 쿼리문인 경우 활용 가능하며 해당하는 쿼리문의 mapper 속성값으로 정의.
MappingStyleColumnMapper	Stored Procedure 수행시 CamelCase, Lower, Upper와 같은 Result Mapping Style을 적용하기 위해 Query 서비스에서 기본적으로 사용하는 Mapper

다음은 조건에 따라 FROM 절이 변경되는 쿼리문의 예이다. 앞서 언급한 바와 같이 QueryService에서는 기본적으로 queryId를 기준으로 한번 실행된 쿼리문에 대해 Result Mapping 정보를 저장해두고 있기 때문에 다음과 같이 SELECT 또는 FROM 절이 변경되는 쿼리문의 경우 처음 실행된 쿼리문과 이후 실행되는 쿼리문이 달라질 수 있게 된다. 따라서 현재 실행된 쿼리문의 결과와 queryId를 기준으로 저장된 Result Mapping 정보가 맞지 않아 원치 않은 결과가 도출될 수 있을 것이다. 이와 같은 경우에는 다음과 같이 <result mapper="..."/>를 이용하여 해당 쿼리문을 별도 Mapper를 통해 처리될 수 있도록 정의해 주어야 함을 기억하도록 하자.

'findUsers' 쿼리문의 경우 쿼리 수행 결과를 Map으로 전달받기 위해 CallbackResultSetMapper를 mapper로 정의하고 있음을 알 수 있다.

```
<query id="findUsers" isDynamic="true">
  <statement>
    SELECT LOGON_ID FROM {{schema}} ORDER BY {{sortColumn}}
  </statement>
  <result mapper="org.anyframe.query.impl.jdbc.mapper.
    callbackResultSetMapper"/>
</query>
```

만일 'findUsers' 쿼리문의 수행 결과를 User Defined Object로 전달받기 위해서는 앞서 언급한 ReflectionResultSetMapper를 mapper로 정의하고, 전달받고자하는 클래스를 명시해주면 된다.

```
<query id="findUsers" isDynamic="true">
  <statement>
```

```

SELECT LOGON_ID FROM {{schema}} ORDER BY {{sortColumn}}
</statement>
<result class="anyframe.core.query.vo.UsersVO"
mapper="org.anyframe.query.impl.jdbc.mapper.
ReflectionResultSetMapper"/>
</query>

```

위에서 제시한 Mapper 외에 특정 쿼리문에 적합한 Custom ResultSetMapper 클래스를 별도 구현할 수 있으며 이것은 Result Mapper 정의 방법을 참고하도록 한다.

3.5.Pagination

Query 서비스를 통해 조회 유형의 쿼리문 실행시 전체 결과를 조회하지 않고 해당 페이지에 속한 데이터만 조회할 수 있으므로, 효율적이고 성능에 유리하다. Query 서비스는 페이징 처리를 위해 별도의 로직이나 특정 DB에 종속적인 쿼리문을 요구하지 않는다. 다만 조회하고자 하는 페이지의 번호와 한 페이지에 보여지는 데이터의 개수만 입력 parameter로 전달하면 된다.

3.5.1.속성 정의 파일 Sample

다음은 Query 서비스를 정의한 context-query.xml 과 Query 서비스에서 읽어들이 매핑 XML 파일의 위치를 정의한 context-query-sqlloader.xml파일의 일부이다. (* 조회 유형의 쿼리에 대해 페이징 처리를 수행하기 위해서는 해당 DB에 맞는 PagingSQLGenerator를 추가 셋팅해주어야 함에 유의하도록 한다.)



Pagination시 유의 사항

QueryService 속성 정의시에는 반드시 DBMS에 적합한 PagingSQLGenerator를 셋팅해 주어야 한다. 적절한 PagingSQLGenerator가 존재하지 않는 경우에는 QueryService에서 제공하는 org.anyframe.query.impl.jdbc.generator.DefaultPagingSQLGenerator를 사용할 수 있으나, DefaultPagingSQLGenerator를 이용하여 findXXX() 메소드를 실행하면 QueryService 내부적으로 조건에 해당하는 모든 데이터를 fetch한 이후 ResultSet Cursor의 위치를 이동시키는 방식으로 특정 페이지에 속한 데이터를 걸러낸다. 이 때 ResultSet Cursor를 움직이는 로직에서 상당한 시간이 소요되어 다량의 데이터 조회시 성능에 심각한 영향을 끼칠 수 있다. 따라서, DefaultPagingSQLGenerator 사용은 권장하지 않는다.

```

<bean id="queryService" class="org.anyframe.query.impl.QueryServiceImpl">
  <property name="jdbcTemplate" ref="jdbcTemplate"/>
  <property name="sqlRepository" ref="sqlLoader"/>
  <property name="pagingSQLGenerator" ref="pagingSQLGenerator"/>
  종략...
</bean>
<bean id="jdbcTemplate" class="org.anyframe.query.impl.jdbc.PagingJdbcTemplate">
  <property name="dataSource" ref="dataSource" />
</bean>
<bean id="pagingSQLGenerator"
  class="org.anyframe.query.impl.jdbc.generator.OraclePagingSQLGenerator"/>

```

```

<bean name="sqlLoader" class="org.anyframe.query.impl.config.loader.SQLLoader">
  <config:configuration>
    <filename>classpath:/query/mapping-query-pagination.xml</filename>
    종략...
  </config:configuration>
</bean>

```

3.5.2.매핑 XML 파일 Sample

다음은 앞서 정의한 Query 서비스를 통해 로드된 mapping-query-pagination.xml의 일부로, 테이블 매핑 정보와 쿼리문을 포함하고 있다.

```
<query id="selectUsingPagination">
  <statement>
    SELECT NAME FROM TBL_CUSTOMER WHERE SSNO like ?
  </statement>
  <param type="VARCHAR" />
  <result class="anyframe.sample.domain.Customer"/>
</query>
```

3.5.3.테스트 코드 Sample

다음은 앞서 언급한 매핑 XML 파일에 정의된 SELECT 쿼리문을 실행하는 테스트 코드의 일부로, 페이지 처리된 수행 결과를 얻어 내기 위한 테스트 로직을 포함하고 있다.

```
/**
 * SELECT 수행 결과를 특정 페이지별로 조회하는 테스트 코드
 * 이 메소드에서는 TBL_CUSTOMER 테이블에 여러 데이터가 입력되었을 때, 한 페이지에 표현하고자
 * 하는 데이터의 개수(5)와 조회 페이지 번호(3)를 입력하여 특정 페이지에 속한 데이터만
 * 조회해 본다.
 */
public void selectCustomer() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    /** findWithRowCount() : 매핑 XML 파일에 정의되어 있는 query id를 이용하여 SELECT를
     * 실행한다. findWithRowCount()는 한번의 호출로 해당 SELECT로
     * 얻을 수 있는 전체 데이터의 개수와 특정 페이지에 해당하는
     * 결과값들을 얻어 올 수 있다.
     */
    HashMap resultMap = queryService.findWithRowCount("selectUsingPagination",
        new Object[] { "%1234%" }, 3, 5);

    Customer customer = new Customer();
    Map rsMap = new HashMap();

    // 특정 페이지에 속한 결과값들 추출
    List resultList = (List) resultMap.get(QueryService.LIST);
    for(int i = 0 ; i < resultList.size() ; i++){
        customer = (Customer)resultList.get(i);
        customer.getNm();
    }

    // 해당 SELECT로 얻어질 수 있는 전체 데이터의 개수 추출
    int totalSize = ((Long) resultMap.get(QueryService.COUNT)).intValue();
    if (resultList.size() != 5 || totalSize != 15) {
        throw new Exception("Select query failed");
    }
}
```

다음은 Query 서비스에서 제공하는 페이지징 처리 기능을 수행하는 findXXX() 메소드 사용시 알아두어야 할 사항들이다. **public Collection find...(int pageIndex, int pageSize);**

1. 페이지 번호가 1보다 작으면 결과 데이터가 없다. (**Anyframe 3.2.0 이후**)
2. 페이지 번호가 1보다 작으면 첫번째 페이지가 조회된다. (**Anyframe 3.2.0 이전**)

3. 한 페이지에 보여줘야 하는 데이터의 개수는 0보다 커야 한다.
4. 페이지 번호와 한 페이지에 보여줘야 하는 데이터의 개수를 사용하여 계산한 값이 전체 결과 데이터의 개수보다 크면 결과 데이터가 없다.(**Anyframe 3.2.0 이후**)
5. 페이지 번호와 한 페이지에 보여줘야 하는 데이터의 개수를 사용하여 계산한 값이 전체 결과 데이터의 개수보다 크면 마지막 페이지가 조회된다. (**Anyframe 3.2.0 이전**)
6. 한 페이지에 보여줘야 하는 데이터의 개수가 생략될 경우 매핑 XML 파일의 <result>에 별도로 정의해 주어야 한다.

```
<query id="selectUsingPaging" isDynamic="false">
  <statement>
    SELECT NAME FROM TBL_CUSTOMER WHERE SSNO like ?
  </statement>
  <param type="VARCHAR" />
  <result length="10" class="anyframe.sample.domain.Customer"/>
</query>
```

3.6.Batch Update

Query 서비스는 JDBC 2.0 batch updates를 사용해서 한번의 호출로 여러 건의 데이터를 INSERT, UPDATE, DELETE할 수 있게 한다.

3.6.1.속성 정의 파일 Sample

다음은 Query 서비스를 정의한 context-query.xml과 Query 서비스에서 읽어들이 매핑 XML 파일의 위치를 정의한 context-query-sqlloader.xml 파일의 일부이다.

```
<bean id="queryService" class="org.anyframe.query.impl.QueryServiceImpl">
  <property name="jdbcTemplate" ref="jdbcTemplate"/>
  <property name="sqlRepository" ref="sqlLoader"/>
  종략...
</bean>
<bean id="jdbcTemplate" class="org.anyframe.query.impl.jdbc.PagingJdbcTemplate">
  <property name="dataSource" ref="dataSource" />
</bean>
```

```
<bean name="sqlLoader" class="org.anyframe.query.impl.config.loader.SQLLoader">
  <config:configuration>
    <filename>classpath:/query/mapping-query-batch.xml</filename>
    종략...
  </config:configuration>
</bean>
```

3.6.2.매핑 XML 파일 Sample

다음은 앞서 정의한 Query 서비스를 통해 로드된 mapping-query-batch.xml 파일의 일부로, 테이블 매핑 정보와 Batch로 처리할 쿼리문을 포함하고 있다.

```
<queryservice>
  <table-mapping>
    <table name="TBL_CUSTOMER"
      class="anyframe.sample.domain.Customer">
      <field-mapping>
        <dbms-column>ssno</dbms-column>
```

```

        <class-attribute>ssno</class-attribute>
    </field-mapping>
    <field-mapping>
        <dbms-column>name</dbms-column>
        <class-attribute>nm</class-attribute>
    </field-mapping>
    <field-mapping>
        <dbms-column>address</dbms-column>
        <class-attribute>addr</class-attribute>
    </field-mapping>
    <primary-key>
        <dbms-column>ssno</dbms-column>
    </primary-key>
</table>
</table-mapping>
<queries>
    <query id="insertbatch">
        <statement>
            INSERT INTO TBL_CUSTOMER ( ssno, name, address ) VALUES (?, ?, ?)
        </statement>
        <param type="VARCHAR" />
        <param type="VARCHAR" />
        <param type="VARCHAR" />
    </query>
</queries>
</queryservice>

```

3.6.3. 테스트 코드 Sample

다음은 앞서 언급한 매핑 XML 파일에 정의된 INSERT 쿼리문을 Batch Update를 이용해 실행하는 테스트 코드의 일부이다. 입력 parameter를 LIST 형태로 담아서 Query 서비스에 전달하면, Query 서비스는 한 번의 실행으로 전달된 List 개수 만큼의 쿼리문을 실행한다.

```

/**
 * 매핑 XML 파일에 정의된 쿼리문을 batch update를 이용해 실행.
 */
public void insertCustomer() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    //ArrayList에 입력할 데이터를 저장한다.
    ArrayList args = new ArrayList();
    Object[] arg = new Object[3];
    arg[0] = "1234567890123";
    arg[1] = "KimMinso";
    arg[2] = "Ansan";
    args.add(arg);
    arg = new Object[3];
    arg[0] = "1234567890124";
    arg[1] = "LeeSungwook";
    arg[2] = "Seoul";
    args.add(arg);
    arg = new Object[3];
    arg[0] = "1234567890125";
    arg[1] = "ParkHeejin";
    arg[2] = "Seoul";
    args.add(arg);

    //ArrayList에 3개의 Object가 포함되어 있으므로 query id가 insertbatch인
    //쿼리문이 3번 실행된다.
    int[] results = queryService.batchUpdate("insertbatch", args);
}

```

```

    for (int i = 0; i < results.length; i++) {
        if (results[i] == -1) {
            throw new Exception("Batch Insert falied");
        }
    }
}

/**
 * 매핑 XML 파일에 쿼리문이 정의되어 있지 않을 때, 실행할 쿼리문을 직접 입력하여
 * batch update를 실행
 */
public void insertCustomerBySQLQuery() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    //실행 할 쿼리문을 정의
    String sql = "INSERT INTO TBL_CUSTOMER ( ssno, name, address ) VALUES (?,?,?)";

    //입력할 parameter의 SQL Type을 정의
    String[] types = new String[3];
    types[0] = "VARCHAR";
    types[1] = "VARCHAR";
    types[2] = "VARCHAR";

    //ArrayList에 입력할 데이터들 저장한다.
    ArrayList args = new ArrayList();
    Object[] arg = new Object[3];
    arg[0] = "1234567890126";
    arg[1] = "HongGildong";
    arg[2] = "Suwon";
    args.add(arg);
    arg = new Object[3];
    arg[0] = "1234567890127";
    arg[1] = "LeeSoonsin";
    arg[2] = "Seongnam";
    args.add(arg);
    arg = new Object[3];
    arg[0] = "1234567890128";
    arg[1] = "ChoiMinsu";
    arg[2] = "Seoul";
    args.add(arg);

    //ArrayList에 3개의 Object가 포함되어 있으므로 해당 쿼리문이 3번 실행된다.
    int[] results = queryService.batchUpdateBySQL(sql, types, args);

    for (int i = 0; i < results.length; i++) {
        if (results[i] == -1) {
            throw new Exception("BatchInsertBySQL falied");
        }
    }
}

/**
 * OR Mapping을 이용해서 batch update를 실행
 */
public void insertCustomerByObject() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    ArrayList args = new ArrayList();
    Customer customer = new Customer();

    customer.setSsno("1234567890129");
    customer.setNm("Smith");

```

```

customer.setAddr("LA");
args.add(customer);

customer = new Customer();
customer.setSsno("1234567890130");
customer.setNm("Brown");
customer.setAddr("Newyork");
args.add(customer);

customer = new Customer();
customer.setSsno("1234567890131");
customer.setNm("Eugene");
customer.setAddr("Boston");
args.add(customer);

// 실행되는 쿼리문은 기본적인 INSERT문인
// insert into TBL_CUSTOMER (ssno ,address ,name ) values ( ?, ?, ? )이다.
int[] results = queryService.batchCreate(args);

for (int i = 0; i < results.length; i++) {
    if (results[i] == -1) {
        throw new Exception("BatchInsertByObject failed");
    }
}
}

```



Oracle-style Batch Update

Oracle-style의 batch update가 필요한 경우 Anyframe에서 제공하는 OraclePagingJdbcTemplate를 사용해야 한다. OraclePagingJdbcTemplate는 입력된 쿼리문에 대해 내부적으로 PreparedStatement 형태로 변경하여 쿼리문을 batch로 처리하고 있다. 그런데 입력된 쿼리문이 Stored Procedure, Function 등일 경우 PreparedStatement 형태로 batch 처리하고자 할 때 SQLException이 발생하게 된다. 이를 해결하기 위해 Anyframe에서는 CallableStatement 형태로 처리되어야 하는 쿼리문(Stored Procedure, Function 등)에 대해 batch update 할 수 있도록 OraclePagingJdbcTemplate 내에 별도의 메소드를 추가하였다. 이 메소드는 QueryService의 batchExecute(), batchExecuteBySQL() 메소드를 통해 내부적으로 호출된다.

따라서 Stored Procedure, Function 등에 대한 Oracle-style의 batch update를 위해서는 QueryService의 batchExecute(), batchExecuteBySQL() 메소드를 호출하여 처리해야 함을 기억하도록 하자.

3.7.Callable Statement

CallableStatement는 표준 DDL, DML이 아닌 DB의 Stored Procedure, Function 등을 호출할 때 사용한다. Stored Procedure는 쿼리문을 하나의 파일 형태로 만들거나 DB에 저장해두고 함수처럼 호출해서 사용하는 것으로, 성능, 코드의 독립성, 보안성 등의 다양한 이점을 제공한다.

3.7.1.속성 정의 파일 Sample

다음은 Query 서비스를 정의한 context-query.xml 과 Query 서비스에서 읽어들이 매핑 XML 파일의 위치를 정의한 context-query-sqlloader.xml 파일의 일부이다.

```

<bean name="oracle_queryservice" class="org.anyframe.query.impl.QueryServiceImpl">
  <property name="jdbcTemplate" ref="jdbcTemplate"/>
  <property name="sqlRepository" ref="sqlLoader"/>
  종략...

```

```

</bean>
<bean id="jdbcTemplate" class="org.anyframe.query.impl.jdbc.PagingJdbcTemplate">
  <property name="dataSource" ref="dataSource" />
</bean>

```

```

<bean name="sqlLoader" class="org.anyframe.query.impl.config.loader.SQLLoader">
  <config:configuration>
    <filename>classpath:/query/mapping-query-callablestatement.xml</filename>
    중략...
  </config:configuration>
</bean>

```

3.7.2.매핑 XML 파일 Sample

다음은 앞서 정의한 Query 서비스를 통해 로드된 mapping-query-callablestatement.xml 파일의 일부로, DB에 생성되어 있는 test_function이라는 Function을 실행시키기 위한 쿼리문을 포함하고 있다.

```

<queryservice>
  <queries>
    <query id="callFunction">
      <statement>{? = call test_function(?)}</statement>
      <param type="NUMERIC" binding="OUT" name="outVal" />
      <param type="NUMERIC" binding="IN" name="inVal" />
    </query>
  </queries>
</queryservice>

```

3.7.3.테스트 코드 Sample

다음은 앞서 언급한 매핑 XML 파일에 정의된 Function을 호출하기 위한 쿼리문을 실행하는 테스트 코드의 일부이다. Query 서비스를 통해 DB Function을 실행시키기 위해서는 사용하는 DB에 해당 Function이 미리 생성되어 있어야 한다.

```

/**
 * 매핑 XML 파일에 정의되어 있는 query id를 이용해 Function을 호출한다.
 */
public void callableStatementFunction() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    HashMap inVal = new HashMap();
    inVal.put("inVal", new Integer(10));

    Map results = queryService.execute("callFunction", inVal);
    BigDecimal rtVal = (BigDecimal) results.get("outVal");

    if (rtVal.intValue() != 1){
        throw new Exception("testCallableStatementProcedure failed");
    }
}

/**
 * 매핑 XML 파일에 정의되어 있지 않은 경우에도 특정 Function을 호출할 수 있다.
 * Map executeBySQL(String sql, String[] types, String[] names,
 * String[] bindings, Map values)
 */
public void callableStatementBySQL() throws Exception {
    QueryService queryService = (QueryService) context.getBean("oracle_queryservice");

    String sql = "{? = call test_function()}";

```

```

String[] types = { "NUMERIC", "NUMERIC" };
String[] bindings = { "OUT", "IN" };
String[] names = { "outVal", "inVal" };

HashMap inVal = new HashMap();
inVal.put("inVal", new Integer(10));

Map results = queryService.executeBySQL(sql, types, names, bindings, inVal);
BigDecimal rtVal = (BigDecimal) results.get("outVal");

if (rtVal.intValue() != 1 ){
    throw new Exception("testCallableStatementBySQL failed");
}
}

/**
 * 매핑 XML 파일에 정의되어 있지 않은 경우에도 특정 Function을 호출할 수 있다.
 * 이 메소드에서는 페이징 처리된 조회 결과를 얻고자 한다.
 * Map executeBySQL(String sql, String[] types, String[] names,
 * String[] bindings, Map values, int pageIndex, int pageSize)
 */
public void callableStatementBySQLwithPaging() throws Exception {
    QueryService queryService = (QueryService) context.getBean("oracle_queryservice");

    String sql = "{? = call test_function(?)}";
    String[] types = { "NUMERIC", "NUMERIC" };
    String[] bindings = { "OUT", "IN" };
    String[] names = { "outVal", "inVal" };

    HashMap inVal = new HashMap();
    inVal.put("inVal", new Integer(10));

    Map results = queryService.executeBySQL(sql, types, names, bindings, inVal, 1, 1);
    BigDecimal rtVal = (BigDecimal) results.get("outVal");

    if (rtVal.intValue() != 1 ){
        throw new Exception("testCallableStatementBySQLwithPaging failed");
    }
}
}

```

3.8.CLOB, BLOB

기본적으로 LOB 유형의 데이터와 다른 유형의 데이터를 다루는 방식은 다르지 않다. 다만, LOB 유형의 데이터 조작이 필요한 경우에는 Query Service 속성 정의시 Lob Handler를 추가로 정의해 주어야 함을 기억하도록 하자. LobHandler 속성 정의에 대한 자세한 내용은 [lobHandler] 속성을 참고하도록 한다. 여기에서는 Oracle 9i 이상, Oracle 8i로 구분하여 LOB 유형의 데이터를 다루는 방법에 대해 살펴볼 것이다.

3.8.1.Oracle 9i 이상일 경우

3.8.1.1.속성 정의 파일 Sample

다음은 Query 서비스를 정의한 context-query.xml 과 Query 서비스에서 읽어들이 매핑 XML 파일의 위치를 정의한 context-query-sqlloader.xml 파일의 일부이다.

```

<bean name="oracle_queryservice" class="org.anyframe.query.impl.QueryServiceImpl">
  <property name="jdbcTemplate" ref="jdbcTemplate"/>
  <property name="sqlRepository" ref="sqlLoader"/>
  <property name="lobHandler" ref="lobHandler"/>

```

```

    종략...
</bean>
<bean id="jdbcTemplate" class="org.anyframe.query.impl.jdbc.PagingJdbcTemplate">
  <property name="dataSource" ref="dataSource" />
</bean>
<bean id="nativeJdbcExtractor"
  class="org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor"
  lazy-init="true"/>
<bean id="lobHandler" class="org.springframework.jdbc.support.lob.OracleLobHandler"
  lazy-init="true">
  <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/>
</bean>

```

```

<bean name="sqlLoader" class="org.anyframe.query.impl.config.loader.SQLLoader">
  <config:configuration>
    <filename>classpath:/query/mapping-query-lob.xml</filename>
    종략...
  </config:configuration>
</bean>

```

3.8.1.2. 매핑 XML 파일 Sample

다음은 앞서 정의한 Query 서비스를 통해 로드된 mapping-query-lob.xml 파일의 일부로, 테이블 매핑 정보와 LOB 유형의 데이터를 다루는 다양한 쿼리문들을 포함하고 있다.

```

<queries>
  <query id="insertLOB">
    <statement>insert into longVarchar values(?,?,?)</statement>
    <param type="INTEGER"/>
    <param type="BLOB"/>
    <param type="CLOB"/>
  </query>
  <query id="selectLOB">
    <statement>select myblob, myclob from longVarchar where count = ?</statement>
    <param type="INTEGER"/>
  </query>
  <query id="updateLOB">
    <statement>update longVarchar set myblob = ?, myclob = ? WHERE count = ?</statement>
    <param type="BLOB"/>
    <param type="CLOB"/>
    <param type="INTEGER"/>
  </query>
  <query id="deleteLOB">
    <statement>delete from longVarchar WHERE count = ?</statement>
    <param type="INTEGER"/>
  </query>
</queries>

```

3.8.1.3. 테스트 코드 Sample

다음은 앞서 언급한 매핑 XML 파일에 정의된 LOB 유형의 데이터를 INSERT, SELECT, UPDATE, DELETE 하는 테스트 코드의 일부이다.

```

/**
 * Query 서비스를 통해 DB에 신규 LOB 유형의 데이터를 입력하는 테스트 코드
 */
public void insertLOB() throws Exception{
  QueryService queryService = (QueryService) context.getBean("oracle_queryservice");

  String strVal1 = "0무궁화꽃이피었습니다";

```

```

String strVal2 = "1무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이
    피었습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무
    궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었
strVal2 += "2무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었습
    니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었
    습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었
    습니다\n";

// skip
strVal2 += "30무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었
    습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었
    습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었습니다무궁화꽃이피었
    습니다\n";

int result = queryService.create("insertLOB"
    , new Object[] { new Integer(5), strVal1.getBytes() , strVal2 });
System.out.println("result : " + result);
}

/**
 * Query 서비스를 통해 DB에 입력된 LOB 유형의 데이터를 조회하는 테스트 코드
 */
public void selectLOB() throws Exception {
    QueryService queryService = (QueryService) context.getBean("oracle_queryservice");

    Collection result = queryService.find("selectLOB", new Object[] { new Integer(5) });

    Iterator resultItr = result.iterator();
    while (resultItr.hasNext()) {
        Map binary = (Map) resultItr.next();
        String myCLOB = (String) binary.get("myclob");
        String myBLOB = new String((byte[]) binary.get("myblob"));
    }

    System.out.println("result : " + result);
}

/**
 * Query 서비스를 통해 DB에 입력된 LOB 유형의 데이터를 수정하는 테스트 코드
 */
public void updateLOB() throws Exception{
    QueryService queryService = (QueryService) context.getBean("oracle_queryservice");

    String strVal1 = "0장미꽃이피었습니다";
    String strVal2 = "1장미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장
        미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장
        미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장
        미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장\n";
    strVal2 += "2장미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장
        미꽃이피었습니다 장미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장
        미꽃이피었습니다장미꽃이피었습니다\n";

    // skip
    strVal2 += "30장미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장
        미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장미꽃이피었습니다장
        미꽃이피었습니다장미꽃이피었습니다\n";

    int result = queryService.create("updateLOB"
        , new Object[] { strVal1.getBytes() , strVal2, new Integer(5)});
}

/**
 * Query 서비스를 통해 DB에 입력된 LOB 유형의 데이터를 삭제하는 테스트 코드
 */
public void deleteLOB() throws Exception{
    QueryService queryService = (QueryService) context.getBean("oracle_queryservice");
    queryService.remove("deleteLOB", new Object[]{new Integer(5)});
}

```

}

3.8.2.Oracle 8i일 경우

3.8.2.1.속성 정의 파일 Sample

Spring에서 제공하는 Oracle LobHandler는 9i 이상에서만 적용 가능하므로, Oracle 8i 기반에서 LOB 유형의 데이터를 다루기 위해서는 Query 서비스 속성 정의시에 Anyframe에서 제공하는 LobHandler를 셋팅해 주어야 한다. 다음은 Query 서비스를 정의한 context-query.xml 파일의 일부로, Oracle 8i용으로 제공된 Oracle8iLobHandler를 정의한 부분이다.

```
<bean id="lobHandler" class="org.anyframe.query.impl.jdbc.lob.Oracle8iLobHandler"
      lazy-init="true">
  <constructor-arg value="net.sf.log4jdbc.ResultSetSpy"/>
  <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/>
</bean>
```

DriverSpy를 써서 실행되는 쿼리문을 로그로 남기고 있을 경우에는 LobHandler의 <constructor-arg>를 정의하고, 아닐 경우에는 제거해도 된다.

3.8.2.2.DataSource 서비스 속성 정의 파일 Sample

반드시 defaultAutoCommit의 속성을 false로 변경한다.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
  <property name="url" value="jdbc:oracle:thin:@localhost:1521:oracle"/>
  <property name="username" value="system"/>
  <property name="password" value="manager"/>
  <property name="defaultAutoCommit" value="false"/>
</bean>
```

3.8.2.3.TransactionManager 속성 정의 파일 Sample

DataSource 서비스의 defaultAutoCommit 속성을 false로 설정했으므로, TransactionManager 셋팅이 필요하다.

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

<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRES_NEW"
              rollback-for="Exception" />
  </tx:attributes>
</tx:advice>

<aop:config>
  <aop:pointcut id="executionMethods"
              expression="execution(* org.anyframe.query..impl.QueryServiceImpl.*(..))" />
  <aop:advisor advice-ref="txAdvice"
              pointcut-ref="executionMethods" />
</aop:config>
```

3.8.2.4.매핑 XML 파일 Sample

신규 LOB 유형의 데이터를 INSERT하는 쿼리문의 경우에는 기존 쿼리문 정의와 다르게 lobStatement tag를 추가 정의해 주어야 한다.

```
<query id="insertBlobClobwithOra8i" isDynamic="false">
  <statement>insert into binary values(?,empty_blob(),empty_clob())</statement>
  <param type="INTEGER"/>
  <lobStatement>
    <statement>select myclob, myblob
      from binary where bin_id = ? for update</statement>
    <param type="INTEGER"/>
  </lobStatement>
</query>
```

3.9.Named Parameter 'vo' 활용

매핑 XML 파일에 정의되어 있는 쿼리문을 실행시키기 위해 필요한 입력 Parameter들을 Query 서비스에 전달할 때, 개별 Parameter들을 포함하고 있는 VO 유형의 객체로 전달 가능하다. 따라서 입력 Parameter 개수나 이름이 변경되더라도, Query 서비스 사용 로직의 변경없이 매핑 XML 내의 해당되는 쿼리문만 변경함으로써 변경 사항 적용이 가능해지므로 쿼리문 변경이 용이해지는 장점을 제공한다.

3.9.1.속성 정의 파일 Sample

다음은 Query 서비스를 정의한 context-query.xml 과 Query 서비스에서 읽어들이 매핑 XML 파일의 위치를 정의한 context-query-sqlloader.xml 파일의 일부이다.

```
<bean id="queryService" class="org.anyframe.query.impl.QueryServiceImpl">
  <property name="jdbcTemplate" ref="jdbcTemplate"/>
  <property name="sqlRepository" ref="sqlLoader"/>
  종략...
</bean>
<bean id="jdbcTemplate" class="org.anyframe.query.impl.jdbc.PagingJdbcTemplate">
  <property name="dataSource" ref="dataSource" />
</bean>
```

```
<bean name="sqlLoader" class="org.anyframe.query.impl.config.loader.SQLLoader">
  <config:configuration>
    <filename>classpath:/query/mapping-query-namedparamvo.xml</filename>
    종략...
  </config:configuration>
</bean>
```

3.9.2.매핑 XML 파일 Sample

다음은 앞서 정의한 Query 서비스를 통해 mapping-query-namedparamvo.xml 로, Named Parameter로 전달된 'vo'를 활용한 다양한 쿼리문들을 포함하고 있다. Parameter명이 'vo'인 객체로부터 . 뒤에 붙은 이름에 대한 getter 메소드를 호출한 결과를 PreparedStatement에 셋팅한 후, 해당 쿼리문을 실행하게 된다.

```
<queryservice>
  <queries>
    <query id="insertCategory">
      <statement>
        INSERT INTO TBL_CATEGORY
```

```

        (CATEGORY_NO, CATEGORY_NAME, CATEGORY_DESC, USE_YN, REG_ID,
        REG_DATE, MODIFY_ID, MODIFY_DATE)
        VALUES (:vo.categoryNo, :vo.categoryName, :vo.categoryDesc,
        :vo.useYn, :vo.regId, sysdate, :vo.regId, sysdate)
    </statement>
</query>
<query id="updateCategory">
    <statement>
        UPDATE TBL_CATEGORY
        SET CATEGORY_NAME = :vo.categoryName,
        CATEGORY_DESC = :vo.categoryDesc,
        USE_YN = :vo.useYn, MODIFY_ID = :vo.modifyId,
        MODIFY_DATE = sysdate
        WHERE CATEGORY_NO = :vo.categoryNo
    </statement>
</query>
<query id="deleteCategory">
    <statement>
        delete from TBL_CATEGORY where CATEGORY_NO = :vo.categoryNo
    </statement>
</query>
<query id="findCategoryList">
    <statement>
        SELECT CATEGORY_NO, CATEGORY_NAME, CATEGORY_DESC, USE_YN, REG_ID
        FROM TBL_CATEGORY
        WHERE CATEGORY_NO like :vo.categoryNo
    </statement>
    <result class="anyframe.sample.domain.Category"></result>
</query>
<query id="findCategory">
    <statement>
        SELECT CATEGORY_NO, CATEGORY_NAME, CATEGORY_DESC, USE_YN, REG_ID
        FROM TBL_CATEGORY
        WHERE CATEGORY_NO = :vo.categoryNo
    </statement>
    <result class="anyframe.sample.domain.Category"></result>
</query>
</queries>
</queryservice>

```

3.9.3. 테스트 코드 Sample

다음은 앞서 언급한 매핑 XML 파일에 정의된 INSERT, SELECT, UPDATE, DELETE 쿼리문을 실행하는 테스트 코드의 일부이다. Query 서비스에 입력 Parameter 전달시 'vo'라는 Parameter명에 Category 또는 SearchVO 객체를 매핑시키고 있음을 알 수 있다.

```

/**
 * QueryService에 특정 쿼리의 입력 Parameter값을 전달할 때, named parameter 형태로 transfer
 * object를 전달한다. QueryService에서는 전달받은 객체의 getter 메소드를 호출하여 INSERT
 * 쿼리문의 Parameter값을 셋팅하고 실행한다.
 */
public Category testInsertCategory() throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    Category category = makeCategory();

    Object[] args = new Object[] { "vo", category };
    int result = queryService.create("insertCategory",
        new Object[] { args });
}

```

```
        if (result != 1) {
            throw new Exception("Insert a new category failed");
        }

        return category;
    }

/**
 * QueryService에 특정 쿼리의 입력 Parameter값을 전달할 때, named parameter 형태로 transfer
 * object를 전달한다. QueryService에서는 전달받은 객체의 getter 메소드를 호출하여 UPDATE
 * 쿼리문의 Parameter값을 셋팅하고 실행한다.
 */
public void updateCategory(Category category) throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");
    category.setCategoryName("testUpdate");

    Object[] args = new Object[] { "vo", category };
    int result = queryService.update("updateCategory",
        new Object[] { args });

    if (result != 1) {
        throw new Exception("Update category failed");
    }
}

/**
 * QueryService에 특정 쿼리의 입력 Parameter값을 전달할 때, named parameter 형태로 transfer
 * object를 전달한다. QueryService에서는 전달받은 객체의 getter 메소드를 호출하여 DELETE
 * 쿼리문의 Parameter값을 셋팅하고 실행한다.
 */
public void deleteCategory(Category category) throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    Object[] args = new Object[] { "vo", category };
    int result = queryService.remove("deleteCategory",
        new Object[] { args });

    if (result != 1) {
        throw new Exception("Delete category failed");
    }

    args = new Object[] { "vo", category };
    Collection rtCollection = queryService.find("findCategory",
        new Object[] { args });

    if (rtCollection.size() != 0) {
        throw new Exception("Find categorylist failed");
    }
}

/**
 * QueryService에 특정 쿼리의 입력 Parameter값을 전달할 때, named parameter 형태로 transfer
 * object를 전달한다. QueryService에서는 전달받은 객체의 getter 메소드를 호출하여 SELECT
 * 쿼리문의 Parameter값을 셋팅하고 실행한다.
 */
public void findCategory(String categoryNo, String categoryName)
    throws Exception {
    QueryService queryService = (QueryService) context.getBean("queryService");

    Category searchVO = new Category();
    searchVO.setCategoryNo(categoryNo);
}
```

```

Object[] args = new Object[] { "vo", searchVO };
Collection rtCollection = queryService.find("findCategory",
    new Object[] { args });

if (rtCollection.size() != 1) {
    throw new Exception("Find category failed");
}

Category category = (Category) rtCollection.iterator().next();
if (!(categoryNo.equals(category.getCategoryNo()) && categoryName
    .equals(category.getCategoryName())) {
    throw new Exception("Find category failed");
}
}
}

```

3.10.extends AbstractDao

일반적으로 Dao(Data Access Object) 클래스에서는 Query 서비스를 활용하여 데이터 접근 로직을 처리한다. Query 서비스에서는 이러한 Dao 클래스를 간편하게 개발할 수 있도록 하기 위해 org.anyfram.query.dao.AbstractDao 클래스를 제공하고 있으며 다음과 같은 특징을 지닌다.

- 입력된 문자열에 정해진 **Prefix, Postfix**를 조합하여 **query id**를 만든다.

INSERT문의 경우 'createXXX', UPDATE문인 경우 'updateXXX', DELETE문인 경우 'removeXXX', 단건 조회 SELECT문인 경우 'findXXXByPk', 다건 조회 SELECT문인 경우 'findXXXList'가 query id가 된다. 따라서 Naming Rule을 정하고 이에 맞게 query id를 정의하면, 공통되는 기본 문자열만을 사용하여 원하는 쿼리문을 실행할 수 있게 된다.

- **query id** 조합을 위한 **Prefix, Postfix**는 수정이 가능하다.

createId(Default='create'), updateId(Default='update'), removeId(Default='remove'), findPrefix(Default='find'), findListPostfix(Default='List'), findByPkPostfix(Default='ByPk') 속성에 대한 setter 메소드를 제공하므로 수정 적용이 가능하다.

- 입력 **Parameter**를 **VO, Map, List, Object[]** 형태에 담아 처리할 수 있다.

입력된 Parameter는 Named Parameter로 처리되기 때문에 VO의 Map, VO의 ArrayList 등에 대한 처리도 가능해진다. 따라서 Dao 코드가 단순해질 수 있게 된다.

- 단건 데이터 조회 결과 처리를 위해 필요한 **공통 로직**을 수행한다.

Query 서비스의 find(), findBySQL() 메소드를 호출하면 SELECT문 실행 결과를 Collection 객체에 담아 전달한다. 따라서 조회 결과가 단건인 경우 Collection으로부터 단건 데이터를 꺼내서 전달하는 로직이 추가되어야 한다.

```

if (collection == null || collection.size() == 0)
    return null;
return collection.iterator().next();

```

AbstractDao에서 제공하는 findByPk(...) 메소드를 호출하면 위에서 언급한 추가 로직을 중복 구현하지 않고 단건 데이터를 얻을 수 있게 된다.

- **UI** 개발 편의를 위해서 다건 데이터 조회 결과를 별도의 **Page** 객체에 저장하는 로직을 수행한다.

Query 서비스의 findWithRowCount() 메소드를 호출하면, SELECT문 실행 결과를 Map 객체에 담아 전달하며, 개발자는 UI 개발 편의를 위해 Map 객체로부터 필요한 데이터를 추출하여 별도의 Page 객체에 담기 위한 로직을 추가해야 한다.

```

List resultList = (List)queryMap.get(QueryService.LIST);

```

```
int totalSize = ((Long) queryMap.get(QueryService.COUNT)).intValue();
return new Page(resultList, (new Integer(pageIndex)).intValue(),
    totalSize, pageUnit, pageSize);
```

AbstractDao에서 제공하는 findListWithPaging() 메소드를 호출하면 위에서 언급한 추가 로직을 중복 구현하지 않고도 org.anyframe.pagination.Page 형태의 객체를 얻을 수 있게 된다.

3.10.1.매핑 XML 파일 Sample

다음은 Query 서비스를 통해 로드된 mapping-query-abstractdao.xml 파일의 일부로, Named Parameter 를 포함한 다양한 쿼리문들을 포함하고 있다.

```
<queryservice>
  <queries>
    <query id="createProduct">
      <statement>
        INSERT INTO PRODUCT
          ( PROD_NO, PROD_NAME, SELLER_ID, CATEGORY_NO, PROD_DETAIL,
            MANUFACTURE_DAY, AS_YN, SELL_QUANTITY, SELL_AMOUNT, IMAGE_FILE,
            REG_DATE )
        VALUES (:vo.prodNo, :vo.prodName, :vo.sellerId, :vo.category.categoryNo,
          :vo.prodDetail, :vo.manufactureDay, :vo.asYn, :vo.sellQuantity,
          :vo.sellAmount, :vo.imageFile, sysdate)
      </statement>
    </query>

    <query id="updateProduct">
      <statement>
        UPDATE PRODUCT
        SET PROD_NAME = :vo.ProdName,
          PROD_DETAIL = :vo.prodDetail,
          MANUFACTURE_DAY = :vo.manufactureDay,
          AS_YN = :vo.asYn,
          SELL_QUANTITY = :vo.sellQuantity,
          SELL_AMOUNT = :vo.sellAmount,
          REG_DATE = sysdate
        WHERE PROD_NO = :vo.ProdNo
      </statement>
    </query>

    <query id="findProductByPk">
      <statement>
        SELECT PROD_NO, PROD_NAME, SELLER_ID, CATEGORY_NO, PROD_DETAIL,
          MANUFACTURE_DAY, AS_YN, SELL_QUANTITY, SELL_AMOUNT, IMAGE_FILE,
          REG_DATE
        FROM PRODUCT
        WHERE PROD_NO = :vo.ProdNo
      </statement>
      <result class="anyframe.sample.domain.Product">
        <result-mapping column="{CATEGORY_NO}" attribute="{category.categoryNo}" />
      </result>
    </query>

    <query id="removeProduct">
      <statement>
        DELETE FROM PRODUCT
        WHERE PROD_NO = :vo.prodNo
      </statement>
    </query>
```

```

<query id="findProductList">
  <statement>
    SELECT product.PROD_NO, product.PROD_NAME, product.SELLER_ID,
           product.CATEGORY_NO, product.PROD_DETAIL, product.MANUFACTURE_DAY,
           product.AS_YN, product.SELL_QUANTITY, product.SELL_AMOUNT,
           product.IMAGE_FILE, product.REG_DATE
    FROM PRODUCT product
    WHERE
      product.PROD_NAME like :vo.prodName
      AND product.AS_YN = :vo.asYn
    ORDER BY product.PROD_NO DESC
  </statement>
  <result class="anyframe.sample.domain.Product">
    <result-mapping column="{CATEGORY_NO}" attribute="{category.categoryNo}" />
  </result>
</query>
</queries>
</queryservice>

```

3.10.2.Dao 클래스 코드 Sample

다음은 AbstractDao를 상속받아 구현한 ProductDaoImpl.java 클래스의 일부이다. AbstractDao에서 제공하는 메소드를 호출함으로써 Product 정보를 INSERT, UPDATE, DELETE, SELECT하는 역할을 수행하는 ProductDaoImpl 로직이 보다 간단해졌음을 알 수 있을 것이다. AbstractDao 클래스를 상속받은 경우 Query 서비스의 API를 직접 호출하려면 getQueryService() 메소드 호출을 통해 Query 서비스를 얻어내어 활용하면 된다.

```

public class ProductDaoImpl extends AbstractDao {
  // query id가 'createProduct'인 INSERT문을 실행
  public int createProduct(Product vo) throws Exception {
    return create("Product", vo);
  }

  // query id가 'deleteProduct'인 DELETE문을 실행
  public int deleteProduct(Product vo) throws Exception {
    return remove("Product", vo);
  }

  // query id가 'findProductByPk'인 SELECT문을 실행하고
  // 조회 결과인 Product 객체를 전달
  public Product selectProduct(String prodNo) throws Exception {
    Product vo = new Product();
    vo.setProdNo(prodNo);
    Product resultVo = (Product) findByPk("Product", vo);
    return resultVo;
  }

  // query id가 'findProductList'인 SELECT문을 실행하고
  // 조회 결과인 Page 객체를 전달
  public Page selectProductList(Product vo, int pageIndex, int pageSize,
    int pageUnit) throws Exception {
    return findListWithPaging("Product", vo, pageIndex,
      pageSize, pageUnit);
  }

  // query id가 'updateProduct'인 UPDATE문을 실행
  public int updateProduct(Product vo) throws Exception {
    return update("Product", vo);
  }
}

```

3.10.3.Dao 클래스 속성 정의 파일 Sample

다음은 AbstractDao 클래스를 상속받은 ProductDaoImpl 클래스의 속성을 정의한 context-service.xml 파일의 일부이다. AbstractDao 클래스에서 내부적으로 Query 서비스를 사용하므로 Query 서비스의 Bean Id를 속성으로 셋팅해주도록 한다.

```
<bean id="productDao" class="anyframe.sample.query.sales.dao.impl.ProductDaoImpl">
  <property name="queryService" ref="queryService" />
  <property name="propertiesService" ref="propertiesService" />
</bean>
```

3.10.4.Dao 클래스 테스트 코드 Sample

다음은 앞서 언급한 매핑 XML 파일에 정의된 INSERT, SELECT, UPDATE, DELETE 쿼리문을 실행하는 테스트 코드의 일부이다.

```
/**
 * AbstractDao를 통해 DB에 신규 데이터를 입력하는 테스트 코드
 */
public void insertProduct() throws Exception {
    Category category = categoryService.get("CATEGORY-00004");
    Product product = new Product();
    product.setProdName("sample.sportsone");
    product.setCategory(category);
    product.setProdDetail("sports one detail");
    product.setSellerId("woos41");
    product.setASyn("Y");
    product.setManufactureDay("20081225");
    product.setSellAmount(new Long(50));
    product.setSellQuantity(new Long(50));

    // 매핑 XML 파일에 정의되어 있는 query id를 이용하여 INSERT를 실행한다.
    productDao.create(product);
}

/**
 * AbstractDao를 통해 DB에 입력된 데이터를 조회하는 테스트 코드
 */
public void selectProduct(String prodNo) throws Exception {
    // 매핑 XML 파일에 정의되어 있는 query id를 이용하여 SELECT를 실행한다.
    Product vo = productDao.get(prodNo);
}

/**
 * AbstractDao를 통해 DB에 입력된 데이터를 페이징 처리하기 위한 테스트 코드
 */
public void selectProductList() throws Exception {
    ProductSearchVO searchVO = new ProductSearchVO();
    searchVO.setSearchCondition("0");
    searchVO.setSearchKeyword("sample.sportsone");
    searchVO.setPageIndex(1);
    searchVO.setPageSize(2);

    // 매핑 XML 파일에 정의되어 있는 query id를 이용하여 SELECT를 실행하고,
    // 페이징 처리된 실행 결과를 얻는다.
    Page products = productDao.getPagingList(searchVO);
}

/**
```

```

* AbstractDao를 통해 DB에 입력된 데이터를 수정하는 테스트 코드
*/
public void updateProduct(Product product) throws Exception {
    // 매핑 XML 파일에 정의되어 있는 query id를 이용하여 UPDATE를 실행한다.
    product.setProdName("sportsone-update");
    product.setProdDetail("sports one detail-update");

    productDao.update(product);
}

/**
* AbstractDao를 통해 DB에 입력된 데이터를 삭제하는 테스트 코드
*/
public void deleteProduct(String prodNo) throws Exception {
    // 매핑 XML 파일에 정의되어 있는 query id를 이용하여 DELETE를 실행한다.
    productDao.remove(prodNo);
}

```

3.11.implements ResultSetMapper

일반적으로 Query 서비스는 매핑 XML 파일에 정의되어 있는 쿼리문을 실행한 후, 그 실행 결과를 <result> 내에 정의된 객체나 HashMap에 담아 전달한다. 그런데 실행 결과를 <result> 내에 정의된 객체에 담기 위해서는 Java Reflection API 호출이 발생하므로 대량의 데이터를 처리할 때 성능 저하가 예상된다. 또한 실행 결과를 HashMap에 담는 경우에는 Java Reflection API 호출없이, 칼럼명을 키값으로 하여 칼럼값을 셋팅하나 특정 메소드 호출을 통해 주고 받는 데이터가 명확히 식별되지 않게 되어 어플리케이션 분석 및 변경시 불리할 수 있다. QueryService에서는 이러한 점을 보완하기 위해서 ResultSetMapper 실행을 지원한다. 즉, 쿼리문 실행 결과 처리를 Query 서비스에 위임하지 않고, 별도 구현된 ResultSetMapper를 통해 직접 처리할 수 있도록 지원한다.

3.11.1.속성 정의 파일 Sample

다음은 Query 서비스를 정의한 context-query.xml 과 Query 서비스에서 읽어들이 매핑 XML 파일의 위치를 정의한 context-query-sqlloader.xml 파일의 일부이다.

```

<bean id="queryService" class="org.anyframe.query.impl.QueryServiceImpl">
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
    <property name="sqlRepository" ref="sqlLoader"/>
    종략...
</bean>
<bean id="jdbcTemplate" class="org.anyframe.query.impl.jdbc.PagingJdbcTemplate">
    <property name="dataSource" ref="dataSource" />
</bean>

```

```

<bean name="sqlLoader" class="org.anyframe.query.impl.config.loader.SQLLoader">
    <config:configuration>
        <filename>classpath:/query/mapping-query-resultsetmapper.xml</filename>
        종략...
    </config:configuration>
</bean>

```

3.11.2.매핑 XML 파일 Sample

다음은 앞서 정의한 Query 서비스를 통해 로드된 mapping-query-resultsetmapper.xml의 일부로, 쿼리문을 포함하고 있다. Query Id가 findCustomerWithResultSetMapper인 쿼리문 실행 결과는 사용자가 정의한 anyframe.sample.query.CustomerMapper를 통해 처리되도록 정의하고 있음을 알 수 있다.

```

<queries>
  <query id="findCustomerWithResultSetMapper" isDynamic="false">
    <statement>select NAME, ADDRESS from TB_CUSTOMER where SSNO like ?</statement>
    <param type="VARCHAR"/>
    <result mapper="anyframe.sample.query.CustomerMapper"/>
  </query>
</queries>

```

3.11.3.ResultSetMapper 코드 Sample

사용자는 다음과 같은 순서로 ResultSetMapper를 구현하면 된다.

1. Query 서비스에서 제공하는 인터페이스 org.anyframe.query.ResultSetMapper를 implements한 클래스 정의
2. ResultSet을 입력 인자로 하는 콜백 메소드 mapRow() 정의
3. mapRow() 메소드 내에서 ResultSet을 이용하여 원하는 객체에 실행 결과 셋팅하고 실행 결과를 return

다음은 앞서 언급한 매핑 XML 파일에 정의된 CustomerMapper.java의 일부이다.

```

public class CustomerMapper implements ResultSetMapper {

    public Object mapRow(ResultSet resultSet) throws SQLException {
        Customer customer = new Customer();
        customer.setNm(resultSet.getString("name"));
        customer.setAddr(resultSet.getString("address"));
        return customer;
    }
}

```

3.11.4.테스트 코드 Sample

다음은 앞서 언급한 매핑 XML 파일에 정의된 SELECT 쿼리문을 실행하는 테스트 코드의 일부이다.

```

/**
 * QueryService의 find() 메소드를 호출하여 매핑 XML에 정의된 쿼리문을 실행시키고, 매핑 XML에
 * 정의된
 * IResultSetMapper 유형의 Mapper를 이용하여, 결과값이 매핑되는지 체크하기 위한 테스트 코드
 */
public void findWithCustomResultSetMapper() throws Exception {
    QueryService queryService = (QueryService) context
        .getBean("queryService");

    // execute query
    Collection rtList = queryService.find(
        "findCustomerWithResultSetMapper", new Object[] { "%123456%" });
    // assert a size of result
    if (rtList.size() != 3)
        throw new Exception("Fail to select with custom ResultSetMapper.");

    // assert in detail
    Iterator resultItr = rtList.iterator();
    while (resultItr.hasNext()) {
        Customer customer = (Customer) resultItr.next();
        if (!customer.getAddr().equals("Seoul"))
            throw new Exception("Fail to compare result in detail.");
    }
}

```

```
}
```