

Spring 기반

nexacroplatform Server

개발자 가이드

v 1.0



|  |  |  |  |
| --- | --- | --- | --- |
| **문서 개정이력표** | | | |
| **문서명** | | Spring 기반 nexacroplatform Server 개발자 가이드 | |
| 버전 | 날짜 | 내용 | 작성자 |
| 1.0 | 2015.08.26 | nexacro platform Server 개발자 가이드 작성 |  |

저작권 및 면책조항

본 문서는 기술상의 부정확함 또는 글자상의 오류를 포함하고 있으며, (주)투비소프트는 여기에 관한 어떠한 책임도 지지 않습니다.

본 문서의 내용은 정기적으로 변경 또는 추가 될 수 있으며, 그에 따른 변경본은 본 문서에 ‘New Edition’의 형태로 순차적으로 추가될 것입니다. (주)투비소프트는 사전 공지 없이 언제든지 본 문서에 기재된 제품 및 프로그램을 변경 또는 수정할 수 있습니다.

목 차

[1. 개요 6](#_Toc434400135)

[2. Spring 기반 – nexacro platform 7](#_Toc434400136)

[2.1. 구성 7](#_Toc434400137)

[2.1.1. NexacroInterceptor 7](#_Toc434400138)

[2.1.2. NexacroMethodArgumentResolver 7](#_Toc434400139)

[2.1.3. NexacroHandlerMethodReturnValueHandler 7](#_Toc434400140)

[2.1.4. NexacroView 7](#_Toc434400141)

[2.1.5. NexacroMappingExceptionResolver 7](#_Toc434400142)

[2.2. Data 변환 9](#_Toc434400143)

[2.2.1. PlatformData 9](#_Toc434400144)

[2.2.2. Data 변환 10](#_Toc434400145)

[2.2.3. Data Type Mapping 13](#_Toc434400146)

[2.3. 기능 명세 14](#_Toc434400147)

[2.3.1. 설정 14](#_Toc434400148)

[2.3.2. Spring 적용 17](#_Toc434400149)

[2.3.3. Controller Arguments 18](#_Toc434400150)

[2.3.4. Controller Return Value 20](#_Toc434400151)

[2.3.5. DAO (데이터 조회 시 0건) 21](#_Toc434400152)

[2.3.6. 다국어 처리 (Internationalization) 24](#_Toc434400153)

[2.3.7. 예외 처리 27](#_Toc434400154)

[2.3.8. 대용량 데이터 처리 27](#_Toc434400155)

[2.3.9. 엑셀 처리 (XENI) 30](#_Toc434400156)

[2.4. 예제 32](#_Toc434400157)

[2.4.1. nexacro platform License 설정 32](#_Toc434400158)

[2.4.2. VO class 구성 32](#_Toc434400159)

[2.4.3. Interceptor 35](#_Toc434400160)

[2.4.4. Controller 35](#_Toc434400161)

[2.4.5. Service 36](#_Toc434400162)

[2.4.6. DAO 37](#_Toc434400163)

[2.4.7. 대용량 데이터 39](#_Toc434400164)

[2.4.8. File handling 42](#_Toc434400165)

[2.4.9. 암호화 44](#_Toc434400166)

[2.4.10. Validation 44](#_Toc434400167)

[2.4.11. 예외처리 47](#_Toc434400168)

# 개요

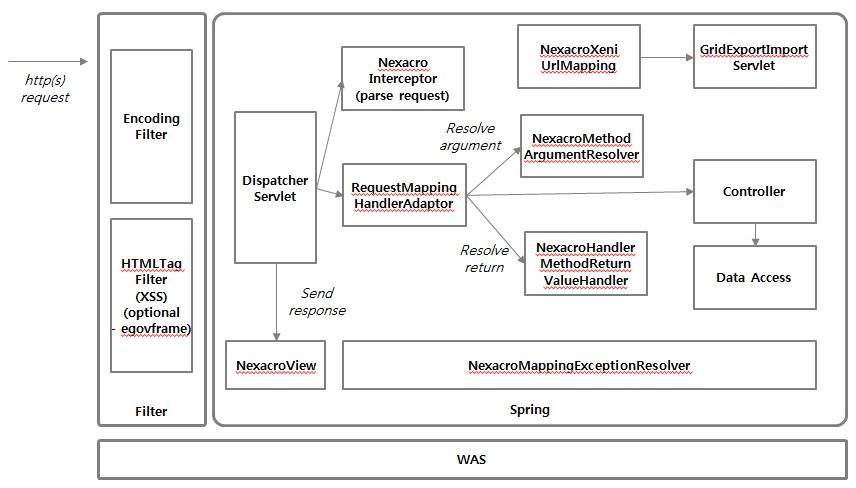
본 문서는 Spring과 nexacroplatform을 기반으로 하는 시스템 구축 시 개발자가 실제로 작업을 해야되는 내용에 대해 Spring의 각 Layer의 클래스를 작성하는 위주로 가이드를 제공함으로써 개발자가 일관성이 있는 개발을 하도록 지원하기 위해 작성되었다. nexacroplatform에 대한 지식은 어느 정도 갖추었거나 교육을 받은 것으로 한정하고 자세한 nexacroplatform 설명은 본 문서의 범주에서 제외했다.

개발 시 본 문서와 레퍼런스 사이트([www.nexacroplatform.co.kr](http://www.nexacroplatform.co.kr))를 통해 배포하는 소프트웨어를 활용하여 호환성의 문제가 발생하지 않도록 한다. 소프트웨어의 추가나 업그레이드가 있을 경우, 본 문서를 수정하여 추가 배포, 공지하도록 한다.

# Spring 기반 – nexacro platform

## 구성

Spring과 연동하기 위해 nexacro platform의 전반적인 architecture는 다음과 같다.



### NexacroInterceptor

nexacro platform으로부터 데이터를 수신받아 PlatformData로의 변환하는 적업을 수행한다.

### NexacroMethodArgumentResolver

개발자가 작성하게 되는 Controller의 argument의 데이터 변환을 수행한다.

Ex> PlatformData 🡪 POJO

### NexacroHandlerMethodReturnValueHandler

Controller에서 반환 되는 값의 데이터 변환을 수행한다.

Ex> POJO 🡪 PlatformData

### NexacroView

nexacro platform으로 데이터를 전달한다.

### NexacroMappingExceptionResolver

nexacro platform 통신 규약에 맞는 오류 처리를 담당한다.

## Data 변환

### PlatformData

nexacro platform에서 사용되는 데이터 형식이다.

데이터는 크게 단일 데이터(**Variable**)와 2차원 데이터(**DataSet**)로 구분한다. 단일 데이터는 데이터를 구분할 수 있는 식별자(name)와 값(value)을 가지고 있으며, 이는 VariableList에 저장된다. 2차원 데이터를 저장하는 DataSet은 열(column)과 행(row)으로 구성되어 있으며, DataSetList를 통해 저장 또는 참조된다.

VariableList와 DatasetList를 가지고 있는 PlatformData는 데이터 구조의 최상위에 위치하고 있으며, 데이터 이동과 데이터 송수신의 기본 단위로 사용되어 진다.

자세한 내용은 nexacro platform의 API 문서를 참조한다.

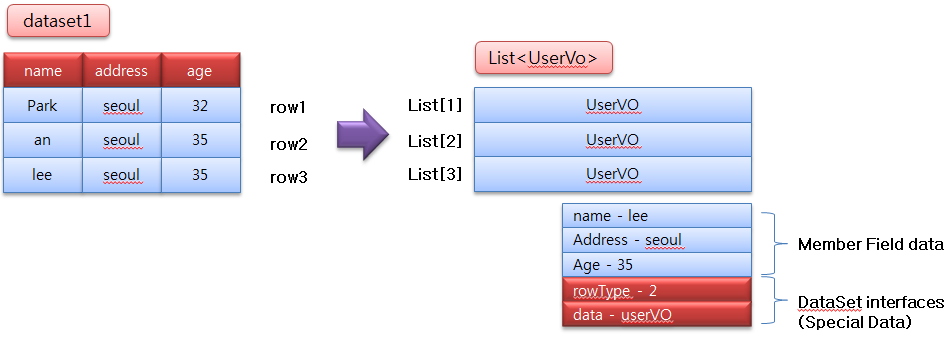
### Data 변환

nexacro platform에 종속성을 탈피하고자 Spring의 Controller level에서 데이터 변환을 수행하게 된다. (PlatformData 🡪 POJO)

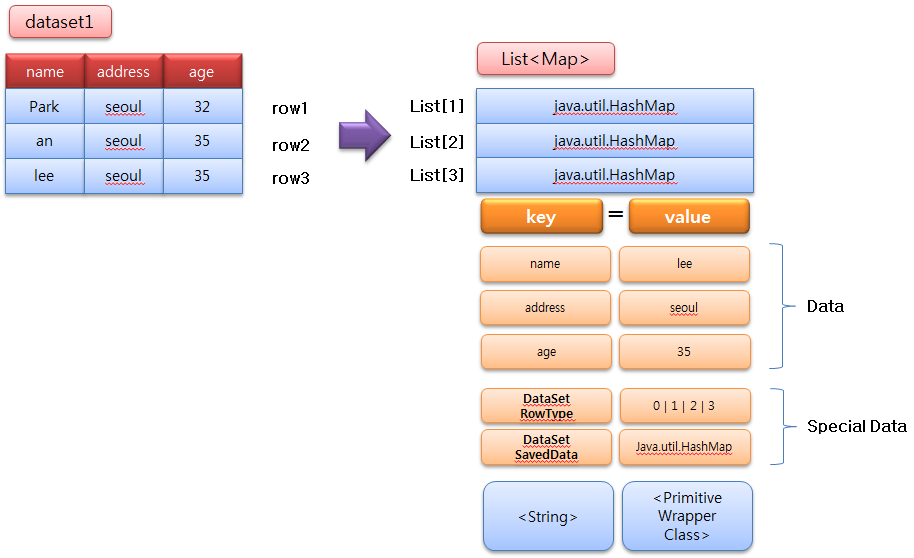
또한, Spring에서 DB Access하기 위해 지원하는 iBatis 모듈을 쉽게 사용하기 위해 고려되었다.

데이터 변환은 크게 2가지 형태로 데이터 변환을 수행한다. (역관계의 데이터 변환도 수행된다.)

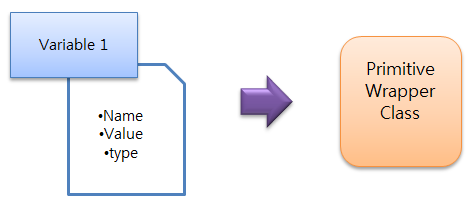
* DataSet (2차원 형태의 자료구조)
  + List<POJO>
    - **Variable과 동일한 데이터 타입만을 지원한다. 그 외 다른 타입의 경우 지원하지 않는다.**
  + List<Map>
    - **Variable과 동일한 데이터 타입만을 지원한다. 그 외 다른 타입의 경우 지원하지 않는다.**
* Variable
  + byte[], int, long, float, double, boolean
    - support wrapper class
  + java.lang.String, java.math.BigDecimal, java.util.Date, java.lang.Object
* DataSet 변환
  + List<VO>
    - DataSet을 List<VO> 형태로 변환 한다.



* + List<Map>
    - DataSet을 List<Map> 형태로 변환 한다.



* Variable 변환
  + Variable을 primitive Type 혹은 Wrapper class로 변환 한다.



* DataSet Special Data (DataSet의 가지고 있는 행에 대한 정보이다.)
  + RowType
    - DataSet에 존재하는 행의 상태를 말한다.
    - 해당 값은 다음과 같다.
      * com.nexacro.xapi.data.DataSet.ROW\_TYPE\_NORMAL – 일반 행을 의미한다.
      * com.nexacro.xapi.data.DataSet.ROW\_TYPE\_INSERTED – 추가된 행을 의미한다.
      * com.nexacro.xapi.data.DataSet.ROW\_TYPE\_UPDATED – 값이 변경된 행을 의미한다.
      * com.nexacro.xapi.data.DataSet.ROW\_TYPE\_DELETED – 삭제 행을 의미한다.
    - VO 일 경우 **com.nexacro.spring.data.DataSetRowTypeAccessor** interface를 구현한다.
      * DataSetRowTypeAccessor는 다음과 같이 정의되어 있다.

**public** **interface** **DataSetRowTypeAccessor** {

**public** **void** setRowType(**int** rowType);

**public** **int** getRowType();

}

* + - Map 일 경우 int 형 데이터가 설정된다.
      * Key : com.nexacro.spring.data.DataSetRowTypeAccessor.NAME 식별자를 사용한다.
      * Value : com.nexacro.xapi.data.DataSet.ROW\_TYPE\_NORMAL | ROW\_TYPE\_INSERTED | ROW\_TYPE\_UPDATED | ROW\_TYPE\_DELETED를 값으로 사용한다.
      * Ex> int rowType =

mapData.get(com.nexacro.spring.data.DataSetRowTypeAccessor.NAME);

* + Saved Data
    - DataSet의 값이 변경되기 이전의 값에 대한 정보이다.
    - VO 일 경우 com.nexacro.spring.data.DataSetSavedDataAccessor<T>를 구현한다.
      * Saved Data를 획득하기 위해선 VO class와 동일한 class로 선언해야 한다.
      * DataSetSavedDataAccessor는 다음과 같이 정의되어 있다.

**public** **interface** **DataSetSavedDataAccessor**<T> {

**public** **void** setOrgData(T t);

**public** T getOrgData();

}

* + - Map 일 경우 Map 내부에 원본데이터가 설정 된다.
      * Key : com.nexacro.spring.data.DataSetSavedDataAccessor.NAME 식별자를 사용한다.
      * Value : Map 형태의 데이터가 설정 된다.
      * Ex> Map savedDataMap =

mapData.get(com.nexacro.spring.data.DataSetSavedDataAccessor.NAME);

* DataSet의 RowType은 nexacro platform에서 transaction 시 입력 데이터셋의 전송옵션이 :U 혹은 :A 일때 행의 타입을 확인할 수 있다. :N 옵션일 경우에는 Normal 상태이다. (default)
* DataSet의 SavedData (원본데이터)의 경우 :A 옵션으로 데이터를 전송할 경우에만 저장된다.
* 자세한 내용은 nexacro studio의 help 창의 transaction 옵션을 참고한다.

### Data Type Mapping

nexacro platform과 java간의 데이터 타입에 대한 Mapping 정보는 다음과 같다.

|  |  |
| --- | --- |
| Java types | nexacro platform types |
| byte[], java.lang.Byte[] | PlatformDataType.BLOB |
| int, java.lang.Integer | PlatformDataType.INT |
| long, java.lang.Long | PlatformDataType.LONG |
| float, java.lang.Float | PlatformDataType.FLOAT |
| double, java.lang.Double | PlatformDataType.DOUBLE |
| boolean, java.lang.Boolean | PlatformDataType.BOOLEAN |
| java.lang.String | PlatformDataType.STRING |
| Java.math.BigDecimal | PlatformDataType.BIG\_DECIMAL |
| java.util.Date | PlatformDataType.DATE |
| PlatformDataType.DATE\_TIME |
| PlatformDataType.TIME |
| java.lang.Object | PlatformDataType.UNDEFINED |

그외 추가적인 타입에 대해선 데이터 손실의 우려가 있기 때문에 지원하지 않는다.

## 기능 명세

### 설정

Spring의 dispatcher-servlet.xml 파일내에 다음과 같은 bean을 등록한다.

<bean id=*"nexacroInterceptor"* class=*"com.nexacro.spring.servlet.NexacroInterceptor"*/>

<!-- nexacro method argument resolver -->

<bean id=*"nexacroMethodArgumentResolver"*

class=*"com.nexacro.spring.resolve.NexacroMethodArgumentResolver"* />

<bean id=*"nexacroView"* class=*"com.nexacro.spring.view.NexacroView"* >

<property name=*"defaultContentType"* value=*"PlatformXml"* />

<property name=*"defaultCharset"* value=*"UTF-8"* />

</bean>

<bean id=*"nexacroFileView"* class=*"com.nexacro.spring.view.NexacroFileView"* />

<!-- nexacro method return value handler -->

<bean id=*"nexacroMethodReturnValueHandler"*   
class=*"com.nexacro.spring.resolve.NexacroHandlerMethodReturnValueHandler"* >

<!-- <property name="view" value="nexacroView" /> -->

<property name=*"view"* ref=*"nexacroView"* />

<property name=*"fileView"* ref=*"nexacroFileView"* />

</bean>

<!-- nexacro exception resolver -->

<bean id=*"exceptionResolver"* class=*"com.nexacro.spring.resolve.NexacroMappingExceptionResolver"* p:order=*"1"*>

<property name=*"view"* ref=*"nexacroView"* />

<property name=*"shouldLogStackTrace"* value=*"true"* />

<property name=*"shouldSendStackTrace"* value=*"true"* />

<!-- shouldSendStackTrace 가 false 일 경우 nexacro platform으로 전송되는 에러메시지 -->

<property name=*"defaultErrorMsg"* value=*"fail.common.msg"* />

<property name=*"messageSource"* value=*"messageSource"* />

</bean>

<!-- XENI (nexacro) Request Mapping -->

<bean class=*"org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"*/>

<bean id=*"xeniUrlMapping"* class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"* p:order=*"0"*>

<!--

<property name="interceptors">

<list>

</list>

</property>

-->

<property name=*"mappings"*>

<props>

<prop key=*"/XExportImport.do"*>xeniWrappingController</prop>

</props>

</property>

</bean>

<bean id=*"xeniWrappingController"*

class=*"org.springframework.web.servlet.mvc.ServletWrappingController"*>

<property name=*"servletClass"*>

<value>com.nexacro.xeni.services.GridExportImportServlet</value>

</property>

</bean>

* nexacroInterceptor
  + DataSet 🡪 List<?> 데이터 변환 시 Spring의 보안 적용을 위한 interceptor class 이다.
    - nexacro platfrom의 DataSet, Variable 자체의 데이터를 Spring의 보안 항목을 적용 시킨다.
* nexacroMethodArgumentResolver
  + Controller에서 사용되는 입력 파라매터에 대한 데이터 변환 작업을 수행한다
  + convertListener
    - nexacroSecurityListener를 등록하여 사용 가능하다.
* nexacroView
  + nexacro platform으로 데이터 송신 역할을 수행한다.
* nexacroFileView
  + nexacro platform으로 File 데이터에 대한 송신 역할을 수행한다.
* nexacroMethodReturnValueHandler
  + Controller에서 반환되는 출력 파라매터에 대한 데이터 변환 작업을 수행한다.
  + properties
    - view
      * 기본적으로 사용되는 View
    - fileView
      * File 처리 시 사용되는 View
* exceptionResolver
  + nexacro platform에서 서비스 요청 시 예외에 대한 처리를 수행한다.
  + nexacro platform에 대한 요청을 선처리 하기 위해 order를 최우선으로 적용한다.
  + properties
    - view
      * 예외 발생시 사용되는 View
    - shouldLogStackTrace
      * 에러정보를 로깅할지 여부 (true | false)
    - shouldSendStackTrace (예외의 실제 메시지를 응답으로 전송할 경우 보안상의 문제가 있을 수 있다.)
      * 에러정보를 응답으로 전송할지 여부 (true | false)
      * 보안상의 이유로 운영시에는 false로 설정한다.
    - defaultErrorMsg
      * shouldSendStackTrace 가 false 일 경우 nexacro platform으로 전송되는 에러메시지
      * 정의된 메시지 명칭 혹은 값을 입력한다.
    - messageSource
      * 메시지를 처리하기 위한 org.springframework.context.MessageSource를 설정한다.
* simpleControllerHandlerAdaptor
  + nexacro platform의 XENI를 사용하기 위한 Controller Handler이다.
* xeniUrlMapping
  + XENI에 대한 Export, Import 요청을 처리하기 위한 URL Mapping이다.
* xeniWrappingController
  + XENI를 Spring 내에서 처리하기 위한 ServletWrappingController 이다.

### Spring 적용

* 데이터 변환
  + 데이터 변환을 위해 customArgumentResolvers, customReturnValueHandlers를 등록한다.

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"*>

<property name=*"customArgumentResolvers"*>

<list>

<!-- regist Nexacro Argument Resolvers.. -->

<ref bean=*"nexacroMethodArgumentResolver"*/>

</list>

</property>

<property name=*"customReturnValueHandlers"*>

<list>

<!-- regist Nexacro Return Value Handler.. -->

<ref bean=*"nexacroMethodReturnValueHandler"*/>

</list>

</property>

...

</bean>

### Controller Arguments

Controller에서 입력 파라매터로 다음과 같은 데이터 형식을 사용할 수 있다.

|  |  |  |
| --- | --- | --- |
| Class 명 | 설명 | 비고 |
| List <?> list | DataSet 을 List로 converting한 Data  (반드시 Generic을 사용해야 한다. Map혹은 VO class만 지정가능하다.) | Converting된 Data  @ParamDataSet함께 사용 |
| Primitive type | Variable을 Object로 converting한 Data | Converting된 Data  @ParamVariable함께 사용 |
| PlatformData | nexacro에서 사용하는 모든 Data | nexacro UI에서 사용되는 Data |
| DataSetList | Dataset의 list | nexacro UI에서 사용되는 Data |
| VariableList | Variable 의 List | nexacro UI에서 사용되는 Data |
| Dataset | 2차원의 Data | nexacro UI에서 사용되는 Data  @ParamDataSet 함께 사용 |
| Variable | 1차원의 Data | nexacro UI에서 사용되는 Data  @ParamaVariable 함께 사용 |
| HttpPlatformRequest | Http요청으로 Platform를 가진다. |  |
| HttpPlatformResponse | Http 응답으로 Platform 을 가진다. |  |
| NexacroFirstRowHandler | Http응답으로 Dataset을 분할 전송한다. |  |
|  |  |  |

Converting 위한 추가 정보는 annotation을 사용하여 처리한다.

Dataset혹 Variable이름 : name 사용

* Variable 표현 방식

예제)

|  |
| --- |
| @ParamVariable(name="var6" required=false) String var6  @ParamVariable(name="var6") int var6  @ParamVariable(name=”var6”) Variable var6 |

주의) Type은 아래만 사용가능하다.

**int, byte[], long, float, double, boolean, String, Object, Date, BigDecimal**

* Dataset 표현 방식

예제)

|  |
| --- |
| @ParamDataset(name="dsHeader" required=false) List <Map> dsList1  @ParamDataset(name="dsHeader") List <Input> dsList1  @ParamDataset(name=”dsHeader") DataSet dsHeader1 |

* NexacroFirstRowHandler
  + 대용량 처리를 위해 제공되는 class
  + Http응답으로 Dataset을 분할 전송한다.

그 외 nexacro platform의 데이터 형식은 nexacro-xapi API 문서를 참조한다.

### Controller Return Value

Controller에서 출력 파라매터로 사용할 수 있는 형식은 다음과 같다.

* NexacroResult
  + nexacro platform으로 데이터 송신을 하기 위한 정보를 가진다.
  + 대표적인 메서드는 다음과 같다.
    - void addDataSet(String name, List<?> list)
      * nexacro platform으로 전송할 출력 DataSet(List<?>)을 추가한다.
      * Controller의 입력 파라매터 정보와 같이 반드시 List<VO> 혹은 List<Map> 형태의 데이터만 설정가능하다.
    - void addVariable(String name, Object value)
      * nexacro platform으로 전송할 출력 Variable(Object)를 추가한다.
      * 2.2 Data 변환의 Data Type Mapping 정보를 참조한다.
    - void setErrorCode(int errorCode)
      * 에러코드를 설정한다.
    - void setErrorMsg(String errorMsg)
      * 에러 메시지를 설정한다.
* NexacroFileResult
  + nexacro platform으로 File 데이터를 송신하기 위한 정보를 가진다.
  + 대표적인 메서드는 다음과 같다.
    - void setContentType(String contentType) – 파일 데이터의 MimeType을 설정한다.
    - void setFile(File file) – 전송하기 위한 파일 데이터를 설정한다.
    - void setOriginalName(String originalName) – 전송 시 사용되는 파일의 명칭을 설정한다.
* PlatformData
  + nexacro platform의 PlatformData 형식

### DAO (데이터 조회 시 0건)

nexacro platform의 경우 서버에서 전송 된 DataSet의 스키마 정보가 존재 할 경우 손쉽게 UI를 구성할 수 있다. 하지만 DataSet의 스키마 정보가 전송되지 않을 경우 UI 개발자는 별도로 스키마 정보를 정의해서 사용해야 하는 불편함이 있다.

iBatis | mybatis를 사용하여 데이터를 조회하였을 때 조회된 데이터가 없을 경우 (Map 형태의 데이터 사용시) 스키마에 대한 정보를 확인 할 수 없어 DataSet으로 변환 시 스키마 정보가 생성되지 않는다.

이를 처리하기 위해 iBatis | mybatis를 통해 조회된 데이터가 0건일 경우 재조회를 통해 ResultSet의 MetaData 정보를 이용하여 스키마 정보를 획득한다.

아래는 스키마 정보 획득 시 처리되는 형태에 대한 정보이다.

* VO 사용 시
  + 추가적인 데이터 조회는 이루어지지 않으며 VO class 정보만을 획득하여 스키마를 구성한다.
* java.util.Map 사용시 - (java.util.Map)을 사용하지 않을 경우 구현 필요 없음)
  + ResultSet의 MetaData와 ResultMap에 명시된 정보를 바탕으로 스키마 정보를 획득한다.
  + ResultSet으로부터 MetaData를 획득 할 경우 DBMS에 따라 상이한 정보로 인하여 추가적인 처리가 필요하다.
    - Dbms 구현 시 com.nexacro.spring.AbstractDbms를 상속받아 구현한다.
      * + 구현 대상 메서드는 다음과 같다.

void handleColumn(DbColumn column)

* + - * + 아래는 Mysql 사용시 dbms 처리에 대한 sample이다.

**public** **class** Mysql **extends** AbstractDbms {

@Override

**public** **void** handleColumnDataType(DbColumn column) {

**if** (column == **null**) {

**return**;

}

**if** ("MEDIUMINT".equals(column.getVendorsTypeName())) {

column.setDataType(PlatformDataType.*INT*);

} **else** **if** ("DATETIME".equals(column.getVendorsTypeName())) {

column.setDataType(PlatformDataType.*DATE\_TIME*);

} **else** **if** ("YEAR".equals(column.getVendorsTypeName())) {

column.setDataType(PlatformDataType.*DATE*);

}

}

}

* 스프링 설정

생성된 Mysql 은 context-nexacro.xml 파일에 등록된다. 기본적으로 등록된 내용은 아래와 같으며, 추가적인 데이터타입 처리가 필요한 경우 생성하여 bean으로 등록한다.

<bean id=*"hsqlDbms"* class=*"com.nexacro.spring.dao.dbms.Hsql"* />

<bean id=*"oracleDbms"* class=*"com.nexacro.spring.dao.dbms.Oracle"* />

<bean id=*"mssqlDbms"* class=*"com.nexacro.spring.dao.dbms.Mssql"* />

<bean id=*"mysqlDbms"* class=*"com.nexacro.spring.dao.dbms.Mysql"* />

<bean id=*"tiberoDbms"* class=*"com.nexacro.spring.dao.dbms.Tibero"* />

<bean id=*"dbmsProvider"* class=*"com.nexacro.spring.dao.DbVendorsProvider"*>

<property name=*"dbverdors"*>

<map>

<entry key=*"HSQL Database Engine"* value-ref=*"hsqlDbms"*/>

<entry key=*"SQL Server"* value-ref=*"mssqlDbms"*/>

<entry key=*"Oracle"* value-ref=*"oracleDbms"*/>

</map>

</property>

</bean>

※ Dbverdors 항목의 entry key 값은 connection의 getProductName()과 동일하다(mybatis DatabaseIdProvider의 key 값과 동일하다.)

#### ibatis

NexacroIbatisAbstractDAO를 상속받아 DAO 클래스를 구현한다.

아래는 조회 된 데이터가 없을 경우 메타데이터 정보를 조회하는 대상 메서드들이다.

* List<?> list(String queryId)
* List<?> list(String queryId, Object parameterObject)
* List<?> list(String queryId, int skipResults, int maxResults)
* List<?> list(String queryId, Object parameterObject, int skipResults, int maxResults)

#### Mybatis

Mybatis는 메타 데이터 정보를 조회하기 위해 설정파일에 Plugin 형태로 등록하여 처리 한다.

아래는 sql-mapper-config.xml (mybatis 설정파일)에 설정되는 Plugin 항목들이다.

|  |
| --- |
| <configuration>  <!-- myBatis Inteceptor for get column information -->  <plugins>  <plugin interceptor=*"com.nexacro.spring.dao.mybatis.NexacroMybatisMetaDataProvider"* />  <plugin interceptor=*"com.nexacro.spring.dao.mybatis.NexacroMybatisResultSetHandler"* />  </plugins>  ...  </configuration> |

* *com.nexacro.spring.dao.mybatis.NexacroMybatisMetaDataProvider*
  + Executor Plugin – 쿼리를 실행하며, List 형태의 데이터 조회 시 조회된 결과가 0건일 경우 결과 메타데이터 정보를 조회 한다. resultType이 Map 일 경우 추가적으로 메타데이터 조회를 위한 쿼리를 실행한다.
* *com.nexacro.spring.dao.mybatis.NexacroMybatisResultSetHandler*
  + ResultSetHandler Plugin – Executor Plugin에서 메타데이터 조회를 위해 쿼리를 실행하였을 경우에만 동작하며, ResultSetMetaData 와 resultType에 존재하는 필드 정보를 확인하여 메타데이터 정보를 조회 한다.

### 다국어 처리 (Internationalization)

Spring 내에서 다국어 처리를 하기 위한 방법으로 Spring MVC에서 제공하고 있는 LocaleResolver를 이용하여 처리 한다.

LocaleResolver의 종류는 다음과 같다.

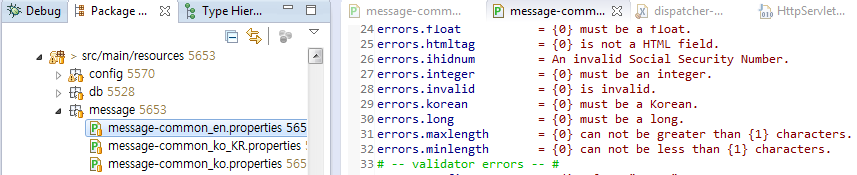
* AcceptHeaderLocaleResolver
  + HTTP Request의 accept-language Header값을 통해 Locale 정보를 처리한다.
* CookieLocaleResolver
  + Cookie에 설정 된 Locale 정보로 처리한다.
* SessionLocaleResolver
  + Session에 설정 된 Locale 정보로 처리한다.
* FiexdLocaleResolver
  + 항상 고정된 Locale 정보를 이용하며, 현재 JVM의 Locale 값이 기본으로 사용된다.

#### 다국어 처리 방법

* 인코딩 설정
  + web.xml에 다음과 같이 Filter를 등록한다.

|  |
| --- |
| <filter>  <filter-name>encodingFilter</filter-name>  <filter-class>  org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>encodingFilter</filter-name>  <url-pattern>\*.do</url-pattern>  </filter-mapping> |

* 메시지(properties) 파일 작성
  + 메시지 파일의 작성규칙은 다음과 같다.
    - [\*]\_[language]\_[country].properties 이다.
    - language – 언어 코드
      * Language Code는 ISO-639에 정의 된 값이다.
    - country – 지역 코드
      * Country Code는 ISO-3166에 정의 된 값이다.
  + 아래와 같은 형태로 메시지 파일을 작성한다.



* MessageSource 정의
  + context-common.xml에 다음과 같이 messageSource 빈을 정의한다.

|  |
| --- |
| <bean id=*"messageSource"*  class=*"org.springframework.context.support.ReloadableResourceBundleMessageSource"*>  <property name=*"basenames"*>  <list>  <!-- 메시지 properties 파일을 등록한다. 등록 하지 않은 경우의 기본값은 message.properties 이다.-->  <value>classpath:/message/message-common</value>  <value>classpath:/message/message-user</value>  </list>  </property>  <property name=*"cacheSeconds"*>  <value>60</value>  </property>  </bean> |

* LocaleResolver 정의
  + Locale 처리를 위해 dispatcher-servlet.xml에 SessionLocaleResolver를 다음과 같이 등록한다.

|  |
| --- |
| ...  <bean id=*"localeResolver"*  class=*"org.springframework.web.servlet.i18n.SessionLocaleResolver"* /> |

* 처리 방법
  + 메시지 처리를 직접하기 위해선 다음과 같이 작성할 수 있다.

|  |
| --- |
| **import** org.springframework.context.MessageSource;  ...  String reason = **this**.messageSource.getMessage(  reason, **null**,reason, LocaleContextHolder.*getLocale*());} |

#### 기본 다국어 처리 대상

기본적으로 메시지 처리되는 대상은 다음과 같다.

* Validator Message
  + Validation 시 메시지 처리를 하기 위해선 validator에 다음과 같이 MessageSource를 등록한다.

|  |
| --- |
| <bean id=*"validator"*  class=*"org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"*>  <property name=*"validationMessageSource"* ref=*"messageSource"* />  </bean> |

* + Validation 시 다음과 같이 정의 된 Message의 식별자를 통해 다국어 처리를 한다.

|  |
| --- |
| @Pattern(regexp = ".+@.+\\.[a-z]+", **message = "{errors.email}"**)  **private** String email; |

* Exception Message
  + 예외 처리 시 메시지 처리를 하기 위해선 다음과 같이 NexacroMappingExceptionResolver에 MessageSource를 등록한다.

|  |
| --- |
| <bean id=*"exceptionResolver"* class=*"com.nexacro.spring.resolve.NexacroMappingExceptionResolver"* p:order=*"1"*>  ...  <property name=*"defaultErrorMsg"* value=*"fail.common.msg"* />  <property name=*"messageSource"* ref=*"messageSource"* />  </bean> |

* + 명시적 예외 처리 시 다음과 같이 정의 된 Message의 식별자를 통해 다국어 처리를 한다.

|  |
| --- |
| **...**  **throw** **new** NexacroException("log message...", -8859, **"fail.common.msg"**); |

### 예외 처리

Spring 내에서 nexacro platform으로 예외를 처리하기 위해서 NexacroException을 이용하여 처리한다.

com.nexacro.spring.NexacroException의 주요 메서드는 다음과 같다.

* + public NexacroException(String message) – 입력받은 message를 통해 NexacroException을 생성한다.
  + public NexacroException(String message, Throwable cause) – 입력받은 message와 cause를 통해 NexacroException을 생성한다.
  + public NexacroException(String message, int errorCode, String errorMsg) – 입력받은 message, 에러코드, 에러메시지를 통해 NexacroException을 생성한다.
  + public NexacroException(String message, Throwable cause, int errorCode, String errorMsg) – 입력받은 message와 cause, 에러코드, 에러메시지를 통해 NexacroException을 생성한다.
  + void setErrorCode(int errorCode) – nexacro platform으로 전송 할 에러코드를 설정한다.
    - 설정하지 않을 경우 ‘**-1’** 을 전송한다.
  + void setErrorMsg(String errorMsg) – nexacro platform으로 전송 할 에러메세지를 설정한다.
    - 설정하지 않을 경우 NexacroException 생성 시 전달 받은 메시지를 전송한다.
* NexacroMappingExceptionResolver 의 **shouldSendStackTrace** property 설정 값에 따라 응답으로 전송되는 에러 정보가 달라진다.
  + shouldSendStackTrace 가 false로 설정되어 있을 경우 예외의 message는 전송되지 않으며, errorMsg(에러메시지) 만 응답으로 전송된다.
    - 에러에 대한 정보는 서버의 로그를 통해 확인한다.
* 추가정보
  + nexacro platform에서의 기본적인 에러처리 형식은 응답헤더 코드는 200(OK)으로 전송되며, 내부에 Variable ( ErrorCode, ErrorMsg )을 통해 처리 한다.
    - transaction 메서드의 콜백 함수에 처리 된다.
  + 그 외 응답코드(4xx, 5xx 등) 에 대한 부분은 ADL 파일의 onerror 이벤트에서 처리 한다.

### 대용량 데이터 처리

#### 대용량 데이터 입력

대용량의 데이터가 입력으로 사용될 경우 보안 적용으로 인해 성능 및 메모리 이슈가 발생할 수 있다.

아래는 대량의 데이터가 입력으로 사용될 경우 성능에 대한 부분이다.

(정확한 데이터는 아니며 참고용으로만 사용한다.)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **입력건수** | **parse request** | **check security** | **convert** | **Total (ms)** |
| 5,000 | 0.024 | 0.272 | 0.012 | 0.308 |
| 10,000 | 0.036 | 0.496 | 0.028 | 0.56 |
| 50,000 | 0.173 | 2.544 | 0.122 | 2.839 |
| 100,000 | 0.362 | 5.071 | 0.246 | 5.679 |

아래는 측정에 사용된 테스트 데이터이다.



위와 같이 성능적인 측면에서 대량의 데이터는 성능 및 메모리 이슈가 발생할 수 있기 때문에 online상에서는 지양해야 하며, 필요 시 서버사이드에서 spring batch 혹은 상용 batch 프로그램을 통해 처리해야 한다.

#### 대용량 데이터 분할 전송

대용량 데이터를 전송하기위해 데이터를 분할하여 전송하는 방식을 지원한다.

Controller에서 입력 파라매터로 NexacroFirstRowHandler를 선언하여 대용량 데이터를 전송하도록 한다.

com.nexacro.spring.data.NexacroFirstRowHandler의 주요내용은 다음과 같다.

* + 전송 규칙은 Variable이 전송 되고 난 뒤 DataSet을 전송한다.
    - DataSet이 전송되고 난 후 Variable이 전송 될 경우 예외를 발생시킨다.
  + 대용량 처리시 UI에서 에러처리를 하기 위해선 기존의 ErrorCode, ErrorMsg 외에 추가적으로 데이터셋을 통해 에러를 처리해야 한다.
    - DataSet Schema
      * DataSet 이름 : FirstRowStatus
      * Column : ErrorCode, ErrorMsg
  + 대표적인 메서드는 다음과 같다.
    - void sendPlatformData(PlatformData platformData) – 입력 받은 PlatformData를 송신한다.
    - void sendDataSet(DataSet ds) – 입력 받은 DataSet을 송신한다.
    - void sendVariable(Variable var) – 입력 받은 Variable을 송신한다.
    - void setContentType(String contentType) – 데이터 송신 시 사용되는 ContentType을 설정한다.
      * PlatformType.CONTENT\_TYPE\_XML
      * PlatformType.CONTENT\_TYPE\_SSV
      * PlatformType.CONTENT\_TYPE\_BINARY

### 엑셀 처리 (XENI)

nexacro platform에서 엑셀을 처리하기 위해선 nexacro-xeni (XENI)를 이용하여 처리 한다.

XENI는 Spring의 ServletWrappingController를 이용하여 처리되었다.

#### 설정

XENI를 사용하기 위한 설정은 다음과 같다.

* web.xml (XENI 환경 설정)
* dispatcher-servlet.xml (XENI Servlet 등록)
* xeni.properties 파일 (Spring의 MultipartRequest를 처리하기 위한 설정)

아래는 XENI의 web.xml 설정이다. (각 항목에 대한 상세한 설명은 XENI 문서를 참고한다.)

|  |
| --- |
| <!-- xeni context param -->  <context-param>  <param-name>export-path</param-name>  <param-value>file://C:/DES/workspace/excel</param-value>  </context-param>  <context-param>  <param-name>import-path</param-name>  <param-value>file://C:/DES/workspace/excel</param-value>  </context-param>  <context-param>  <param-name>monitor-enabled</param-name>  <param-value>true</param-value>  </context-param>  <context-param>  <param-name>monitor-cycle-time</param-name>  <param-value>30</param-value>  </context-param>  <context-param>  <param-name>file-storage-time</param-name>  <param-value>10</param-value>  </context-param> |

아래는 dispatcher-servlet.xml의 설정이다.

|  |
| --- |
| <!-- XENI (nexacro) Request Mapping -->  <bean class=*"org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"*/>  <bean id=*"xeniUrlMapping"* class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"* p:order=*"0"*>  <property name=*"mappings"*>  <props>  <prop key=*"/XExportImport.do"*>xeniWrappingController</prop>  </props>  </property>  </bean>    <bean id=*"xeniWrappingController"*  class=*"org.springframework.web.servlet.mvc.ServletWrappingController"*>  <property name=*"servletClass"*>  <value>com.nexacro.xeni.services.GridExportImportServlet</value>  </property>  </bean> |

아래는 xeni.properties 파일의 내용이다. (src/main/resources 폴더에 xeni.properties 파일을 생성한다.)

|  |
| --- |
| xeni.multipart.proc=com.nexacro.spring.servlet.XeniMultipartHandler |

## 예제

### nexacro platform License 설정

nexacro platform과 통신을 위해 사용되는 nexacro-xapi-1.x.jar의 경우 ‘nexacro14\_server\_license.xml’ License 파일이 필요하다. 혹은 바이너리 형식의 파일로 구성된다.

License 파일의 설치는 현재 개발 중인 **src/main/resources** 폴더 하단에 위치시킨다.

License 파일의 위치는 다음과 같은 순서로 확인 된다.

* jar 파일과 동일한 디렉토리
* 클래스 경로 (일반적으로 Web Application의 /WEB-INF/classes)

### VO class 구성

nexacro platform 에서 VO class로 데이터 변환을 하기 위해서는 다음과 같이 구성하도록 한다.

* VO class 구성
  + VO class는 xxx.vo package 하단에 생성한다.
  + Data Type Mapping 부분을 참고하여 Data 변환이 가능한 Member Field 만 데이터가 할당 된다.
  + 반드시 해당 Member Field의 이름으로 해당 하는 Getter, Setter가 존재해야 한다.
    - private String name;
      * public String getName()
      * public void setName()
  + DataSet의 행의 타입(rowType)과 원본데이터를 획득하고자 한다면 아래 interface를 구현하여야 한다.
    - com.nexacro.spring.data.DataSetRowTypeAccessor
    - com.nexacro.spring.data.DataSetSavedDataAccessor
* DataSet의 RowType은 nexacro platform에서 transaction 시 입력 데이터셋의 전송옵션이 :U 혹은 :A 일때 행의 타입을 확인할 수 있다. :N 옵션일 경우에는 Normal 상태이다. (default)
* DataSet의 SavedData (원본데이터)의 경우 :A 옵션으로 데이터를 전송할 경우에만 저장된다.
* 자세한 내용은 nexacro studio의 help 창의 transaction 옵션을 참고한다.
* 예제
  + VO class 구성에 대한 예제이다.

|  |
| --- |
| **public** **class** UnitVO **implements** DataSetRowTypeAccessor  , DataSetSavedDataAccessor<UnitVO>{  **private** String id;  **private** String name;  **private** **int** age;  **private** **float** height;  **private** **int** width;  **private** **int** rowType;  **private** UnitVO savedData;    **public** String getId() {  **return** id;  }  **public** **void** setId(String id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** **float** getHeight() {  **return** height;  }  **public** **void** setHeight(**float** height) {  **this**.height = height;  }  **public** **int** getWidth() {  **return** width;  }  **public** **void** setWidth(**int** width) {  **this**.width = width;  }  @Override  **public** UnitVO getData() {  **return** **this**.savedData;  }  @Override  **public** **void** setData(UnitVO data) {  **this**.savedData = data;  }  @Override  **public** **int** getRowType() {  **return** **this**.rowType;  }  @Override  **public** **void** setRowType(**int** rowType) {  **this**.rowType = rowType;  }  } |

### Interceptor

서비스 개발 시 세션 체크 및 권한 체크를 위해 Spring의 Interceptor를 구현하는 경우가 있다.

interceptor에서 nexacro platform의 데이터를 사용하기 위해선 다음과 같이 사용할 수 있다.

* + Interceptor는 반드시 NexacroInterceptor가 우선 동작 해야 한다.

|  |
| --- |
| **public** **boolean** preHandle(HttpServletRequest request  , HttpServletResponse response, Object object) **throws** Exception {  **NexacroContext context = NexacroContextHolder.*getNexacroContext*();**  **PlatformData platformData = context.getPlatformData();**    DataSet dsUserInfo = platformData.getDataSet("gdsUser");  // ...    **return** **true**;  } |

### Controller

nexacro platform과 Spring 연동 시 Controller에서는 다음과 같이 사용한다.

* Controller 구성
  + xxx.web package 하단에 생성한다.
  + @Controller 어노테이션을 통해 선언한다.

* Controller 입출력
  + 입력
    - Controller에서 사용되는 형식은 @ParamDataSet 혹은 @ParamVariable을 이용하여 nexacro platform에서 전달 받은 DataSet 혹은 Variable의 명칭을 통해 획득한다.
  + 출력
    - 반환되는 값은 NexacroResult를 통해 데이터를 전달하도록 한다.
* 예제
  + Controller에서 처리되는 형태에 대한 예제 이다.

|  |
| --- |
| @RequestMapping(value = "/test.do")  **public** NexacroResult test(  @ParamDataSet(name="dsUnit" required=false) List<UnitVO> unitList  , @ParamDataSet(name="dsUnit") List<Map> unitMapList  , @ParamDataSet(name="dsUnit") DataSet dsUnit  , @ParamVariable(name="intValue" ) **int** iVar1  , @ParamVariable(name="intValue") Variable iVar2  , @ParamVariable(name="stringValue") String sVar1  , @ParamVariable(name="stringValue") Variable sVar2    , DataSetList dsList  , VariableList varList  , PlatformData platformData  , HttpPlatformRequest platformRequest  , HttpPlatformResponse platformResponse  , NexacroFirstRowHandler firstRowHandler  ){    **if** (log.isDebugEnabled()) {  log.debug("debug\_message : nexacro supported parameter types..");  }  // some logic…  List<UnitVO> resultList = service.selectVo(unitList);  // control nexacro result...  NexacroResult result = **new** NexacroResult();  result.addDataSet("dsUnitList", resultList);  result.addVariable("responseInt", iVar1);  result.addVariable("responseString", sVar1);      **return** result;  } |

Controller의 입력 파라매터와 출력에 대한 상세한 내용은 기능명세 부분을 참고한다.

### Service

nexacro platform과 Spring 연동 시 Service에서는 다음과 같이 사용한다.

* Service 구성
  + xxx.service package 하단에 생성한다.
  + Service class는 interface를 생성하여 구현한다.
  + EgovFramework를 사용하는 경우 Service Implementation class는 EgovAbstractServiceImpl 를 구현한다.
* 예제
  + 본 예제는 데이터셋의 행의 타입(RowType)을 이용하여 CRUD 처리를 하는 부분의 예제이다.

|  |
| --- |
| @Autowired  **private** SampleDAO sampleDAO;  @Override  **public** **void** modifyMultiSampleVO(List<SampleVO> modifyList) {  **int** size = modifyList.size();  **for** (**int** i=0; i<size; i++) {  SampleVO sample = modifyList.get(i);  **if** (sample **instanceof** DataSetRowTypeAccessor){  DataSetRowTypeAccessor accessor  = (DataSetRowTypeAccessor) sample;  // row type  **if** (accessor.getRowType() == DataSet.*ROW\_TYPE\_INSERTED*){  insertSampleVO(sample);  }**else** **if** (accessor.getRowType() == DataSet.*ROW\_TYPE\_UPDATED*){  updateSampleVO(sample);  }**else** **if** (accessor.getRowType() == DataSet.*ROW\_TYPE\_DELETED*){  deleteSampleVO(sample);  }  }    }  }    @Override  **public** List<SampleVO> selectSampleVOList(SampleVO searchVO) {  **return** sampleDAO.selectSampleVoList(searchVO);  }  **public** **void** insertSampleVO(SampleVO sample) {  sampleDAO.insertSampleVO(sample);  }    **public** **void** updateSampleVO(SampleVO sample) {  sampleDAO.updateSampleVO(sample);  }    **public** **void** deleteSampleVO(SampleVO sample) {  sampleDAO.deleteSampleVO(sample);  } |

### DAO

#### iBatis

iBatis를 사용하여 데이터를 조회하였을 때 조회된 데이터가 없을 경우 (Map 형태의 데이터 사용시) 스키마에 대한 정보를 확인 할 수 없어 DataSet으로 변환 시 스키마 정보가 생성되지 않는다.

이를 처리하기 위해 iBatis를 통해 조회된 데이터가 0건일 경우 재조회를 통해 ResultSet의 MetaData 정보를 이용하여 스키마 정보를 획득한다.

DAO를 구성할 경우 NexacroIbatisAbstractDAO를 상속받아 처리 한다.

* + 구현에 대한 정보는 기능명세의 DAO 부분을 참고 한다.

아래는 Hsql을 사용할 경우 NexacroIbatisAbstractDAO를 사용하는 sample이다.

selectSampleVoList 조회 시 조회된 값이 0건일 경우 스키마 정보를 확인하여 처리 된다.

|  |
| --- |
| @Repository("sampleDAO")  **public** **class** SampleDAO **extends** NexacroIbatisAbstractDao {  **public** List<SampleVO> selectSampleVoList(SampleVO searchVO) {  **return** list("sampleDAO.selectSampleVOList", searchVO);  }  **public** **void** insertSampleVO(SampleVO sample) {  insert("sampleDAO.insertSampleVO", sample);  }    **public** **void** updateSampleVO(SampleVO sample) {  insert("sampleDAO.updateSampleVO", sample);  }  **public** **void** deleteSampleVO(SampleVO sample) {  insert("sampleDAO.deleteSampleVO", sample);  }  } |

#### mybatis

mybatis를 적용하기 위한 방법은 3가지 형태로 처리 될수 있다.

* DAO class 방식 (기존)
  + @Repository 지정 및 NexacroIbatisAbstractDAO 를 상속
* interface 사용
  + mybatis에서 지원하는 interface 기반 동작
    - 내부적으로 Mapper 인스턴스화 되어져 동작 한다.
  + Spring - @Repository
  + Egovframework - @Mapper
* Mapper Annotation 방식 (raw 방식)
  + 별도의 sqlMap (xml 방식)을 정의할 필요가 없으며, 메서드 상단에 어노테이션 (쿼리)을 통해 실행 된다.
  + 단, xml 방식에 비해 기능이 제한적이다. (동적 SQL X)

아래는 interface를 사용하여 Mapper를 정의한 예제이다. (Spring을 사용할 경우 @Mapper 대신 @Repository 를 사용한다.)

|  |
| --- |
| @Mapper("sampleMapper")  **public** **interface** SampleMapper {  **public** List<SampleVO> selectSampleVoList(SampleVO searchVO);  **public** **void** insertSampleVO(SampleVO sample);    **public** **void** updateSampleVO(SampleVO sample);  **public** **void** deleteSampleVO(SampleVO sample);  } |

### 대용량 데이터

본 예제는 대용량 데이터 조회 시 서버의 성능 및 메모리에 대한 부하를 줄이기 위해 사용할 수 있는 NexacroFirstRowHandler에 대한 예제 이다.

#### iBatis 대용량 처리

1. Controller에서 입력 파라매터로 NexacroFirstRowHandler를 선언한다.
   1. 전송 되는 ContentType을 설정 한다.
2. DAO 호출 시 NexacroFirstRowHandler를 통해 대용량 데이터 처리를 한다.
   1. DAO Sample을 참고하여 DAO class를 구성한다.
   2. queryWithFirstRowHandler(String queryId, Object parameterObject, NexacroFirstRowHandler firstRowHandler, String sendName)를 이용하여 iBatis로 쿼리를 실행하고 조회되는 데이터를 FirstRow로 전송한다.
      1. queryId : sqlMap의 쿼리 id를 입력한다.
      2. parameterObject : 쿼리가 실행할 때 필요한 파라매터를 입력한다.
      3. firstRowHandler : Controller에서 선언한 NexacroFirstRowHandler를 입력한다.
      4. sendName : 쿼리가 실행한 결과에 대한 전송되는 데이터셋의 명칭을 입력한다.

아래는 Controller에서 NexacroFirstRowHandler를 선언하는 sample 이다.

|  |
| --- |
| @RequestMapping(value = "/sampleLargeData.do")  **public** NexacroResult largeData(NexacroFirstRowHandler firstRowHandler  , @ParamVariable(name="firstRowCount" required=false) **int** firstRowCount){  firstRowHandler.setContentType(PlatformType.*CONTENT\_TYPE\_SSV*);  String sendDataSetName = "firstRowData";  **int** initDataCount = *DATA\_CNT*; // this is dummy data!!    largeDataService.selectLargeData  (firstRowHandler, sendDataSetName  firstRowCount, initDataCount);    NexacroResult result = **new** NexacroResult();  **return** result;  } |

아래는 DAO에서 대용량 처리를 하는 sample이다.

|  |
| --- |
| @Repository("largeDataDAO")  **public** **class** LargeDataDAO **extends** NexacroHsqlAbstractDao {    **public** **void** selectLargeData(NexacroFirstRowHandler firstRowHandler  ,String sendName, int firstRowCount) {  queryWithFirstRowHandler("largeDataDAO.selectLargeData", **null**, firstRowHandler, sendName  , firstRowCount);  } |

#### mybatis 대용량 처리.

1. Controller에서 입력 파라매터로 NexacroFirstRowHandler를 선언한다.
   1. 전송 되는 ContentType을 설정 한다.
2. Service에서 Mapper(mybatis interface) 호출 시 NexacroFirstRowHandler를 통해 대용량 데이터 처리를 한다.
   1. Service 클래스 작성 sample을 참고하여 클래스를 작성한다.
   2. 서비스 메서드 내부에서 MybatisRowHandler 를 생성하여 mapper를 실행한다.
   3. 해당 메서드의 파라매터 정보는 다음과 같다.
      1. queryId : sqlMap의 쿼리 id를 입력한다.
      2. parameterObject : 쿼리가 실행할 때 필요한 파라매터를 입력한다.
      3. firstRowHandler : Controller에서 선언한 NexacroFirstRowHandler를 입력한다.
      4. sendName : 쿼리가 실행한 결과에 대한 전송되는 데이터셋의 명칭을 입력한다.
3. Mapper class 작성 시 메서드 Argument로 org.apache.ibatis.session.ResultHandler를 선언한다.

아래는 Controller에서 NexacroFirstRowHandler를 선언하는 sample 이다.

|  |
| --- |
| @RequestMapping(value = "/sampleLargeData.do")  **public** NexacroResult largeData(NexacroFirstRowHandler firstRowHandler  , @ParamVariable(name="firstRowCount" required=false) **int** firstRowCount){  firstRowHandler.setContentType(PlatformType.*CONTENT\_TYPE\_SSV*);  String sendDataSetName = "firstRowData";  **int** initDataCount = *DATA\_CNT*; // this is dummy data!!    largeDataService.selectLargeData  (firstRowHandler, sendDataSetName  firstRowCount, initDataCount);    NexacroResult result = **new** NexacroResult();  **return** result;  } |

아래는 Service 클래스 구현체에서 대용량 처리를 하는 sample이다.

|  |
| --- |
| @Service("largeDataMybatisService")  **public** **class** LargeDataMybatisService **implements** LargeDataService {  @Resource(name = "largeDataMybatisMapper")  **private** LargeDataMybatisMapper largeDataMybatisMapper;  **public** **void** selectLargeData(NexacroFirstRowHandler firstRowHandler  ,String sendName, int firstRowCount) {  MybatisRowHandler rowHandler = **new** MybatisRowHandler(firstRowHandler  , sendName, firstRowCount);  largeDataMybatisMapper.selectLargeData(rowHandler);  rowHandler.sendRemainData();  } |

아래는 Mapper interface 클래스 sample 이다.

|  |
| --- |
| @Repository("largeDataMapper") or @Mapper("largeDataMapper")  **public** **interface** LargeDataMybatisMapper {  **public** **void** selectLargeData(ResultHandler resultHandler);  } |

@Repository – Spring 사용 시 Annotation

@Mapper – EgovFramework 사용 시 Annotation

#### JDBC 대용량 처리.

1. Controller에서 입력 파라매터로 NexacroFirstRowHandler를 선언한다.
   1. 전송 되는 ContentType을 설정 한다.
2. DAO 호출 시 NexacroFirstRowHandler를 통해 대용량 데이터 처리를 한다.
   1. DAO class는 JdbcDaoSupport를 상속받는다.
      1. Jdbc는 iBatis와 같이 추상화된 class는 제공하지 않는다.
   2. Spring의 RowCallHandler를 이용한다. (DBMS 별 처리가 필요하다. – 데이터 타입에 대한)
      1. ResultSet으로부터 메타데이터를 읽어 DataSet을 생성한다.
      2. ResultSet으로부터 데이터를 읽어 DataSet에 할당한다.
      3. NexacroFirstRowHandler를 이용하여 데이터를 전송한다.

아래는 Controller에서 NexacroFirstRowHandler를 선언하는 sample 이다.

|  |
| --- |
| @RequestMapping(value = "/sampleLargeData.do")  **public** NexacroResult largeData(NexacroFirstRowHandler firstRowHandler , Variable(name="firstRowCount" required=false) **int** firstRowCount){  firstRowHandler.setContentType(PlatformType.*CONTENT\_TYPE\_SSV*);  String sendDataSetName = "firstRowData";  **int** initDataCount = *DATA\_CNT*; // this is dummy data!!    largeDataService.selectJdbcLargeData  (firstRowHandler, sendDataSetName, initDataCount);    NexacroResult result = **new** NexacroResult();  **return** result;  } |

아래는 DAO에서 대용량 처리를 하는 sample이다. (Hsql DB를 이용)

|  |
| --- |
| @Repository("largeDataDAO")  **public** **class** LargeJdbcDataDAO **extends** JdbcDaoSupport {    **public** **void** selectLargeData(NexacroFirstRowHandler firstRowHandler  , String sendName) {  String sql = "SELECT LARGE\_ID, NAME FROM TB\_LARGE";  RowCallbackHandler callbackHandler = **new** LargeDataRowCallbackHandler(  firstRowHandler, sendName, **new** Hsql());  getJdbcTemplate().query(sql, callbackHandler);  } |

LargeDataRowCallbackHandler는 Sample의 데이터를 참고한다.

### File handling

#### Upload

파일 데이터를 업로드 하는 경우 Spring의 MultipartHttpServletRequest를 이용하여 처리 한다.

아래는 해당 처리에 대한 sample 이다.

|  |
| --- |
| @RequestMapping(value = "/uploadFiles.do")  **public** NexacroResult uploadFiles(HttpServletRequest request, HttpServletResponse response) **throws** Exception {    **if**(!(request **instanceof** MultipartHttpServletRequest)) {  **if**(log.isDebugEnabled()) {  log.debug("Request is not a MultipartHttpServletRequest");  }  **return** **new** NexacroResult();  }    MultipartHttpServletRequest multipartRequest = (MultipartHttpServletRequest) request;  // parameter and multipart parameter  Enumeration<String> parameterNames = multipartRequest.getParameterNames();    // files..  Map<String, MultipartFile> fileMap = multipartRequest.getFileMap();  String filePath = getFilePath();    Set<String> keySet = fileMap.keySet();  **for**(String name: keySet) {    MultipartFile multipartFile = fileMap.get(name);  String originalFilename = multipartFile.getOriginalFilename();  // upload some logic…  **if**(log.isDebugEnabled()) {  log.debug("uploaded file write success.  file="+originalFilename);  }  }  **return** **new** NexacroResult();  } |

#### Download

파일을 다운로드 할 경우 NexacroFileResult를 이용하여 데이터를 전달한다.

아래는 파일 출력에 대한 sample 이다.

|  |
| --- |
| @RequestMapping(value = "/advancedDownloadFile.do")  **public** NexacroFileResult downloadFile(  HttpServletRequest request  , @RequestParam String fileName  ) **throws** Exception {    String characterEncoding = request.getCharacterEncoding();  **if**(characterEncoding == **null**) {  characterEncoding = PlatformType.*DEFAULT\_CHAR\_SET*;  }  String realFileName;  // some logic…  File file = **new** File(realFileName);    NexacroFileResult result = **new** NexacroFileResult(file);  // default - application/octet-stream  // result.setContentType(contentType); // set MIME TYPE    **return** result;  } |

### 암호화

암호화는 nexacro platfrom에서 제공하지 않는 기능이고 구간암호화를 정상적으로 하기 위해서는 HTTPS 혹은 상용 암호화솔루션을 사용하는 방법을 권장한다.

다만 이번에 샘플로 제공한 이유는 중요한 데이터(패스워드, 주민번호 등)를 암호화 알고리즘을 활용하여 값을 변환 할 수 있다는 예제이며, nexacro platform에서 protocol을 추가하여 암호화 처리를 할 수 있다는 것을 보여주기 위함이다.

(샘플을 프로젝트에 바로 적용할 경우 보안상 심각한 문제를 야기 할 수 있으며, 반드시 주의가 필요하다.)

### Validation

Spring에서는 사용자가 입력한 값에 대한 유효성을 체크하기 위해 JSR-303 Validator를 사용할 수 있도록 지원하고 있다. JSR-303 Validator를 적용하기 위해서는 아래의 dependency를 pom.xml에 추가 후 dispatcher-servlet.xml, VO Class, Controller Class를 아래의 예제와 같이 작성한다.

아래는 사용자가 입력한 값을 Validation하는 Sample이다.

pom.xml

|  |
| --- |
| <dependencies>  ....  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-validator</artifactId>  <version>4.0.0.GA</version>  </dependency>  ....  </dependencies> |

dispatcher-servlet.xml에 추가

|  |
| --- |
| <bean id=*"validator"* class=*"org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"*> |

VO Class 작성 예제

|  |
| --- |
| **import** javax.validation.constraints.NotNull;  **import** javax.validation.constraints.Pattern;  **import** javax.validation.constraints.Size;  // Fields  @NotNull  @Size(min = 4, max = 20, message = "Please, Enter your id at least 4 Characters.")  **private** String userId;    @NotNull  @Size(max = 20, message = "Please, Enter your name.")  **private** String userName;    @NotNull  @Size(min = 4, max = 50, message = "Please, Enter your password at least 4 Characters.")  **private** String password;  @Pattern(regexp = ".+@.+\\.[a-z]+", message = "Please, Verify your e-mail address.")  **private** String email;  ..... |

아래는 JSR-303 지원 Bean Validation 목록이다.

|  |  |
| --- | --- |
| Annotation | 설명 |
| @AssertFalse | 필드 값이 false인지 확인 |
| @AssertTrue | 필드 값이 true인지 확인 |
| @DecimalMax | 필드 값이 가질 수 있는 최대값을 확인 |
| @DecimalMin | 필드 값이 가질 수 있는 최소값을 확인 |
| @Digits | 필드 값이 가질 수 있는 자리수 확인  (정수부(interger), 소수부(fraction) 자리수를 지정할 수 있다.) |
| @Future | 현재일보다 이후인지 확인 |
| @Max | 필드 값이 가질 수 있는 최대 길이를 확인 |
| @Min | 필드 값이 가질 수 있는 최소 길이를 확인 |
| @NotNull | 필드 값이 Null이 아닌지 확인 |
| @Null | 필드 값이 Null인지 확인 |
| @Past | 현재일보다 이전인지 확인 |
| @Pattern | 필드 값에 패턴을 적용해 값이 유효한지 확인 |
| @Size | 필드 값이 가질 수 있는 길이(min, max)를 확인 |
| @Valid | 필드 값에 대해 Validation 확인 |

Bean Validation에 대한 다양한 예제는 아래의 주소에서 얻을 수 있다.

<http://beanvalidation.org/1.0/spec/>

Controller Class 예제

|  |
| --- |
| .....  **import** org.springframework.validation.BeanPropertyBindingResult;  **import** org.springframework.validation.BindingResult;  **import** org.springframework.validation.ObjectError;  **import** org.springframework.validation.Validator;  **import** org.springframework.web.bind.WebDataBinder;  **import** org.springframework.web.bind.annotation.InitBinder;  **import** org.springframework.web.bind.annotation.RequestMapping;  .....  @Resource  **private** Validator validator;  @InitBinder  **public** **void** initBinder(WebDataBinder dataBinder){  dataBinder.setValidator(**this**.validator);  }  .....  @RequestMapping(value = "/userModifyVO.do")  **public** NexacroResult modifyVO(  @ParamDataSet(name = "input1" required=false) List<UserVO> modifyList,  PlatformData platformData) {  ....  validate(modifyList);  ....  }  **private** **void** validate(List<UserVO> modifyList) **throws** NexacroException {  BindingResult bindingResult = **null**;  **for** (UserVO userVO : modifyList) {  bindingResult = **new** BeanPropertyBindingResult(userVO, "userVO");  validator.validate(userVO, bindingResult);  **if** (bindingResult.hasErrors()) {  String errorMessages = getErrorMessages(bindingResult);  NexacroException nexacroException = **new** NexacroException(errorMessages);  nexacroException.setErrorCode(NexacroException.*DEFAULT\_ERROR\_CODE*);  nexacroException.setErrorMsg(errorMessages);  **throw** nexacroException;  }  }  }    **private** String getErrorMessages(BindingResult bindingResult) {  StringBuffer sb = **new** StringBuffer();    **for** (ObjectError error : bindingResult.getAllErrors()) {  sb.append(error.getDefaultMessage()).append("\n");  }    **return** sb.toString();  } |

Validation에 대한 자세한 정보는 아래의 주소에서 얻을 수 있다.

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html>

### 예외처리

예외 처리가 필요한 경우 아래의 예제를 참고할 수 있다.

dispatcher-servlet.xml 설정

|  |
| --- |
| <!-- nexacro exception resolver -->  <bean id=*"exceptionResolver"* class=*"com.nexacro.spring.resolve.NexacroMappingExceptionResolver"* p:order=*"1"*>  <property name=*"view"* ref=*"nexacroView"* />  <property name=*"shouldLogStackTrace"* value=*"true"* />  <property name=*"shouldSendStackTrace"* value=*"true"* />  <!-- shouldSendStackTrace 가 false 일 경우 nexacro platform으로 전송되는 에러메시지 -->  <property name=*"defaultErrorMsg"* value=*"fail.common.msg"* />  <property name=*"messageSource"* value=*"messageSource"* />  </bean> |

shouldLogStackTrace와 shouldSendStackTrace의 value를 true또는 false로 변경할 수 있다.

* shouldLogStackTrace : 에러 정보를 logging 할지에 대한 여부
* shouldSendStackTrace : Exception의 실제 에러메세지를 응답으로 전송할지에 대한 여부
  + 보안상의 이유로 운영시에는 false로 처리한다.

예외가 필요한 부분에 아래의 예제와 같은 소스를 작성한다.

|  |
| --- |
| **...**  **throw** **new** NexacroException*("Error Message 내용 작성", -1, “send Error Message”*); |

NexacroException.setErrorMsg를 지정하지 않은 경우 shouldSendStackTrace의 설정에 따라 defaultErrorMsg 혹은 Exception의 메시지 정보가 응답으로 전송된다.