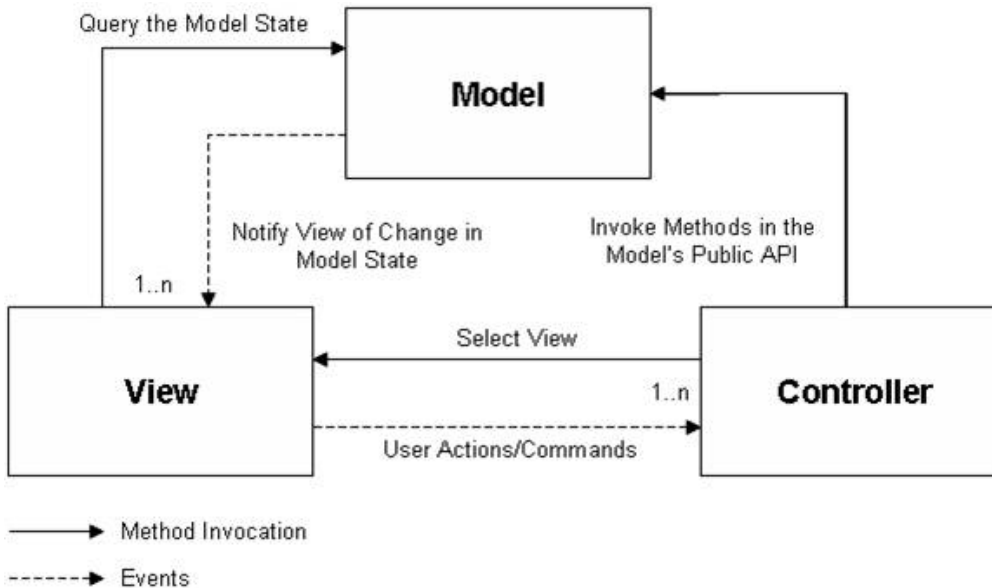


## 개요

- MVC 서비스
  - 개요
  - 설명
    - Spring MVC
    - 예제 실행
  - 참고자료



MVC(Model-View-Controller) 패턴은 코드를 기능(역)에 따라 Model, View, Controller 3가지 요소로 분리한다.

- Model : 어플리케이션의 데이터와 비즈니스 로직을 담는 객체이다.
- View : Model의 정보를 사용자에게 표시한다. 하나의 Model을 다양한 View에서 사용할 수 있다.
- Controller : Model과 View의 중계역할을 한다. 사용자의 요청을 받아 Model에 변경된 상태를 반영하고, 응답을 위한 View를 선택한다.

MVC 패턴은 UI 코드와 비즈니스 코드를 분리함으로써 종속성을 줄이고, 재사용성을 높이고, 보다 쉬운 변경이 가능하도록 한다.

MVC 패턴이 Web Framework에만 사용되는 단어는 아니지만, 전자정부프레임워크에서 "MVC 서비스"란 MVC 패턴을 활용한 Web MVC Framework를 의미한다.

## 설명

오픈소스 Web MVC Framework에는 Spring MVC, Struts, Webwork, JSF등이 있으며, 각각의 장점을 가지고 사용되고 있다. 기능상에서 큰차이는 없으나, 아래와 같은 장점을 고려 전자정부프레임워크에서는 Spring Web MVC를 MVC 서비스의 기반 오픈 소스로 채택하였다.

- Framework내의 특정 클래스를 상속하거나, 참조, 구현해야 하는 등의 제약사항이 비교적 적다. Controller(Spring 2.5 @MVC)나 Form 클래스등이 좀 더 POJO-style에 가까워 비즈니스 로직에 집중된 코드를 작성할 수 있다.
- IOC Container가 Spring 이라면 (간단한 설정으로 Struts나 Webwork같은 Web Framework를 사용할 수 있겠지만) 연계를 위한 추가 설정없이 Spring MVC를 사용할 수 있다. 전자정부프레임워크의 IOC Container는 Spring이다.
- 오픈소스 프로젝트가 활성화(꾸준한 기능 추가, 빠른 bug fix와 Q&A) 되어 있으며 로드맵이 신뢰할만 하다.
- 국내 커뮤니티 활성화 정도, 관련 참고문서나 도서를 쉽게 구할 수 있다.

## Spring MVC

Spring MVC 에 대한 설명은 아래 상세 페이지를 참고하라.

- [Spring MVC Architecture](#)
- [DispatcherServlet](#)
- [HandlerMapping](#)

- Controller
- Annotation-based Controller
- Validation
- Declarative Validation
- View
- SpEL

## 예제 실행

- [easycompany 설치 가이드](#) : MVC와 Ajax Support, Security의 예제코드인 easycompany 설치와 실행 방법을 가이드 한다.

## 참고자료

-  SUN Java BluePrints, Model-View-Controller

## 개요

- Spring MVC Architecture
  - 개요
  - 설명
  - 참고자료

Spring Framework은 간단한 설정만으로 Struts나 Webwork같은 Web Framework을 사용할 수 있지만, 자체적으로 MVC Web Framework을 가지고 있다.

Spring MVC는 기본요소인 Model, View, Controller 외에도, 아래와 같은 특성을 가지고 있다.

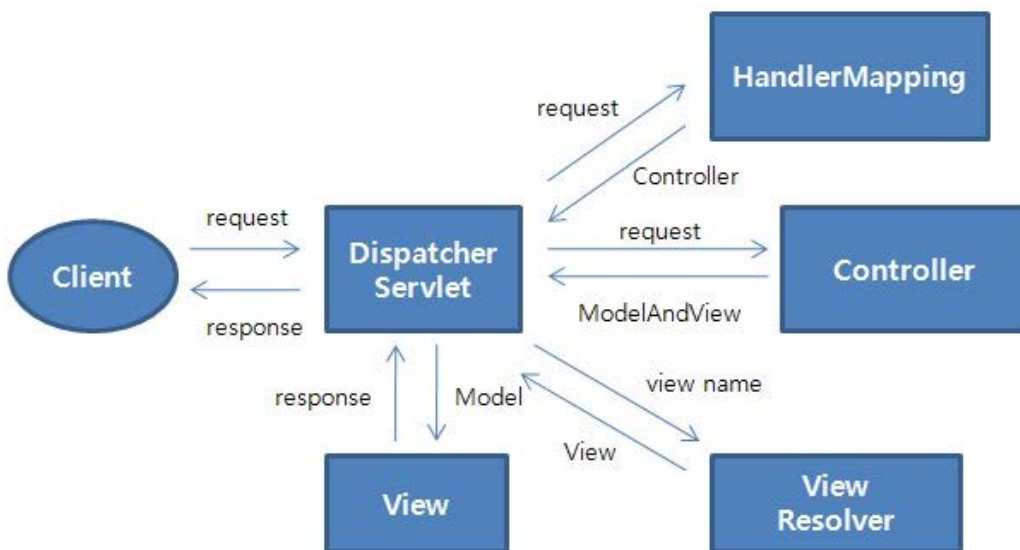
- DispatcherServlet, HandlerMapping, Controller, Interceptor, ViewResolver, View등 각 컴포넌트들의 역할이 명확하게 분리되어 있다.
- HandlerMapping, Controller, View등 컴포넌트들에 다양한 인터페이스 및 구현 클래스를 제공함으로써 경우에 따라 선택하여 사용할 수 있다.
- Controller(@MVC)나 폼 클래스(커맨드 클래스) 작성시에 특정 클래스를 상속받거나 참조할 필요 없이 POJO 나 POJO-style의 클래스를 작성함으로써 비즈니스 로직에 집중한 코드를 작성할 수 있다.
- 웹요청 파라미터와 커맨드 클래스간에 데이터 매핑 기능을 제공한다.
- 데이터 검증을 할 수 있는, Validator와 Error 처리 기능을 제공한다.
- JSP Form을 쉽게 구성하도록 Tag를 제공한다.

## 설명

Spring MVC(Model-View-Controller)의 핵심 Component는 아래와 같다.

Component	개요
DispatcherServlet	Spring MVC Framework의 Front Controller, 웹요청과 응답의 Life Cycle을 주관한다.
HandlerMapping	웹요청시 해당 URL을 어떤 Controller가 처리할지 결정한다.
Controller	비즈니스 로직을 수행하고 결과 데이터를 ModelAndView에 반영한다.
ModelAndView	Controller가 수행 결과를 반영하는 Model 데이터 객체와 이동할 페이지 정보(또는 View객체)로 이루어져 있다.
ViewResolver	어떤 View를 선택할지 결정한다.
View	결과 데이터인 Model 객체를 display한다.

이들 컴포넌트간의 관계와 흐름을 그림으로 나타내면 아래와 같다.



1. Client의 요청이 들어오면 DispatcherServlet이 가장 먼저 요청을 받는다.
2. HandlerMapping이 요청에 해당하는 Controller를 return한다.
3. Controller는 비즈니스 로직을 수행(호출)하고 결과 데이터를 ModelAndView에 반영하여 return한다.

4. `ViewResolver`는 `view name`을 받아 해당하는 `View` 객체를 `return`한다.
5. `View`는 `Model` 객체를 받아 `rendering`한다.

이 가이드문서는 Spring 2.5.6 버전을 기준으로 작성되었다.

## 참고자료

- The Spring Framework - Reference Documentation 2.5.6
- Spring Framework [API](#) Documentation 2.5.6

## 개요

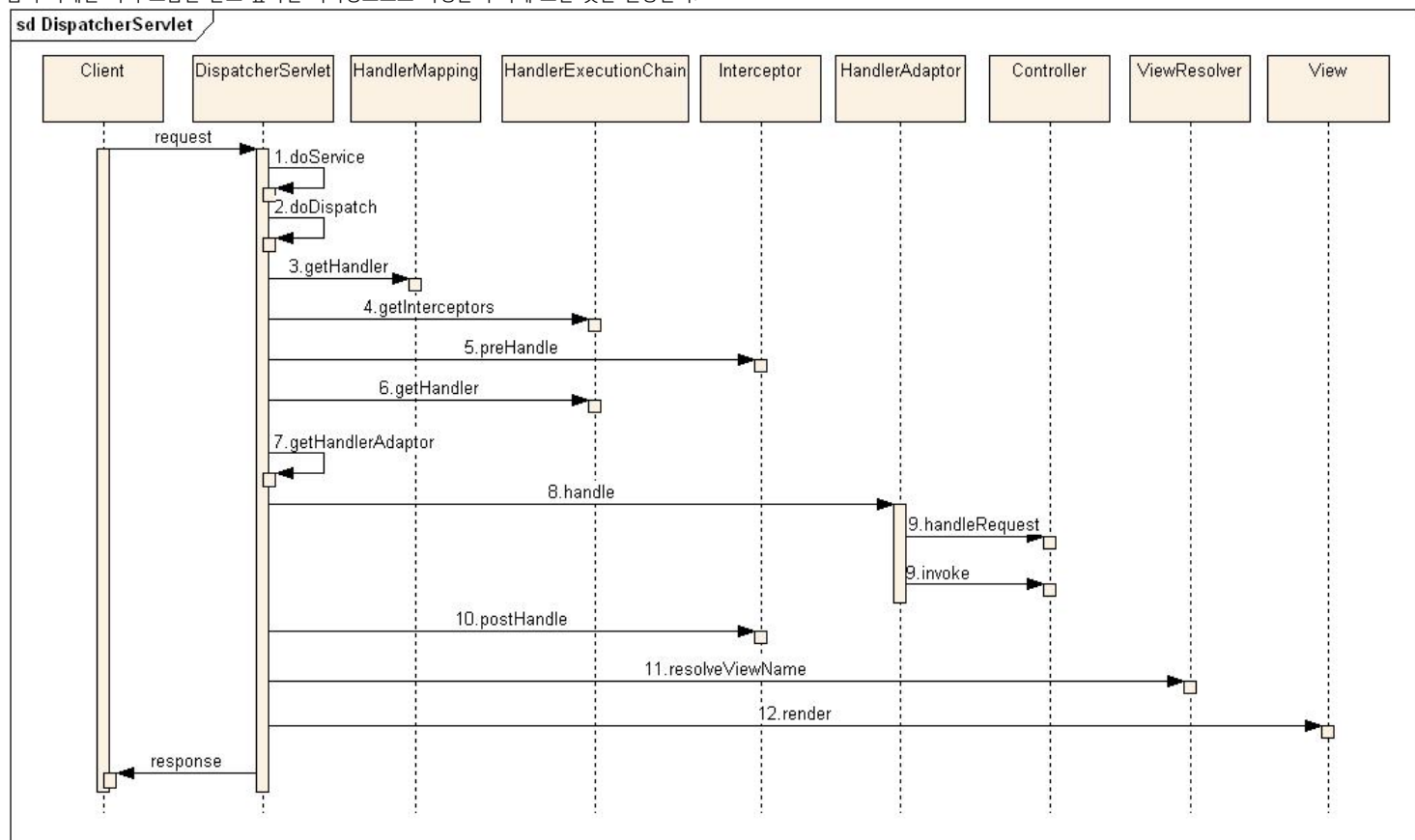
Spring MVC Framework의 유일한 Front Controller인 DispatcherServlet은 Spring MVC의 핵심 요소이다. DispatcherServlet은 Controller로 향하는 모든 웹요청의 진입점이며, 웹요청을 처리하며, 결과 데이터를 Client에게 응답 한다. DispatcherServlet은 Spring MVC의 웹요청 Life Cycle을 주관한다 할 수 있다.

- DispatcherServlet
  - 개요
  - 설명
  - DispatcherServlet에서의 웹요청 흐름
  - web.xml에 DispatcherServlet 설정하기
  - 참고자료

## 설명

### DispatcherServlet에서의 웹요청 흐름

Client의 웹요청시에 DispatcherServlet에서 이루어지는 처리 흐름은 아래와 같다. 좀더 자세한 처리 흐름을 알고 싶다면 디버깅모드로 과정을 추적해 보는 것을 권장한다.



1. doService 메소드에서부터 웹요청의 처리가 시작된다. DispatcherServlet에서 사용되는 몇몇 정보를 request 객체에 담는 작업을 한 후 doDispatch 메소드를 호출한다.
2. 아래 3번~13번 작업이 doDispatch 메소드안에 있다. Controller, View 등의 컴포넌트들을 이용한 실제적인 웹요청처리가 이루어 진다.
3. getHandler 메소드는 RequestMapping 객체를 이용해서 요청에 해당하는 Controller를 얻게 된다.
4. 요청에 해당하는 Handler를 찾았다면 Handler를 HandlerExecutionChain 객체에 담아 리턴하는데, 이때 HandlerExecutionChain는 요청에 해당하는 interceptor들이 있다면 함께 담아 리턴한다.
5. 실행될 interceptor들이 있다면 interceptor의 preHandle 메소드를 차례로 실행한다.
6. Controller의 인스턴스는 HandlerExecutionChain의 getHandler 메소드를 이용해서 얻는다.
7. HandlerMapping과 마찬가지로 여러개의 HandlerAdaptor를 설정할 수 있는데, getHandlerAdaptor 메소드는 Controller에 적절한 HandlerAdaptor 하나를 리턴한다.
8. 선택된 HandlerAdaptor의 handle 메소드가 실행되는데, 실제 실행은 파라미터로 넘겨 받은 Controller를 실행한다.
9. 계층형 Controller인 경우는 handleRequest 메소드가 실행된다. @Controller인 경우는 HandlerAdaptor(AnnotationMethodHandlerAdapter)가 HandlerMethodInvoker를 이용해 실행할 Controller의 메소드를 invoke()한다.
10. interceptor의 postHandle 메소드가 실행된다.
11. resolveViewName 메소드는 논리적 뷰 이름을 가지고 해당 View 객체를 반환한다.
12. Model 객체의 데이터를 보여주기 위해 해당 View 객체의 render 메소드가 수행된다.

### web.xml에 DispatcherServlet 설정하기

Spring MVC Framework를 사용하기 위해서는 web.xml에 DispatcherServlet을 설정하고, DispatcherServlet이 WebApplicationContext를 생성할수 있도록 빈(Beans) 정보가 있는 파일들도 설정해주어야 한다.

#### 기본 설정

```

<web-app>
  <!-- easycompnay라는 웹어플리케이션의 웹요청을 DispatcherServlet이 처리한다. -->
  <servlet>
    <servlet-name>easycompany</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  </servlet>
</web-app>
    
```

servlet-name은 DispatcherServlet이 기본(default)으로 참조할 빈 설정 파일 이름의 prefix가 되는데, (servlet-name)-servlet.xml 같은 형태이다. 위 예제와 같이 web.xml을 작성했다면 DispatcherServlet은 기본으로 /WEB-INF/easycompany-servlet.xml을 찾게 된다.

#### contextConfigLocation을 이용한 설정

빈 설정 파일을 하나 이상을 사용하거나, 파일 이름과 경로를 직접 지정해주고 싶다면 contextConfigLocation 라는 초기화 파라미터 값에 빈 설정 파일 경로를 설정해준다.

```
...
<servlet>
  <servlet-name>easycompany</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/config/easycompany-web.xml
    </param-value>
  </init-param>
</servlet>
```

ContextLoaderListener를 이용한 설정

일반적으로 빈 설정 파일은 하나의 파일만 사용되기 보다는 persistence, service, web등 layer 단위로 나뉘게 된다. 또한, 같은 persistence, service layer의 빈을 2개 이상의 DispatcherServlet이 공통으로 사용할 경우도 있다. 이럴때는 공통빈(persistence, service)설정 정보는 ApplicationContext에, web layer의 빈들은 WebApplicationContext에 저장하는 아래와 같은 방법을 추천한다. 공통빈 설정 파일은 서블릿 리스너로 등록된 org.springframework.web.context.ContextLoaderListener로 로딩해서 ApplicationContext을 만들고, web layer의 빈설정 파일은 DispatcherServlet이 로딩해서 WebApplicationContext을 만든다.

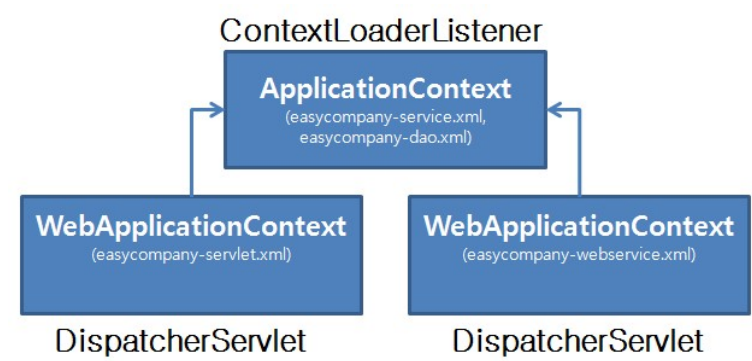
```
....
<!-- ApplicationContext 빈 설정 파일-->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    <!-- 빈 설정 파일들간에 구분은 줄바꿈(\n), 콤마(,), 세미콜론(;) 등으로 한다.-->
    /WEB-INF/config/easycompany-service.xml,/WEB-INF/config/easycompany-dao.xml
  </param-value>
</context-param>

<!-- 웹 어플리케이션이 시작되는 시점에 ApplicationContext을 로딩하며, 로딩된 빈정보는 모든 WebApplicationContext들이 참조할 수 있다.-->
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<servlet>
  <servlet-name>employee</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/config/easycompany-service.xml
    </param-value>
  </init-param>
</servlet>

<servlet>
  <servlet-name>webservice</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/config/easycompany-webservice.xml
    </param-value>
  </init-param>
</servlet>
....
```

이 ApplicationContext의 빈 정보는 모든 WebApplicationContext들이 참조할 수 있다. 예를 들어, DispatcherServlet은 2개 사용하지만 같은 Service, DAO를 사용하는 web.xml을 아래와 같이 작성했다면, easycompany-servlet.xml에 정의된 빈정보는 easycompany-webservice.xml가 참조할 수 없지만, easycompany-service.xml, easycompany-dao.xml에 설정된 빈 정보는 easycompany-servlet.xml, easycompany-webservice.xml 둘 다 참조한다. ApplicationContext과 WebApplicationContext과의 관계를 그림으로 나타내면 아래와 같다.



참고자료

- The Spring Framework - Reference Documentation 2.5.6
- Spring Framework API Documentation 2.5.6

# HandlerMapping

## Table of Contents

### 개요

DispatcherServlet에 Client로부터 Http Request가 들어 오면 HandlerMapping은 요청처리를 담당할 Controller를 mapping한다.

Spring MVC는 interface인 HandlerMapping의 구현 클래스도 가지고 있는데, 용도에 따라 여러 개의 HandlerMapping을 사용하는 것도 가능하다.

빈 정의 파일에 HandlerMapping에 대한 정의가 없다면 Spring MVC는 기본(default) HandlerMapping을 사용한다.

기본 HandlerMapping은 BeanNameUrlHandlerMapping이며, jdk1.5 이상의 실행환경이면, DefaultAnnotationHandlerMapping 역시 기본 HandlerMapping이다.

- HandlerMapping
  - 개요
  - 설명
    - BeanNameUrlHandlerMapping
    - ControllerClassNameHandlerMapping
    - SimpleUrlHandlerMapping
    - DefaultAnnotationHandlerMapping
    - SimpleUrlAnnotationHandlerMapping
  - 참고자료

### 설명

BeanNameUrlHandlerMapping, SimpleUrlHandlerMapping 등 주요 HandlerMapping 구현 클래스는 상위 추상 클래스인 AbstractHandlerMapping과 AbstractUrlHandlerMapping을 확장하기 때문에 이 추상클래스들의 프로퍼티를 사용한다. 주요 프로퍼티는 아래와 같다.

- defaultHandler : 요청에 해당하는 Controller가 없을 경우, defaultHandler에 등록된 Controller를 반환한다.
- alwaysUseFullPath : URL과 Controller 매핑시에 URL full path를 사용할지 여부를 나타낸다.  
예를 들어, servlet-mapping이 /easycompany/\* 이고, alwaysUseFullPath가 true이면 /easycompany/employeeList.do, alwaysUseFullPath가 false이면 /employeeList.do 이다.
- interceptors : Controller가 요청을 처리하기 전,후로 특정한 로직을 수행되기 원할때 interceptor를 등록한다. 복수개의 interceptor를 등록할 수 있다. interceptor에 대한 자세한 설명은 이곳을 참고하라.
- order : 여러개의 HandlerMapping 사용시에 우선순위를 정한다.

```
...  
<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" p:order="2"/>  
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping" p:order="3">  
...
```

- pathMatch : 사용자 요청 URL path와 설정정보의 URL path를 매칭할때, 특정 스타일의 매칭을 지원하는 PathMatcher를 등록할 수 있다. 기본값은 Ant-style의 패턴매칭을 제공하는 AntPathMatcher이다.

Spring MVC가 제공하는 주요 HandlerMapping 구현 클래스는 아래와 같다.

- BeanNameUrlHandlerMapping
- ControllerClassNameHandlerMapping
- SimpleUrlHandlerMapping
- DefaultAnnotationHandlerMapping

@MVC에서 DefaultAnnotationHandlerMapping은 URL 단위로 interceptor를 적용할 수 없기에 전자정부프레임워크에서 아래와 같은 HandlerMapping 구현 클래스를 추가했다.

- SimpleUrlAnnotationHandlerMapping

## BeanNameUrlHandlerMapping

BeanNameUrlHandlerMapping은 bean 정의 태그에서 name attribute에 선언된 URL과 class attribute에 정의된 Controller를 매핑하는 방식으로 동작한다.

예를 들어, 아래와 같이 정의되어 있다면,

```
<beans ...>  
...  
<!--HandlerMapping이 BeanNameUrlHandlerMapping 밖에 없다면 BeanNameUrlHandlerMapping에 대한 별도의 빈정의는 필요 없다.-->  
<!--<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>-->  
<bean name="/insertEmployee.do" class="com.easycompany.controller.InsertEmployeeController">  
...  
</bean>  
...  
</beans>
```

Client에서 URL ~/insertEmployee.do 요청이 들어오면 InsertEmployeeController 클래스가 요청 처리를 담당한다.

앞 개요에서 언급했듯이 WAC(WebgApplicationContext)에 HandlerMapping 빈정의가 없다면 BeanNameUrlHandlerMapping 이 (별도의 빈 정의 없이) 사용된다.

하지만, SimpleUrlHandlerMapping 같은 다른 HandlerMapping과 같이 써야 한다면, BeanNameUrlHandlerMapping도 bean 정의가 되어야 한다.

```
<beans ...>
...
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    ...
  </property>
</bean>

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
<bean name="/insertEmployee.do" class="com.easycompany.controller.InsertEmployeeController">
  ...
</bean>
...
</beans>
```

## ControllerClassNameHandlerMapping

ControllerClassNameHandlerMapping은 빈정의된 Controller의 클래스 이름중 suffix인 Controller를 제거한 나머지 이름의 소문자로 url mapping한다.

```
<beans ...>
...
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>
<bean class="com.easycompany.controller.hierarchy.EmployeeListController"/>
...
</beans>
```

빈 정의가 위와 같다면, EmployeeListController ↔ /employeeList\*, InsertEmployeeController ↔ /insertemployee\* 과 같이 url mapping이 이루어 진다.

ControllerClassNameHandlerMapping에 프로퍼티 값으로 caseSensitive나 pathPrefix, basePackage등을 설정할 수 있는데,

- caseSensitive : Controller 이름으로 URL 경로 mapping시에 대문자 사용여부. (ex. /insertemployee\* 가 아니라 /easycompany/insertEmployee\*로 사용하기 원할때).
- pathPrefix : URL 경로에 기본적인 prefix 값. 기본값은 false이다.
- basePackage : URL mapping에 사용되는 Controller의 기본 패키지 이름이다. 사용되는 Controller의 패키지명에 기본 패키지에 추가되는 subpackage가 있다면 해당 subpackage 이름이 URL 경로에 추가된다.

```
<beans ...>
...
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping">
  <property name="pathPrefix" value="/easycompany"/>
  <property name="caseSensitive" value="true"/>
  <property name="basePackage" value="com.easycompany.controller"/>
</bean>

<bean class="com.easycompany.controller.hierarchy.EmployeeListController"/>
<bean class="com.easycompany.controller.hierarchy.InsertEmployeeController"/>
...
</beans>
```

하면, EmployeeListController ↔ /easycompany/hierarchy/employeeList\*, InsertEmployeeController ↔ /easycompany/hierarchy/insertEmployee\* 과 같이 url mapping이 이루어 진다.

## SimpleUrlHandlerMapping

SimpleUrlHandlerMapping은 Ant-Style 패턴 매칭을 지원하며, 하나의 Controller에 여러 URL을 mapping 할 수 있다. proerty의 key 값에 URL 패턴을 지정하고 value에는 Controller의 id 혹은 이름을 지정한다.

```
<beans ...>
...
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/employeeList.do">employeeListController</prop>
      <prop key="/insertEmployee.do">insertEmployeeController</prop>
      <prop key="/updateEmployee.do">updateEmployeeController</prop>
      <prop key="/loginProcess.do">loginController</prop>
      <prop key="/**/login.do">staticPageController</prop>
      <prop key="/static/*.html">staticPageController</prop>
    </props>
  </property>
</bean>

<bean id="loginController" class="com.easycompany.controller.hierarchy.LoginController"/>
<bean id="employeeListController" class="com.easycompany.controller.hierarchy.EmployeeListController"/>
<bean id="insertEmployeeController" class="com.easycompany.controller.hierarchy.InsertEmployeeController"/>
<bean id="updateEmployeeController" class="com.easycompany.controller.hierarchy.UpdateEmployeeController"/>
<bean id="staticPageController" class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
...
</beans>
```

SimpleUrlHandlerMapping을 사용하면 Interceptor를 특정 URL 단위로 적용하는게 가능하다.

프로퍼티 interceptors에 적용하려는 Interceptor들을 리스트로 선언해주면 된다.

URL /employeeList.do, /insertEmployee.do, /updateEmployee.do 요청에 대해서 사용자 인증여부를 interceptor로 검증한다고 하면,아래와 같이 정의한다.

```
<beans ...>
```



```

...
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="interceptors">
    <list>
      <ref local="authenticInterceptor"/>
    </list>
  </property>
  <property name="mappings">
    <props>
      <prop key="/employeeList.do">employeeListController</prop>
      <prop key="/insertEmployee.do">insertEmployeeController</prop>
      <prop key="/updateEmployee.do">updateEmployeeController</prop>
    </props>
  </property>
</bean>
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/**/login.do">staticPageController</prop>
      <prop key="/static/*.html">staticPageController</prop>
    </props>
  </property>
</bean>

<bean id="loginController" class="com.easycompany.controller.hierarchy.LoginController"/>
<bean id="employeeListController" class="com.easycompany.controller.hierarchy.EmployeeListController"/>
<bean id="insertEmployeeController" class="com.easycompany.controller.hierarchy.InsertEmployeeController"/>
<bean id="updateEmployeeController" class="com.easycompany.controller.hierarchy.UpdateEmployeeController"/>
<bean id="staticPageController" class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

<bean id="authenticInterceptor" class="com.easycompany.interceptor.AuthenticInterceptor"/>
...
</beans>

```

## DefaultAnnotationHandlerMapping

@MVC 개발을 하려면 위에서 언급한 HandlerMapping이 아니라 DefaultAnnotationHandlerMapping을 사용해야 한다. 단, jdk 1.5 이상의 개발환경이어야 한다.

jdk 1.5이상의 개발환경이라면, BeanNameUrlHandlerMapping과 함께 DefaultAnnotationHandlerMapping도 기본 HandlerMapping이다.

따라서 빈 설정 파일에 별도로 선언해주지 않아도 된다. (단, 다른 HandlerMapping과 함께 사용한다면 선언해주어야 한다.)

아래와 같이 컴포넌트 스캔할 패키지를 지정해 주면,

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-2.5.xsd">

  <context:component-scan base-package="org.mycode.controller" />

</beans>

```

패키지 org.mycode.controller 아래의 @Controller중에 @RequestMapping에 선언된 URL과 해당 @Controller 클래스의 메소드와 매핑한다.

간단한 예제를 보면,

```

package org.mycode.controller;

...
@Controller
public class HelloController {

    @RequestMapping(value="/hello.do")
    public String hellomethod() {
        ....
    }
}

```

/hello.do로 URL 요청이 들어 오면 HelloController의 메소드 hellomethod가 실행된다.

## SimpleUrlAnnotationHandlerMapping

DefaultAnnotationHandlerMapping에 interceptor를 등록하면, 모든 @Controller에 interceptor가 적용되는 문제점이 있다. SimpleUrlAnnotationHandlerMapping은 @Controller 사용시에 url 단위로 Interceptor를 적용하기 위해 개발됐다.

비슷한 고민과 비슷한 해결 방법을 제시한 분이 있다. [Scott Murphy의 블로그](#)를 참고하라.

기능상에 큰 차이가 없는것 같아 일단 개발한 소스를 소개한다.

url 단위(Controller의 메소드 단위)로 Interceptor를 적용할 수 있는 대안이 Spring Source에서 나온다면

SimpleUrlAnnotationHandlerMapping는 deprecated 되어야 한다.

SimpleUrlAnnotationHandlerMapping은 아래와 같은 3가지 사항이 고려됐다.

- HandlerMapping:Interceptors 관계의 스프링 구조를 깨뜨리지 말자. (ex. Controller:Interceptor (X))
- 쉬운 사용을 위해 기존의 HandlerMapping과 비슷한 방식의 사용법을 선택하자. (ex.SimpleUrlHandlerMapping)
- 최소한의 커스터마이징을 하자. → 짧은 시간... 또한 추후 deprecated시에 시스템에 영향을 최소화 하기 위해.

웹 어플리케이션이 초기 구동될때, DefaultAnnotationHandlerMapping은 2가지 주요한 작업을 한다. (다른 HandlerMapping도 유사한 작업을 한다.)

1. @RequestMapping의 url 정보를 읽어 들여 해당 Controller와 url간의 매핑 작업.
2. 설정된 Interceptor들에 대한 정보를 읽어 들임.

1번 작업은 DefaultAnnotationHandlerMapping의 상위 클래스인 AbstractDetectingUrlHandlerMapping에서 이루어 지는데, 매핑을 위한 url리스트를 가져오는 determineUrlsForHandler 메소드는 하위 클래스에서 구현하도록 abstract 선언 되어 있다.

```
public abstract class AbstractDetectingUrlHandlerMapping extends AbstractUrlHandlerMapping {
    ...
    protected void detectHandlers() throws BeansException {
        if (logger.isDebugEnabled()) {
            logger.debug("Looking for URL mappings in application context: " + getApplicationContext());
        }
        String[] beanNames = (this.detectHandlersInAncestorContexts ?
            BeanFactoryUtils.beanNamesForTypeIncludingAncestors(getApplicationContext(),
                Object.class) :
            getApplicationContext().getBeanNamesForType(Object.class));

        // Take any bean name that we can determine URLs for.
        for (int i = 0; i < beanNames.length; i++) {
            String beanName = beanNames[i];
            String[] urls = determineUrlsForHandler(beanName);
            if (!ObjectUtils.isEmpty(urls)) {
                // URL paths found: Let's consider it a handler.
                registerHandler(urls, beanName);
            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Rejected bean name '" + beanNames[i] + "': no URL paths
identified");
                }
            }
        }
    }
    protected abstract String[] determineUrlsForHandler(String beanName);
}
```

DefaultAnnotationHandlerMapping의 determineUrlsForHandler 메소드는 @RequestMapping의 url 리스트를 전부 가져오기 때문에, 빈 설정 파일에 정의한 url 리스트만 가져오도록 SimpleUrlAnnotationHandlerMapping에서 determineUrlsForHandler 메소드를 구현 한다.

```
package egovframework.rte.ptl.mvc.handler;
...
public class SimpleUrlAnnotationHandlerMapping extends DefaultAnnotationHandlerMapping {

    //url리스트, 중복값을 허용하지 않음으로 Set 객체에 담는다.
    private Set<String> urls;

    public void setUrls(Set<String> urls) {
        this.urls = urls;
    }

    /**
     * @RequestMapping로 선언된 url 중에 프로퍼티 urls에 정의된 url만 remapping해 return
     * url mapping시에는 PathMatcher를 사용하는데, 별도로 등록한 PathMatcher가 없다면 AntPathMatcher를 사용한다.
     * @param urlsArray - @RequestMapping로 선언된 url list
     * @return urlsArray중에 설정된 url을 필터링해서 return.
     */
    private String[] remappingUrls(String[] urlsArray) {
        if(urlsArray==null){
            return null;
        }
        ArrayList<String> remappedUrls = new ArrayList<String>();
        for(Iterator<String> it = this.urls.iterator(); it.hasNext();){
            String urlPattern = (String)it.next();
            for(int i=0;i<urlsArray.length;i++){
                if(getPathMatcher().matchStart(urlPattern, urlsArray[i])){
                    remappedUrls.add(urlsArray[i]);
                }
            }
        }
        return (String[]) remappedUrls.toArray(new String[remappedUrls.size()]);
    }

    /**
     * @RequestMapping로 선언된 url을 필터링하기 위해
     * org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping의
     * 메소드 protected String[] determineUrlsForHandler(String beanName)를 override.
     *
     * @param beanName - the name of the candidate bean
     * @return 빈에 해당하는 URL list
     */
    protected String[] determineUrlsForHandler(String beanName) {
        return remappingUrls(super.determineUrlsForHandler(beanName));
    }
}
```

인터셉터를 적용할 url들을 프로퍼티 urls에 선언하면 되며, Ant-style의 패턴 매칭이 지원된다. SimpleUrlAnnotationHandlerMapping은 선언된 url만을 Controller와 매핑처리한다. 따라서, 아래와 같이 선언된 DefaultAnnotationHandlerMapping과 같이 선언되어야 하며, 우선순위는 SimpleUrlAnnotationHandlerMapping이 높아야 한다.

```
<bean id="selectAnnotationMapper"
      class="egovframework.rte.ptl.mvc.handler.SimpleUrlAnnotationHandlerMapping"
      p:order="1">
    <property name="interceptors">
        <list>
            <ref local="authenticInterceptor"/>
        </list>
    </property>
    <property name="urls">
```

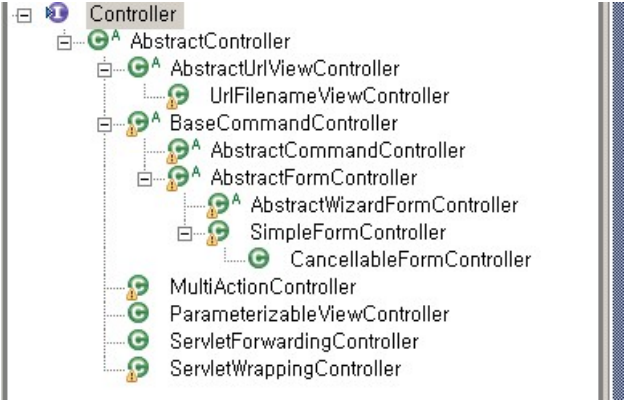
```
        <set>
            <value>/*Employee.do</value>
            <value>/employeeList.do</value>
        </set>
    </property>
</bean>
<bean id="annotationMapper"
      class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"
      p:order="2"/>
<bean id="authenticInterceptor" class="com.easycompany.interceptor.AuthenticInterceptor" />
```

## 참고자료

- The Spring Framework - Reference Documentation 2.5.6
- Spring Framework [API](#) Documentation 2.5.6

개요

DispatcherServlet은 HandlerMapping를 이용해서 해당 요청을 처리할 Controller를 결정한다. 이 Controller는 요청에 대해서 처리를 하고 데이터를 Model 객체에 반영한다. Spring MVC는 다양한 종류의 Controller를 제공하는데, 데이터 바인딩이나 폼 처리 또는 멀티 액션등의 편의 기능을 제공한다. 이 Controller들은 org.springframework.web.servlet.mvc.Controller 인터페이스를 구현한 클래스들이다.(@Controller는 예외다. 여기서는 @Controller에 대한 설명은 제외한다.) eclipse에서 인터페이스 Controller를 Hierarchy View에서 열어보면 아래와 같은 구조를 보여준다.



이중에 주요 Controller의 용도 및 특징을 표로 나타내보면 아래와 같다.

클래스	용도 및 특징
Controller	기본적인 Controller 인터페이스이다. Struts의 Action과 비교될 수 있다. Spring에서는 Controller 작성시에 직접 Controller 인터페이스를 구현하지 말고, 아래의 구현 클래스를 확장해서 작성할것을 권장한다.
AbstractController	웹 요청과 응답을 처리하는 기본적인 Controller이다. WebContentGenerator를 상속받기 때문에, HTTP 메소드(POST,GET) 지정, 세션 필수 여부등의 편의기능을 추가로 받는다.
AbstractCommandController	HttpServletRequest의 파라미터를 동적으로 데이터 객체(Command)에 바인딩 할 수 있다. 하지만 HTML 폼 처리시엔 SimpleFormController를 사용하라.
SimpleFormController	HTML 폼 처리시에 사용하는 Controller이다. AbstractCommandController처럼 HttpServletRequest의 파라미터와 Command 객체를 바인딩할뿐 아니라, 입력폼에 필요한 데이터를 채워 보여주거나(referenceData, formBackingObject), 일반적인 폼 처리 시나리오에 따른 view 분기(formView, successView) 등의 편의 기능을 제공한다.
MultiActionController	연관된 여러 액션을 한 Controller에서 처리할때 사용한다.
UrlFilenameViewController	Controller에서 처리 로직이 없이 바로 view로 이동하는 경우에 사용한다.

설명

AbstractController

단순히 요청을 처리하고 그 결과를 ModelAndView 객체에 반영하는 작업을 할 때는 AbstractController을 상속한 Controller를 구현하면 된다. 구현 Controller에서는 AbstractController의 추상 메소드인 handleRequestInternal을 구현하면 된다.

```
protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response) throws Exception;
```

Controller를 작성할때 인터페이스 Controller를 바로 구현하는 대신, AbstractController를 상속받아 구현하면 특정 HTTP 메소드(GET,POST)에 대한 필터링이나 세션 필수 체크등의 편의 기능을 제공받는다.

AbstractController의 작업 흐름은 아래와 같다.

- 1. DispatcherServlet에 의해 handleRequest()가 호출된다.
- 2. 특정 HTTP 메소드(GET,POST)에 대한 필터링을 수행한다.
- 3. 세션필수여부값이 true이면 HttpServletRequest 객체에서 세션을 꺼낸다.
- 4. cacheSeconds 프로퍼티에 설정된 값에 따라 캐시 헤더를 설정한다.
- 5. 추상 메소드인 handleRequestInternal()을 호출한다. AbstractController를 상속받은 구현 Controller 클래스를 작성했다면 구현된 handleRequestInternal() 메소드가 실행된다.

관련 프로퍼티를 정리하면 아래와 같다.

이름	기본값	설명
supportedMethods	GET,POST	Controller가 지원하는 HTTP 메소드 리스트(GET, POST and PUT)로 콤마(,)로 구분한다.

requireSession	false	Controller에서 요청 처리시에 session이 반드시 필요한지 여부이다. 값이 true인데 세션이 없다면 ServletException이 발생한다.
cacheSeconds	-1	응답의 캐시 헤더에 설정하는 시간값으로 단위는 초단위이다. 값이 0이면 캐시를 수행하지 않는 헤더를 갖게 되며, -1(기본값)이면 어떤 헤더도 생성하지 않으며, 양수값을 설정하면 설정한 값(초)만큼 내용을 캐시를 수행하는 헤더를 생성한다.
synchronizeOnSession	false	메소드 handleRequestInternal()을 호출할때 세션(HttpSession)에 동기화(synchronized)해서 호출할지 여부이다. 만일 세션이 없다면 아무 영향이 없다.

## 예제

사용자 인증처리를 위해 아이디와 패스워드를 입력받는 페이지가 아래와 같다고 하자.

아이디 :
패스워드:
로그인

```

<%@ page contentType="text/html; charset=UTF-8"%>
<html>
<head>
<title>Login Page</title>
<link type="text/css" rel="stylesheet" href="scripts/easycompany.css" />
</head>
<body>
<form action="/easycompany/loginProcess.do" method="post">
아이디 : <input type="text" name="id"> 패스워드: <input type="password" name="password"> <input type="submit" value="로그인">
</form>
</body>
</html>

```

로그인 처리를 하는 LoginController를 AbstractController를 확장해서 작성해 보자.  
먼저 아래와 같이 빈설정을 하고

```

<bean name="/loginProcess.do" class="com.easycompany.controller.hierarchy.LoginController"
p:loginService-ref="loginService"
p:supportedMethods="POST"/> <!--HTTP 메소드가 POST일때만 처리한다-->

```

메소드 handleRequestInternal()에 실제 구현 로직을 넣어 준다.

```

package com.easycompany.controller.hierarchy;
...
public class LoginController extends AbstractController{

    private LoginService loginService;

    public void setLoginService(LoginService loginService){
        this.loginService = loginService;
    }

    /**
     * AbstractController의 추상 메소드인 handleRequestInternal의 구현 메소드이다.
     * * 사용자로부터 아이디, 패스워드를 입력받아 인증 성공이면 세션 객체에 계정정보를 담고 사원정보리스트 페이지로 포워딩한다.
     * * 인증에 실패하면 로그인 페이지로 다시 리턴한다.
     */
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        String id = request.getParameter("id"); //아이디
        String password = request.getParameter("password"); //패스워드

        //로그인 인증을 처리한 후, 로그인 성공이면 Account 객체에 계정 관련정보를 리턴한다. 로그인 실패하면 null 리턴.
        Account account = (Account) loginService.authenticate(id,password);

        if (account != null) { //로그인 성공
            request.getSession().setAttribute("UserAccount", account); //계정정보를 세션에 저장.
            return new ModelAndView("redirect:/employeeList.do"); //사원리스트 페이지로 이동.
        } else { //실패
            return new ModelAndView("login"); //로그인페이지로 이동
        }
    }
}

```

## AbstractCommandController

AbstractCommandController는 요청 파라미터값을 커맨드(Command) 클래스의 필드값과 자동으로 바인딩할 때 사용된다. 커맨드 클래스는 일반적인 JavaBean이면 되는데, ActionForm 같이 프레임워크에 종속적인 구조의 폼 클래스를 사용해야 하는 Struts와의 차이점이라 할 수 있다. 파라미터와 커맨드 클래스의 데이터 바인딩은 일반적으로 알려진 JavaBeans 프로퍼티 표시법을 따른다. firstName 이란 이름의 파라미터가 있다면 커맨드 클래스의 setFirstName([value]) 메소드를 찾아 값을 바인딩한다. 파라미터 address.city 는 커맨드 클래스의 getAddress().setCity([value]) 메소드를 찾아 값을 바인딩한다. 이 기능은 HTML 폼 처리에 유용한 편의 기능이지만, 일반적으로 HTML 폼 처리에는 AbstractCommandController대신 SimpleFormController를 사용한다. AbstractCommandController을 상속받는 구현 Controller에서는 추상메소드 handle()을 구현하면 된다.

```

protected abstract ModelAndView handle(
    HttpServletRequest request, HttpServletResponse response, Object command, BindException errors) throws Exception;

```

## 예제

사원번호, 부서번호, 사원이름등의 검색 조건에 따라 사원리스트를 보여주는 페이지를 만들어 보자.

사원번호 : <input type="text"/>	부서번호 : <input type="text"/>	이름 : <input type="text"/>	검색
-----------------------------	-----------------------------	---------------------------	----

	사원번호	부서번호	이름	나이	이메일
	<u>1</u>	1200	김길동	28	kkd@easycompany.com
	<u>2</u>	1100	김길수	39	kks@easycompany.com
	<u>3</u>	1200	강감찬	17	kkc@easycompany.com

검색 조건을 담은 빈은 아래와 같다.

```
package com.easycompany.domain;
public class SearchCriteria {

    private String searchEid;
    private String searchDid;
    private String searchName;

    위의 변수들에 대한 set/get 함수들...
}
```

검색 조건을 화면으로 부터 입력받아 검색 조건 빈(bean)인 SearchCriteria에 담아서 서비스에 넘겨주고 리스트로 결과를 받아오는 EmployeeListController 작성해 보자.  
EmployeeListController를 AbstractController를 이용해 만든다면, 파라미터의 값을 꺼내고 값을 객체에 담는 코드를 직접 작성해야 한다.

```
package com.easycompany.controller.hierarchy;
...
public class EmployeeListController extends AbstractController{
    ...
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        //request 객체의 파라미터값을 꺼내서
        String searchEid = request.getParameter("searchEid"); //사원번호
        String searchDid = request.getParameter("searchDid"); //부서번호
        String searchName = request.getParameter("searchName"); //사원이름
        //객체에 저장한다.
        SearchCriteria searchCriteria = new SearchCriteria();
        searchCriteria.setSearchEid(searchEid);
        searchCriteria.setSearchDid(searchDid);
        searchCriteria.setSearchName(searchName);

        List<Employee> employeeList = employeeService.getAllEmployees(searchCriteria);

        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("employeeList", employeeList);
        modelAndView.addObject("searchCriteria", searchCriteria);
        modelAndView.setViewName("employeeList");

        return modelAndView;
    }
}
```

매퍼링해야할 파라미터가 많다면 상당히 번거로운 작업이고, 단순 작업 코드의 라인이 길어져 코드의 가독성도 떨어진다.  
EmployeeListController를 AbstractCommandController를 상속받아 구현해 보면 아래와 같이 변경될 것이다.

```
package com.easycompany.controller.hierarchy;
...
public class EmployeeListController extends AbstractCommandController{
    public EmployeeListController(){
        //Command 객체에 대한 선언, 빈 설정 파일에 Command 객체에 대한 선언이 있다면 이 코드는 필요없다.
        setCommandClass(SearchCriteria.class);
        setCommandName("searchCriteria");
    }
    ...
    @Override
    protected ModelAndView handle(HttpServletRequest request,
        HttpServletResponse response, Object command, BindException errors)
        throws Exception {
        //이미 파라미터와 Command 객체의 바인딩이 되어 있다.
        SearchCriteria searchCriteria = (SearchCriteria)command;

        List<Employee> employeeList = employeeService.getAllEmployees(searchCriteria);

        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("employeeList", employeeList);
        modelAndView.addObject("searchCriteria", searchCriteria);
        modelAndView.setViewName("employeeList");

        return modelAndView;
    }
}
```

커맨드 클래스 설정은 setCommandClass, setCommandName 메소드 대신에 빈 설정 파일에 정의할 수 있다.

```
<bean id="employeeListController" class="com.easycompany.controller.hierarchy.EmployeeListController"
    p:employeeService-ref="employeeService"
    p:commandName="searchCriteria"
    p:commandClass="com.easycompany.domain.SearchCriteria"/>
```

이 데이터 바인딩은 spring의 폼 태그 <form:form> 와 함께 쓰면 더욱 편리하게 사용할 수 있다.  
폼 태그의 변수 **commandName**은 커맨드 클래스의 이름과 일치해야 한다.  
커맨드 객체에 사원번호, 부서번호등의 값이 들어 있다면 JSP는 커맨드 객체의 필드값과 폼필드값을 자동으로 바인딩하여 보여주게 된다.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
...
<form:form commandName="searchCriteria" action="/easycompany/employeeList.do">
<table width="50%" border="1">
    <tr>
        <td>사원번호 : <form:input path="searchEid"/></td>
        <td>부서번호 : <form:input path="searchDid"/></td>
        <td>이름 : <form:input path="searchName"/></td>
```



```

<td><input type="submit" value="검색" onclick="this.disabled=true,this.form.submit();" /></td>
</tr>
</table>
</form:form>
<table>
<tr>
<th></th>
<th>사원번호</th>
<th>부서번호</th>
<th>이름</th>
<th>나이</th>
<th>이메일</th>
</tr>
<c:forEach items="${employeeList}" var="empinfo">
<tr>
<td></td>
<td><a href="javascript:getEmployeeInfo('${empinfo.employeeid}')">${empinfo.employeeid}</a></td>
<td>${empinfo.departmentid}</td>
<td>${empinfo.name}</td>
<td>${empinfo.age}</td>
<td>${empinfo.email}</td>
</tr>
</c:forEach>
</table>
...

```

## SimpleFormController

HTML 폼을 보여주거나 전송(submission)하는 등에 폼처리를 다루는 Controller를 작성 한다면, SimpleFormController를 상속한 Controller를 구현 하면 된다.

SimpleFormController는 상위 클래스인 BaseCommandController와 AbstractFormController가 제공하는 파라미터와 커맨드(폼) 클래스의 데이터 바인딩, 세션 폼 모드, 입력값 검증(validation), 입력폼에 초기 데이터 세팅등의 편의 기능을 그대로 사용하면서 폼 전송시에 결과에 따른 화면 분기(formView, successView)등 편의 기능을 추가로 제공한다. 폼을 보여주고 전송하는 Controller를 각각 만들지 않고 하나로 만들 수 있다.

SimpleFormController의 작업 흐름을 보려면 상위 클래스인 AbstractFormController의 handleRequestInternal() 메소드를 참고하면 되는데, 작업 흐름은 아래와 같다.

### GET 방식 호출

1. Controller가 폼 페이지에 대한 요청을 받는다. (GET 방식 호출)
2. formBackingObject() 메소드는 기본적으로 요청에 대한 커맨드 객체를 생성해서 반환한다. 일반적으로는 formBackingObject를 오버라이딩 해서 GET 방식 호출시에 폼에 채우고자 하는 기본값을 가져오는 로직을 넣는다.
3. initBinder() 메소드가 실행되는데, 커맨드 클래스의 특정 필드에 대해서 커스텀 에디터를 사용할수 있도록 한다.
4. 프로퍼티 bindOnNewForm이 true이면, 초기 요청 파라미터들을 가지고 새로운 커맨드 객체에 값을 채우는데 ServletRequestDataBinder가 적용되며, onBindOnNewForm() 콜백 메소드가 호출된다.
5. showForm() 메소드는 referenceData()를 호출해서 폼 페이지에서 보여주고자 하는 참조 데이터(주로 선택박스, 체크박스 같은 유형)를 ModelAndView에 저장한다.
6. formView에서는 Model 데이터를 바탕으로 폼에 필요한 데이터를 채워서 표시한다.

### POST 방식 호출

1. 사용자가 폼데이터를 전송한다(submit). (POST 방식 호출)
2. getCommand() 메소드가 커맨드 객체를 반환하는데, 만일 sessionForm이 false이면 formBackingObject() 메소드를 호출해서 커맨드 클래스의 인스턴스를 반환하고, sessionForm이 true이면 세션에서 커맨드 객체를 꺼내서 반환한다. 만일 해당 객체를 세션에서 찾지 못하면 handleInvalidSubmit() 메소드를 호출한다. handleInvalidSubmit()는 새로운 커맨드 객체를 생성하고 다시 폼 전송을 시도한다.
3. 요청 파라미터들로 커맨드 객체를 채우기 위해 ServletRequestDataBinder가 사용된다.
4. onBind() 메소드를 호출한다. 유효성검사(validation) 수행전에 필요한 작업들을 수행할 수 있다.
5. 프로퍼티 validateOnBinding값이 true이면, 등록된 Validator가 호출된다. Validator는 커맨드 객체의 필드값에 대한 유효성을 검사한다.
6. onBindAndValidate() 메소드를 호출한다. 여기서 바인딩과 유효성 검사 이후 사용자 정의 작업을 수행할 수 있다.
7. processFormSubmission() 메소드에서 전송을 처리한다. 유효성검사 결과 에러가 있는 경우 showForm() 메소드가 호출되어 다시 formView로 이동하고 에러가 없으면 onSubmit() 메소드가 수행되면서 폼 제출이 된다.
8. 폼 제출이 성공하면 successView로 이동한다.

관련 프로퍼티는 아래와 같다.

이름	기본값	설명	해당클래스
commandName	command	커맨드 클래스의 이름(별칭)	BaseCommandController
commandClass	null	요청 파라미터와 데이터 바인딩하게 될 커맨드 클래스	BaseCommandController
validators	null	커맨드 객체의 데이터 유효성검사를 수행할 Validator 빈의 배열	BaseCommandController
validator	null	Validator가 한개인 경우 사용.	BaseCommandController
validateOnBinding	true	유효성검사를 수행할지 여부. true이면 수행한다.	BaseCommandController
bindOnNewForm	false	새로운 폼이 보여지는 시점에서 데이터 바인딩을 할지 여부.	AbstractFormController
sessionForm	false	커맨드 객체를 세션에 저장하여 사용할지 여부.	AbstractFormController
formView	null	사용자가 입력하는 폼페이지나 유효성검사에 에러났을 경우에 사용하는 뷰를 표시한다.	SimpleFormController
successView	null	폼 제출이 성공했을때 보여줄 뷰를 표시한다.	SimpleFormController

## 예제

부서 정보 수정 페이지(/easycompany/webapp/WEB-INF/jsp/modifydepartment.jsp)를 만들어 보자.  
이 페이지의 처리 흐름은 아래와 같다.

1. 사용자에게 입력폼페이지를 보여주되 기존의 부서정보를 채워서 보여준다. → 위에서 언급한 **GET 방식 호출**시의 프로세스에 따라 처리 된다.
2. 사용자는 수정할 내용을 수정한 후에 저장 버튼을 누른다. → **POST 방식 호출**시의 프로세스에 따라 처리 된다.
  - I. 저장에 실패하거나 입력값 검증에 문제가 있으면 다시 초기 입력 폼페이지로 이동한다.
  - II. 저장에 성공하면 부서 정보 리스트페이지로 이동한다.

부서번호	1100
부서이름	<input type="text" value="회식메뉴혁신팀"/>
상위부서	<input type="text" value="경영기획실"/>
설명	<div>         매번 삼겹살 지겹지 않으세요? 저희 회식메뉴 혁신팀에서는 지속적인 즐거운 회사문화 조성을 위해...       </div>

```
<form:form commandName="department">
<table>
  <tr>
    <th>부서번호</th>
    <td><c:out value="${department.deptid}" /></td>
  </tr>
  <tr>
    <th>부서이름</th>
    <td><form:input path="deptname" size="20" /></td>
  </tr>
  <tr>
    <th>상위부서</th>
    <td>
      <form:select path="superdeptid">
        <option value="">상위부서를 선택하세요.</option>
        <form:options items="${deptInfoOneDepthCategory}" />
      </form:select>
    </td>
  </tr>
  <tr>
    <th>설명</th>
    <td><form:textarea path="description" rows="10" cols="40" /></td>
  </tr>
</table>
<table width="80%" border="1">
  <tr>
    <td>
      <input type="submit" value="저장" />
      <input type="button" value="리스트페이지" onclick="location.href='/easycompany/departmentList.do?depth=1'" />
    </td>
  </tr>
</table>
</form:form>
```

처리를 담당할 UpdateDepartmentController를 빈 설정 파일(xxx-servlet.xml)에 아래와 같이 등록한다.

```
<bean id="updateDepartmentController" class="com.easycompany.controller.hierarchy.UpdateDepartmentController"
  p:departmentService-ref="departmentService"
  p:commandName="department"
  p:commandClass="com.easycompany.domain.Department"
  p:formView="modifydepartment"
  p:successView="redirect:/departmentList.do?depth=1"/>
```

## formBackingObject 메소드

```
protected Object formBackingObject(HttpServletRequest request) throws Exception
```

일반적으로 수정 폼페이지는 기존의 데이터를 폼에 채우고 사용자가 원하는 부분을 수정한 후에 전송(submit)하는데, 기존 데이터를 불러와 폼에 채우는 역할을 formBackingObject 메소드가 담당한다. formBackingObject 메소드는 커맨드 객체를 생성해서 리턴하는데, 필요에 따라 이 메소드를 오버라이드해서 필요한 데이터를 커맨드 객체에 채워 주면 된다.

부서 정보 수정 페이지에서 기존의 부서 정보를 채워서 보여 주는 부분을 먼저 작성해 보자. 오버라이드한 메소드에HTTP GET 메소드 요청이면 파라미터에 있는 부서 아이디로 부서 정보 테이블을 조회해서 결과를 객체 Department에 담아 반환하는 로직을 추가해 보자.

```
package com.easycompany.controller.hierarchy;
...
public class UpdateDepartmentController extends SimpleFormController{
  private DepartmentService departmentService;
```



```

    public void setDepartmentService(DepartmentService departmentService) {
        this.departmentService = departmentService;
    }

    @Override
    protected Object formBackingObject(HttpServletRequest request) throws Exception {
        if(!isFormSubmission(request)) { // GET 요청이면
            String deptid = request.getParameter("deptid");
            Department department = departmentService.getDepartmentInfoById(deptid); //부서 아이디로 DB를 조회한 결과가 커맨드 객체 반영.
        } else { // POST 요청이면
            //AbstractFormController의 formBackingObject를 호출하면 요청객체의 파라미터와 설정된 커맨드 객체간에 기본적인 데이터 바인딩이 이루어진다.
            return super.formBackingObject(request);
        }
        ...
    }
}

```

## referenceData 메소드

```

protected Map referenceData(HttpServletRequest request) throws Exception
protected Map referenceData(HttpServletRequest request, Object command, Errors errors) throws Exception

```

폼 페이지에 미리 보여 주어야 할 데이터중에 커맨드 객체에 포함하기 어려운 경우가 있다.

부서 정보 수정 페이지에 보면 상위 부서 정보가 선택박스로 되어 있는데, 커맨드 객체인 부서(Department)객체에는 해당 부서의 상위 부서번호만 있을 뿐 이 회사에 어떤 상위부서들이 있는 지에 대한 정보는 없다.

커맨드 객체에 없지만 페이지에 필요한 이런 참조성 데이터들을 사용하기 위해서 referenceData 메소드를 사용하면 된다.

referenceData 메소드는 맵 객체를 반환하는데 이 맵은 모델 객체에 담겨 ModelAndView에 저장된다.

우리는 그저 referenceData 메소드를 오버라이드해서 참조성 데이터를 맵 객체에 넣어 주면 된다.

```

package com.easycompany.controller.hierarchy;
...
public class UpdateDepartmentController extends SimpleFormController{

    private DepartmentService departmentService;

    public void setDepartmentService(DepartmentService departmentService) {
        this.departmentService = departmentService;
    }

    @Override
    protected Map referenceData(HttpServletRequest request, Object command, Errors errors) throws Exception{
        Map param = new HashMap();
        param.put("depth", "1");
        Map referenceMap = new HashMap();
        referenceMap.put("deptInfoOneDepthCategory", departmentService.getDepartmentIdNameList(param)); //상위부서정보를 가져와서 Map에 담는다.
        return referenceMap;
    }
    ...
}

```

부서 정보 수정페이지를 열었을때 참조 데이터로 가져온 상위 부서 정보 리스트 중에 해당 부서의 상위 부서값이 선택 박스에서 기본으로 선택("selected")되서 보여져야 한다면,

참조데이터 중에 커맨드 객체와 일치하는 값을 일일이 조건문을 사용해서 비교하는 로직을 넣어주는 중노동을 해야 하나,

스프링 폼태그를 사용하면 간단하게 해결된다.

referenceData 메소드가 반환한 상위 부서 정보 맵 데이터중에 해당 부서의 상위부서와 일치하는게 있으면 <form:options>은 해당 옵션을 선택("selected")으로 프린트한다.

```

<form:select path="superdeptid">
    <option value="">상위부서를 선택하세요.</option>
    <form:options items="${deptInfoOneDepthCategory}" />
</form:select>

```

## onSubmit 메소드

```

protected ModelAndView onSubmit(Object command) throws Exception
protected ModelAndView onSubmit(Object command, BindException errors) throws Exception
protected ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response, Object command, BindException errors) throws Exception

```

onSubmit() 메소드를 오버라이드한 메소드를 만들어서 폼의 내용을 전송을 하는 로직을 넣어보자.

커맨드 객체인 부서 정보 객체(Department)의 내용을 DB에 반영하고, 처리가 성공했으면 successView로 이동하고 실패하면 showForm 메소드를 호출한다.

이 showForm 메소드는 폼페이지를 다시 보여주기 위해 필요한 데이터와 에러정보를 ModelAndView에 넣고 formView로 이동한다.

```

package com.easycompany.controller.hierarchy;
...
public class UpdateDepartmentController extends SimpleFormController{

    private DepartmentService departmentService;

    public void setDepartmentService(DepartmentService departmentService) {
        this.departmentService = departmentService;
    }

    @Override
    protected ModelAndView onSubmit(HttpServletRequest request,
        HttpServletResponse response, Object command, BindException errors) throws Exception{

        Department department = (Department) command;

        try {
            departmentService.updateDepartment(department);
        } catch (Exception ex) {
            return showForm(request, response, errors);
        }

        return new ModelAndView(getSuccessView(), "department", department);
    }
    ...
}

```

지금까지 살펴본 Controller들은 하나의 액션에 하나의 Controller를 만드는 방식이다. (SimpleFormController가 하나의 url에 대해 GET, POST 메소드에 따라 분기 처리를 하기는 하지만.)  
연관있는 여러 액션들을 하나의 Controller에 모으고 싶다면 MultiActionController를 상속받아 Controller를 작성하면 되는데 하나의 액션을 하나의 메소드로 작성한다.  
그 메소드는 아래의 형식을 갖는다.

```
public (ModelAndView | Map | String | void) actionName(HttpServletRequest request, HttpServletResponse response);
```

그러면 어떤 액션(url)을 어떤 메소드가 처리 할지를 결정하는 일이 필요한데, 이때 도움을 주는 인터페이스가 MethodNameResolver이다. MethodNameResolver는 요청에 대해서 어떤 메소드가 처리 할지 메소드 이름을 반환하는데, 스프링은 다음과 같은 3가지 MethodNameResolver 구현 클래스를 제공한다.

- ParameterMethodNameResolver : 특정 파라미터에 메소드 이름을 준다. 기본 파라미터는 action 이다.
- InternalPathMethodNameResolver : URL 경로의 마지막 부분중에 확장자를 제외한 부분이 메소드 이름이 된다.
- PropertiesMethodNameResolver : mapping 프로퍼티에 URL과 메소드 이름을 설정한다.

## 예제

상위 부서 리스트를 가져오는 액션과 특정 상위 부서에 속한 하위 부서 리스트를 가져 오는 액션을 처리하는 DepartmentListController를 작성해 보자.

```
package com.easycompany.controller.hierarchy;
...
public class DepartmentListController extends MultiActionController {

    //상위 부서 리스트를 가져 온다.
    public ModelAndView departmentList(HttpServletRequest request, HttpServletResponse response){

        String depth = request.getParameter("depth");

        Map paramMap = new HashMap();
        paramMap.put("depth", depth);

        List<Department> departmentlist = departmentService.getDepartmentList(paramMap);

        ModelAndView mav = new ModelAndView("departmentlist");
        mav.addObject("departmentlist", departmentlist);
        return mav;
    }

    //특정 상위 부서에 속한 하위 부서 리스트를 가져 온다.
    public ModelAndView subDepartmentList(HttpServletRequest request, HttpServletResponse response){

        String superdeptid = request.getParameter("superdeptid");
        String depth = request.getParameter("depth");

        Map paramMap = new HashMap();
        paramMap.put("depth", depth);
        paramMap.put("superdeptid", superdeptid);

        List<Department> departmentlist = departmentService.getDepartmentList(paramMap);

        ModelAndView mav = new ModelAndView("departmentsublist");
        mav.addObject("departmentlist", departmentlist);
        return mav;
    }
}
```

### ParameterMethodNameResolver를 사용한 경우

```
<bean name="/departmentList.do" class="com.easycompany.controller.hierarchy.DepartmentListController"
      p:departmentService-ref="departmentService"
      p:methodNameResolver-ref="paramResolver"/> <!-- ParameterMethodNameResolver를 MethodNameResolver로 사용-->

<bean id="paramResolver"
      class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver"
      p:paramName="method"/> <!-- 파라미터이름은 "method"-->
```

ParameterMethodNameResolver는 어떤 메소드를 호출할지에 대한 정보를 파라미터에 지정하는데, 파라미터 이름은 프로퍼티 "paramName"의 값이다.

- /easycompany/departmentList.do?depth=1&method=departmentList → departmentList()
- /easycompany/departmentList.do?superdeptid=1000&depth=2&method=subDepartmentList → subDepartmentList()

### InternalPathMethodNameResolver를 사용한 경우

```
<bean id="departmentController" class="com.easycompany.controller.hierarchy.DepartmentListController"
      p:departmentService-ref="departmentService"
      p:methodNameResolver-ref="pathResolver"/> <!-- InternalPathMethodNameResolver를 MethodNameResolver로 사용-->

<bean id="pathResolver"
      class="org.springframework.web.servlet.mvc.multiaction.InternalPathMethodNameResolver"/>
```

InternalPathMethodNameResolver를 사용하면 URL /abc/foo.do로 요청이 들어올때 public ModelAndView foo(HttpServletRequest request, HttpServletResponse response) 메소드로 매핑된다.

예제에서 보면, URL과 메소드간에 매핑이 아래와 같이 이루어 진다.

- /easycompany/departmentList.do?depth=1 → departmentList()
- /easycompany/subDepartmentList.do?superdeptid=1000&depth=2 → subDepartmentList()

### PropertiesMethodNameResolver를 사용한 경우

```
<bean id="departmentController" class="com.easycompany.controller.hierarchy.DepartmentListController"
      p:departmentService-ref="departmentService"
      p:methodNameResolver-ref="propResolver"/> <!-- InternalPathMethodNameResolver를 MethodNameResolver로 사용-->

<bean id="propResolver"
      class="org.springframework.web.servlet.mvc.multiaction.PropertiesMethodNameResolver">
    <property name="mappings">
```

```

        <props>
        <prop key="/departmentList.do">departmentList</prop>
        <prop key="/subDepartmentList.do">subDepartmentList</prop>
        </props>
    </property>
</bean>

```

PropertiesMethodNameResolver는 URL과 메소드의 매핑 관계를 "mappings" 프로퍼티에 명시해준다.

## UrlFilenameViewController

UrlFilenameViewController는 Controller에서 처리 로직이 없이 바로 view로 이동하는 경우에 사용하는 Controller이다. DispatcherServlet을 거쳐야 하지만, html 위주의 static한 페이지를 보여줄때 사용한다. 아래와 같이 URL path가 곧 뷰이름이 되며, prefix와 suffix를 지정할수도 있다.

```

"/index" -> "index"
"/index.html" -> "index"
"/index.html" + prefix "pre " and suffix "_suf" -> "pre_index_suf"
"/products/view.html" -> "pRoduCts/view"

```

Controller를 따로 만들 필요가 없으며, 아래와 같이 빈 설정 파일에서 url과 UrlFilenameViewController를 매핑만 해주면 된다.

```

<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/login.do">urlFilenameViewController</prop>
            <prop key="/validator.do">urlFilenameViewController</prop>
        </props>
    </property>
</bean>
<bean id="urlFilenameViewController" class="org.springframework.web.servlet.mvc.UrlFilenameViewController" />


```

InternalResourceViewResolver가 아래와 같이 선언되어 있다면,

```

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

```

URL  http://localhost:8080/easycompany/login.do로 요청이 들어올때, /easycompany/webapp/WEB-INF/jsp/login.jsp을 찾아서 보여준다.

## 참고자료

- The Spring Framework - Reference Documentation 2.5.6
- Spring Framework API Documentation 2.5.6

개요

스프링 프레임워크는 2.5 버전 부터 Java 5+ 이상이면 @Controller(Annotation-based Controller)를 개발할 수 있는 환경을 제공한다.  
인터페이스 Controller를 구현한 SimpleFormController, MultiActionController 같은 기존의 계층형(Hierarchy) Controller와의 주요 차이점 및 개선점은 아래와 같다.

- 1. 어노테이션을 이용한 설정 : XML 기반으로 설정하던 정보들을 어노테이션을 사용해서 정의한다.
- 2. 유연해진 메소드 시그니처 : Controller 메소드의 파라미터와 리턴 타입을 좀 더 다양하게 필요에 따라 선택할 수 있다.
- 3. POJO-Style의 Controller : Controller 개발시에 특정 인터페이스를 구현 하거나 특정 클래스를 상속해야할 필요가 없다.  
하지만, 폼 처리, 다중 액션등 기존의 계층형 Controller가 제공하던 기능들을 여전히 쉽게 구현할 수 있다.

계층형 Controller로 작성된 폼 처리를 @Controller로 구현하는 예도 설명한다.  
예제 코드 easycompany의 Controller는 동일한 기능(또한 공통의 Service, DAO, JSP를 사용)을 계층형 Controller와 @Controller로 각각 작성했다.

- 계층형 Controller - 패키지 com.easycompany.controller.hierarchy
- @Controller - 패키지 com.easycompany.controller.annotation

설명

어노테이션을 이용한 설정

계층형 Controller들을 사용하면 여러 정보들(요청과 Controller의 매핑 설정 등)을 XML 설정 파일에 명시 해줘야 하는데, 복잡할 뿐 아니라 설정 파일과 코드 사이를 빈번히 이동 해야하는 부담과 번거로움이 될 수 있다.  
@MVC는 Controller 코드안에 어노테이션으로 설정함으로써 좀 더 편리하게 MVC 프로그래밍을 할 수 있도록 했다.  
@MVC에서 사용하는 주요 어노테이션은 아래와 같다.

이름	설명
@Controller	해당 클래스가 Controller임을 나타내기 위한 어노테이션
@RequestMapping	요청에 대해 어떤 Controller, 어떤 메소드가 처리할지를 맵핑하기 위한 어노테이션
@RequestParam	Controller 메소드의 파라미터와 웹요청 파라미터와 맵핑하기 위한 어노테이션
@ModelAttribute	Controller 메소드의 파라미터나 리턴값을 Model 객체와 바인딩하기 위한 어노테이션
@SessionAttributes	Model 객체를 세션에 저장하고 사용하기 위한 어노테이션

@Controller

@MVC에서 Controller를 만들기 위해서는 작성한 클래스에 @Controller를 붙여주면 된다. 특정 클래스를 구현하거나 상속할 필요가 없다.

```
package com.easycompany.controller.annotation;  
  
@Controller  
public class LoginController {  
    ...  
}
```

앞서 DefaultAnnotationHandlerMapping에서 언급한 대로 <context:component-scan> 태그를 이용해 @Controller들이 있는 패키지를 선언해 주면 된다.  
@Controller만 스캔 한다면 include, exclude 등의 필터를 사용하라.

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"  
        xsi:schemaLocation="http://www.springframework.org/schema/beans  
            http://www.springframework.org/schema/beans/spring-beans-2.5.xsd  
            http://www.springframework.org/schema/context  
            http://www.springframework.org/schema/context/spring-context-2.5.xsd">  
  
    <context:component-scan base-package="com.easycompany.controller.annotation" />  
  
</beans>
```

## @RequestMapping

@RequestMapping은 요청에 대해 어떤 Controller, 어떤 메소드가 처리할지를 맵핑하기 위한 어노테이션이다.  
@RequestMapping이 사용하는 속성은 아래와 같다.

이름	타입	설명
value	String[]	URL 값으로 맵핑 조건을 부여한다. @RequestMapping(value="/hello.do") 또는 @RequestMapping(value={"/hello.do", "/world.do"})와 같이 표기하며, 기본값이기 때문에 @RequestMapping("/hello.do")으로 표기할 수도 있다. "/myPath/*.do"와 같이 Ant-Style의 패턴매칭을 이용할 수도 있다.
method	RequestMethod[]	HTTP Request 메소드값을 맵핑 조건으로 부여한다. HTTP 요청 메소드값이 일치해야 맵핑이 이루어 지게 한다. @RequestMapping(method = RequestMethod.POST)같은 형식으로 표기한다. 사용 가능한 메소드는 GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE이다
params	String[]	HTTP Request 파라미터를 맵핑 조건으로 부여한다. params="myParam=myValue"이면 HTTP Request URL중에 myParam이라는 파라미터가 있어야 하고 값은 myValue이어야 맵핑한다. params="myParam"와 같이 파라미터 이름만으로 조건을 부여할 수도 있고, "!myParam"하면 myParam이라는 파라미터가 없는 요청 만을 맵핑한다. @RequestMapping(params={"myParam1=myValue", "myParam2", "!myParam3"})와 같이 조건을 주었다면, HTTP Request에는 파라미터 myParam1이 myValue값을 가지고 있고, myParam2 파라미터가 있어야 하고, myParam3라는 파라미터는 없어야 한다.

@RequestMapping은 클래스 단위(type level)나 메소드 단위(method level)로 설정할 수 있다.

### type level

/hello.do 요청이 오면 HelloController의 hello 메소드가 수행된다.

```
@Controller
@RequestMapping("/hello.do")
public class HelloController {

    @RequestMapping //type level에서 URL을 정의하고 Controller에 메소드가 하나만 있어도 요청 처리를 담당할 메소드 위에
    @RequestMapping 표기를 해야 제대로 맵핑이 된다.
    public String hello() {
        ...
    }
}
```

### method level

/hello.do 요청이 오면 hello 메소드,

/helloForm.do 요청은 GET 방식이면 helloGet 메소드, POST 방식이면 helloPost 메소드가 수행된다.

```
@Controller
public class HelloController {

    @RequestMapping(value="/hello.do")
    public String hello() {
        ...
    }

    @RequestMapping(value="/helloForm.do", method = RequestMethod.GET)
    public String helloGet() {
        ...
    }

    @RequestMapping(value="/helloForm.do", method = RequestMethod.POST)
    public String helloPost() {
        ...
    }
}
```

### type + method level

둘 다 설정할 수도 있는데, 이 경우엔 type level에 설정한 @RequestMapping의 value(URL)를 method level에서 재정의 할 수 없다.

/hello.do 요청시에 GET 방식이면 helloGet 메소드, POST 방식이면 helloPost 메소드가 수행된다.

```
@Controller
@RequestMapping("/hello.do")
public class HelloController {

    @RequestMapping(method = RequestMethod.GET)
    public String helloGet() {
        ...
    }

    @RequestMapping(method = RequestMethod.POST)
    public String helloPost() {
        ...
    }
}
```

AbstractController 상속받아 구현한 예제 코드 LoginController를 어노테이션 기반의 Controller로 구현해 보겠다. 기존의 LoginController는 URL /loginProcess.do로 오는 요청의 HTTP 메소드가 POST일때 handleRequestInternal 메소드가 실행되는 Controller였는데, 다음과 같이 구현할 수 있겠다.

```
package com.easycompany.controller.annotation;
...
@Controller
public class LoginController {

    @Autowired
    private LoginService loginService;

    @RequestMapping(value = "/loginProcess.do", method = RequestMethod.POST)
    public String login(HttpServletRequest request) {

        String id = request.getParameter("id");
        String password = request.getParameter("password");

        Account account = (Account) loginService.authenticate(id,password);

        if (account != null) {
            request.getSession().setAttribute("UserAccount", account);
            return "redirect:/employeeList.do";
        } else {
            return "login";
        }
    }
}
```

위 예제 코드에서 서비스 클래스를 호출하기 위해서 @Autowired가 사용되었는데 자세한 내용은 여기를 참

## @RequestParam

@RequestParam은 Controller 메소드의 파라미터와 웹요청 파라미터와 맵핑하기 위한 어노테이션이다. 관련 속성은 아래와 같다.

이름	타입	설명
value	String	파라미터 이름
required	boolean	해당 파라미터가 반드시 필수 인지 여부. 기본값은 true이다.

아래 코드와 같은 방법으로 사용되는데,  
해당 파라미터가 Request 객체 안에 없을때 그냥 null값을 바인드 하고 싶다면, pageNo 파라미터 처럼 required=false 로 명시해야 한다.  
name 파라미터는 required가 true이므로, 만일 name 파라미터가 null이면 org.springframework.web.bind.MissingServletRequestParameterException이 발생한다.

```
@Controller
public class HelloController {

    @RequestMapping("/hello.do")
    public String hello(@RequestParam("name") String name, //required 조건이 없으면 기본값은 true, 즉 필수 파라미터 이
다. 파라미터 pageNo가 존재하지 않으면 Exception 발생.
        @RequestParam(value="pageNo", required=false) String pageNo){ //파라미터 pageNo가 존재하지
않으면 String pageNo는 null.
        ...
    }
}
```

위에서 작성한 LoginController의 login 메소드를 보면 파라미터 아이디와 패스워드를 Http Request 객체에서 getParameter 메소드를 이용해 구하는데,  
@RequestParam을 사용하면 아래와 같이 변경할수 있다.

```
package com.easycompany.controller.annotation;
...
@Controller
public class LoginController {

    @Autowired
    private LoginService loginService;

    @RequestMapping(value = "/loginProcess.do", method = RequestMethod.POST)
    public String login(
        HttpServletRequest request,
        @RequestParam("id") String id,
        @RequestParam("password") String password) {

        Account account = (Account) loginService.authenticate(id,password);

        if (account != null) {
            request.getSession().setAttribute("UserAccount", account);
            return "redirect:/employeeList.do";
        } else {
            return "login";
        }
    }
}
```

## @ModelAttribute

@ModelAttribute의 속성은 아래와 같다.

이름	타입	설명
value	String	바인드하려는 Model 속성 이름

@ModelAttribute는 실제로 ModelMap.addAttribute와 같은 기능을 발휘하는데, Controller에서 2가지 방법으로 사용된다.

### 1. 메소드 리턴 데이터와 Model 속성(attribute)의 바인딩.

메소드에서 비즈니스 로직(DB 처리같은)을 처리한 후 결과 데이터를 ModelMap 객체에 저장하는 로직은 자

```
...
@RequestMapping(value = "/updateDepartment.do", method = RequestMethod.GET)
public String formBackingObject(@RequestParam("deptid") String deptid, ModelMap model) {
    Department department = departmentService.getDepartmentInfoById(deptid); //DB에서 부서정보 데이터를
    가져온다.
    model.addAttribute("department", department); //데이터를 모델 객체에 저장한다.
    return "modifyDepartment";
}
...
```

@ModelAttribute를 메소드에 선언하면 해당 메소드의 리턴 데이터가 ModelMap 객체에 저장된다. 위 코드를 아래와 같이 변경할 수 있는데, 사용자로부터 GET방식의 /updateDepartment.do 호출이 들어오면, formBackingObject 메소드가 실행 되기 전에 DefaultAnnotationHandlerMapping이 org.springframework.web.bind.annotation.support.HandlerMethodInvoker를 이용해서 (@ModelAttribute가 선언된) getEmployeeInfo를 실행하고, 결과를 ModelMap객체에 저장한다. 결과적으로 getEmployeeInfo 메소드는 ModelMap.addAttribute("department", departmentService.getDepartmentInfoById(...)) 작업을 하게 되는 것이다.

```
...
@RequestMapping(value = "/updateDepartment.do", method = RequestMethod.GET)
public String formBackingObject() {
    return "modifyDepartment";
}

@ModelAttribute("department")
public Department getEmployeeInfo(@RequestParam("deptid") String deptid) {
    return departmentService.getDepartmentInfoById(deptid); //DB에서 부서정보 데이터를 가져온다.
}
//또는
public @ModelAttribute("department") Department getDepartmentInfoById(@RequestParam("deptid") String deptid) {
    return departmentService.getDepartmentInfoById(deptid);
}
...
```

### 2. 메소드 파라미터와 Model 속성(attribute)의 바인딩.

@ModelAttribute는 ModelMap 객체의 특정 속성(attribute) 메소드의 파라미터와 바인딩 할때도 사용될 수 있다.

아래와 같이 메소드의 파라미터에 "@ModelAttribute("department") Department department" 선언하면 department에는 (Department)ModelMap.get("department") 값이 바인딩된다.

따라서, 아래와 같은 코드라면 formBackingObject 메소드 파라미터 department에는 getDepartmentInfo 메소드가 ModelMap 객체에 저장한 Department 데이터가 들어 있다.

```
...
@RequestMapping(value = "/updateDepartment.do", method = RequestMethod.GET)
public String formBackingObject(@ModelAttribute("department") Department department) { //department에는
getDepartmentInfo에서 구해온 데이터들이 들어가 있다.
    System.out.println(employee.getEmployeeid());
    System.out.println(employee.getName());
    return "modifyDepartment";
}

@ModelAttribute("department")
public Department getDepartmentInfo(@RequestParam("deptid") String deptid) {
    return departmentService.getDepartmentInfoById(deptid); //DB에서 부서정보 데이터를 가져온다.
}
...
```

## @SessionAttributes

@SessionAttributes는 model attribute를 session에 저장, 유지할 때 사용하는 어노테이션이다. @SessionAttributes는 클래스 레벨(type level)에서 선언할 수 있다. 관련 속성은 아래와 같다.

이름	타입	설명
types	Class[]	session에 저장하려는 model attribute의 타입
value	String[]	session에 저장하려는 model attribute의 이름

## 유연해진 메소드 시그니처



@RequestMapping을 적용한 Controller의 메소드는 아래와 같은 메소드 파라미터와 리턴 타입을 사용할 수 있다. 특정 클래스를 확장하거나 인터페이스를 구현해야 하는 제약이 없기 때문에 계층형 Controller 비해 유연한 메소드 시그니처를 갖는다.

## @Controller의 메소드 파라미터

사용가능한 메소드 파라미터는 아래와 같다.

- **Servlet API** - ServletRequest, HttpServletRequest, HttpServletResponse, HttpSession 같은 요청, 응답, 세션 관련 Servlet API들.
- **WebRequest, NativeWebRequest** - org.springframework.web.context.request.WebRequest, org.springframework.web.context.request.NativeWebRequest
- **java.util.Locale**
- **java.io.InputStream / java.io.Reader**
- **java.io.OutputStream / java.io.Writer**
- **@RequestParam** - HTTP Request의 파라미터와 메소드의 argument를 바인딩하기 위해 사용하는 어노테이션.
- **java.util.Map / org.springframework.ui.Model / org.springframework.ui.ModelMap** - 뷰에 전달할 모델 데이터들.
- **Command/form 객체** - HTTP Request로 전달된 parameter를 바인딩한 커맨드 객체, @ModelAttribute을 사용하여 aliag " "
- **Errors, BindingResult** - org.springframework.validation.Errors / org.springframework.validation.BindingResult 유효성 검사 후 결과 데이터를 저장한 객체.
- **SessionStatus** - org.springframework.web.bind.support.SessionStatus 세션폼 처리시에 해당 세션을 제거하기 위해 사용"

메소드는 임의의 순서대로 파라미터를 사용할 수 있다. 단, BindingResult가 메소드의 argument로 사용될 때는 바인딩할 커맨드 객체가 바로 앞에 와야 한다.

```
public String updateEmployee (...,@ModelAttribute("employee") Employee employee,
                             BindingResult bindingResult,...) <!-- (O) -->

public String updateEmployee (...,BindingResult bindingResult,
                             @ModelAttribute("employee") Employee employee,...) <!-- (X) -->
```

## 이 외의 타입을 메소드 파라미터로 사용하려면?

스프링 프레임워크는 위에서 언급한 타입이 아닌 custom arguments도 메소드 파라미터로 사용할 수 있도록 org.springframework.web.bind.support.WebArgumentResolver라는 인터페이스를 제공한다.

## @Controller의 메소드 리턴 타입

사용가능한 메소드 리턴 타입은 아래와 같다.

- **ModelAndView** - 커맨드 객체와 @ModelAttribute이 적용된 메소드의 리턴 데이터가 담긴 Model 객체와 View 정보가 담겨 있다.

```
@RequestMapping(value = "/updateDepartment.do", method = RequestMethod.GET)
public ModelAndView formBackingObject(@RequestParam("deptid") String deptid) {
    Department department = departmentService.getDepartmentInfoById(deptid);
    ModelAndView mav = new ModelAndView("modifydepartment");
    mav.addObject("department", department);
    return mav;
}

또는

public ModelAndView formBackingObject(@RequestParam("deptid") String deptid, ModelMap model) {
    Department department = departmentService.getDepartmentInfoById(deptid);
    model.addAttribute("department", department);
    ModelAndView mav = new ModelAndView("modifydepartment");
    mav.addAllObjects(model);
    return mav;
}
```



- **Model(또는 ModelMap)** - 커맨드 객체와 @ModelAttribute이 적용된 메소드의 리턴 데이터가 Model 객체에 담겨 있다.

View 이름은 RequestToViewNameTranslator가 URL을 이용하여 결정한다. 인터페이스

RequestToViewNameTranslator의 구현클래스인 DefaultRequestToViewNameTranslator가 View 이름을 결정하는 방식은 아래와 같다.

```
http://localhost:8080/gamecast/display.html -> display
http://localhost:8080/gamecast/displayShoppingCart.html -> displayShoppingCart
http://localhost:8080/gamecast/admin/index.html -> admin/index
```

```
@RequestMapping(value = "/updateDepartment.do", method = RequestMethod.GET)
public Model formBackingObject(@RequestParam("deptid") String deptid, Model model) {
    Department department = departmentService.getDepartmentInfoById(deptid);
    model.addAttribute("department", department);
    return model;
}
```

또는

```
@RequestMapping(value = "/updateDepartment.do", method = RequestMethod.GET)
public Model formBackingObject(@RequestParam("deptid") String deptid) {
    Department department = departmentService.getDepartmentInfoById(deptid);
    Model model = new ExtendedModelMap();
    model.addAttribute("department", department);
    return model;
}
```

- **Map** - 커맨드 객체와 @ModelAttribute이 적용된 메소드의 리턴 데이터가 Map 객체에 담겨 있으며, View 이름은 역시 RequestToViewNameTranslator가 결정한다.

```
@RequestMapping(value = "/updateDepartment.do", method = RequestMethod.GET)
public Map formBackingObject(@RequestParam("deptid") String deptid) {
    Department department = departmentService.getDepartmentInfoById(deptid);
    Map model = new HashMap();
    model.put("department", department);
    return model;
}
```

또는

```
@RequestMapping(value = "/updateDepartment.do", method = RequestMethod.GET)
public Map formBackingObject(@RequestParam("deptid") String deptid, Map model) {
    Department department = departmentService.getDepartmentInfoById(deptid);
    model.put("department", department);
    return model;
}
```

- **String** - 리턴하는 String 값이 곧 View 이름이 된다. 커맨드 객체와 @ModelAttribute이 적용된 메소드의 리턴 데이터가 Model(또는 ModelMap)에 담겨 있다. 리턴할 Model(또는 ModelMap)객체가 해당 메소드의 argument에 선언되어 있어야 한다.

```
<!-- (O) -->
@RequestMapping(value = "/updateDepartment.do", method = RequestMethod.GET)
public String formBackingObject(@RequestParam("deptid") String deptid, ModelMap model) {
    Department department = departmentService.getDepartmentInfoById(deptid);
    model.addAttribute("department", department);
    return "modifydepartment";
}
```

```
<!-- (X) -->
```

```
@RequestMapping(value = "/updateDepartment.do", method = RequestMethod.GET)
```

```

public String formBackingObject (@RequestParam ("deptid") String deptid) {
    Department department = departmentService.getDepartmentInfoById (deptid);
    ModelMap model = new ModelMap ();
    model.addAttribute ("department", department);
    return "modifydepartment";
}

```

- **View** - View를 리턴한다. 커맨드 객체와 @ModelAttribute이 적용된 메소드의 리턴 데이터가 Model(또는 ModelMap)에 담겨 있다.
- **void** - 메소드가 ServletResponse / HttpServletResponse등을 사용해서 직접 응답을 처리하는 경우. View 이름은 RequestToViewNameTranslator가 결정한다.

## POJO-Style의 Controller

@MVC는 Controller 개발시에 특정 인터페이스를 구현 하거나 특정 클래스를 상속해야할 필요가 없다. Controller의 메소드에서 Servlet API를 반드시 참조하지 않아도 되며, 훨씬 유연해진 메소드 시그니처로 개발이 가능하다. 여기서는 SimpleFormController의 폼 처리 액션을 @Controller로 구현함으로써, POJO-Style에 가까워졌지만 기존의 계층형 Controller에서 제공하던 기능들을 여전히 구현할 수 있음을 보이고자 한다.

### FormController by SimpleFormController -> @Controller

앞서 SimpleFormController을 설명하면서 예제로 작성된 com.easycountry.controller.hierarchy.UpdateDepartmentController를 @ModelAttribute와 @RequestMapping을 이용해서 같은 기능을 @Controller로 작성해 보겠다.  
JSP 소스는 동일한 것을 사용한다.  
기존의 UpdateDepartmentController를 보면 3가지 메소드로 이루어졌다.

- **referenceData** - 입력폼에 필요한 참조데이터인 상위부서정보를 가져와서 Map 객체에 저장한다. 이후에 이 Map 객체는 스프링 내부 로직에 의해 ModelMap 객체에 저장된다.
- **formBackingObject** - GET 방식 호출일때 초기 입력폼에 들어갈 부서 데이터를 리턴한다. 이 데이터 역시 ModelMap 객체에 저장된다.
- **onSubmit** - POST 전송시에 호출되며 폼 전송을 처리한다.

```

package com.easycountry.controller.hierarchy;
...
public class UpdateDepartmentController extends SimpleFormController{
    private DepartmentService departmentService;

    public void setDepartmentService (DepartmentService departmentService){
        this.departmentService = departmentService;
    }

    //상위부서리스트(selectbox)는 부서정보클래스에 없으므로 , 상위부서리스트 데이터를 DB에서 구해서 별도의 참조데이터로 구성한
    다.
    @Override
    protected Map referenceData (HttpServletRequest request, Object command, Errors errors) throws Exception{
        Map referenceMap = new HashMap ();
        referenceMap.put ("deptInfoOneDepthCategory", departmentService.getDepartmentIdNameList ("1"));
        //상위부서정보를 가져와서 Map에 담는다.
        return referenceMap;
    }

    @Override
    protected Object formBackingObject (HttpServletRequest request) throws Exception {
        if (!isFormSubmission (request)) { // GET 요청이면
            String deptid = request.getParameter ("deptid");
            Department department = departmentService.getDepartmentInfoById (deptid); //부서 아이디로
            DB를 조회한 결과가 커맨드 객체 반영.
            return department;
        } else { // POST 요청이면
            //AbstractFormController의 formBackingObject을 호출하면 요청객체의 파라미터와 설정된 커맨드 객체간
            에 기본적인 데이터 바인딩이 이루어 진다.
            return super.formBackingObject (request);
        }
    }

    @Override
    protected ModelAndView onSubmit (HttpServletRequest request,
        HttpServletResponse response, Object command, BindException errors) throws Exception{
        Department department = (Department) command;

        try {
            departmentService.updateDepartment (department);
        } catch (Exception ex) {
            return showForm (request, response, errors);
        }

        return new ModelAndView (getSuccessView (), "department", department);
    }
}

```

```
}
@Controller로 작성된 com.easycompany.controller.annotation.UpdateDepartmentController은 3개의 메소드로 이루어져 있다.
```

계층형 Controller인 기존의 UpdateDepartmentController와는 달리 각 메소드는 Override 할 필요없기 때문에 메소드 이름은 자유롭게 지을 수 있다.

쉬운 비교를 위해 SimpleFormController과 동일한 메소드 이름을 선택했다.

- referenceData - 입력폼에 필요한 참조데이터인 상위부서정보를 가져와서 ModelMap에 저장한다.(by @ModelAttribute)
- formBackingObject - GET 방식 호출일때 처리를 담당한다. 초기 입력폼 구성을 위한 부서데이터를 가져와서 ModelMap에 저장한다.
- onSubmit - POST 전송시에 호출되며 폼 전송을 처리한다.

(POJO에 가까운) 프레임워크 코드들은 감춰졌고, 보다 직관적으로 비즈니스 내용을 표현할 수 있게 되었다고 생각한다.

```
package com.easycompany.controller.annotation;

...
@Controller
public class UpdateDepartmentController {

    @Autowired
    private DepartmentService departmentService;

    //상위부서리스트(selectbox)는 부서정보클래스에 없으므로 , 상위부서리스트 데이터를 DB에서 구해서 별도의 참조데이터로 구성한
    다.
    @ModelAttribute("deptInfoOneDepthCategory")
    public Map<String, String> referenceData() {
        return departmentService.getDepartmentIdNameList("1");
    }


    // 해당 부서번호의 부서정보 데이터를 불러와 입력폼을 채운다
    @RequestMapping(value = "/updateDepartment.do", method = RequestMethod.GET)
    public String formBackingObject(@RequestParam("deptid") String deptid, ModelMap model) {
        Department department = departmentService.getDepartmentInfoById(deptid);
        model.addAttribute("department", department); //form tag의 commandName은 이 attribute name과 일치해
    야 한다. <form:form commandName="department">.
        return "modifydepartment";
    }

    //사용자가 데이터 수정을 끝내고 저장 버튼을 누르면 수정 데이터로 저장을 담당하는 서비스(DB)를 호출한다.
    //저장이 성공하면 부서리스트 페이지로 이동하고 에러가 있으면 다시 입력폼페이지로 이동한다.
    @RequestMapping(value = "/updateDepartment.do", method = RequestMethod.POST)
    public String onSubmit(@ModelAttribute("department") Department department, BindingResult bindingResult)
    {

        //validation code
        new DepartmentValidator().validate(department, bindingResult);
        if(bindingResult.hasErrors()){
            return "modifydepartment";
        }

        try {
            departmentService.updateDepartment(department);
            return "redirect:/departmentList.do?depth=1";
        } catch (Exception e) {
            e.printStackTrace();
            return "modifydepartment";
        }
    }
}
```

## 참고자료

- The Spring Framework - Reference Documentation 2.5.6
- Spring Framework API Documentation 2.5.6
-  SpringSource Team Blog, Annotated Web MVC Controllers in Spring 2.5, Juergen Hoeller

## 개요

객체의 유효성 검증을 위해 스프링 프레임워크는 `org.springframework.validation.Validator`라는 인터페이스를 제공한다.

`Validator`는 특정 계층에 종속적인 구조가 아니라서, web이나 data-access등 어떤 계층의 객체라도 유효성 검증이 가능하게 한다.

Jakarta Commons Validator나 Valang 같은 외부 Validator들도 Spring 프레임워크에서 사용할 수 있다.

Spring Modules를 이용한 Jakarta Commons Validator 사용 방법에 대해서는 [Spring Framework에서 Commons Validator 사용](#) 을 참고하라.

- Validation
  - 개요
  - 설명
    - Validator 구현
    - 에러 메시지 설정
    - Controller에서 validation
    - JSP
    - TEST
  - 참고자료

## 설명

부서 정보를 수정하는 페이지에서 커맨드 객체인 부서 정보 클래스를 유효성 검증하는 코드를 작성해 보자.

부서 클래스인 `Department` 클래스는 아래와 같다.

```
package com.easycompany.domain;

public class Department {

    private String deptid;    //부서아이디
    private String deptname;  //부서이름
    private String superdeptid; //상위부서아이디
    private String superdeptname; //상위부서이름
    private String depth;    //부서레벨
    private String description; //부서설명

    //위 프로퍼티들의 setter/getter
}
```

## Validator 구현

인터페이스 `org.springframework.validation.Validator`의 메소드는 다음과 같다.

- `boolean supports(Class clazz)` : 주어진 객체(clazz)에 대해 `Validator` 지원 가능 여부
- `void validate(Object target, Errors errors)` : 주어진 객체(target)에 대해서 유효성 체크를 수행하고, 유효성 에러 발생시 주어진 `Errors`객체에 관련 정보가 저장

구현 `Validator` 클래스를 만들때는 위 두 메소드를 구현해야 한다.

`Department`를 유효성 검증 하기 위한 `DepartmentValidator`를 만들어 보자.

Validation 조건은 부서이름(deptname) 프로퍼티는 반드시 값이 존재해야 하며, 부서설명(description) 프로퍼티는 입력값의 길이가 10 이상이어야 한다.

```
package com.easycompany.validator;

import org.springframework.validation.Errors;
import org.springframework.validation.Validator;
import com.easycompany.domain.Department;

public class DepartmentValidator implements Validator {

    public boolean supports(Class clazz) {
        return Department.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        Department department = (Department) target;

        if (isEmptyOrWhitespace(department.getDeptname())) { //부서 이름 프로퍼티 값이 존재하는가?
            errors.rejectValue("deptname", "required");
        }

        if (department.getDescription() == null || department.getDescription().length() < 10) { //부서설명 프로퍼티는 값의 길
            errors.rejectValue("description", "lengthsize", new Object[]{10}, "description's length must be
larger than 10.");
        }

        public boolean isEmptyOrWhitespace(String value) {
            if (value == null || value.trim().length() == 0) {
                return true;
            } else {
                return false;
            }
        }
    }
}
```

위 코드에서 처럼 유효성 검증이 실패한 경우 `Errors` 인터페이스의 `rejectValue` 메소드를 실행하는데, `Errors` 인터페이스에 대한 자세한 설명은 아래와 같다.

`errors.rejectValue("deptname", "required");`

⇒ deptname 프로퍼티에 대해서 유효성 검증시 에러가 발생했고, 관련 메시지 key는 "required"란 의미이다.

`errors.rejectValue("description", "lengthsize", new Object[]{10}, "description's length must be larger than 10.");`  
⇒ description 프로퍼티에 대해서 유효성 검증시 에러가 발생했고, 관련 메시지 key는 "lengthsize" 이며 메시지에 전달될 argument는 10이며, 해당 메시지 key가 존재 하지 않으면 "description's length must be larger than 10."란 메시지를 사용한다는 의미이다.

스프링에서는 유효성 검증을 위한 ValidationUtils라는 유틸 클래스를 제공한다.  
부서 이름 프로퍼티(deptname) 값이 null또는 white space인지 체크하는 부분은 ValidationUtils의 rejectIfEmptyOrWhitespace 메소드를 사용해서 작성할 수 있다.

```
package com.easycompany.validator;

import org.springframework.validation.Errors;
import org.springframework.validation.Validator;
import org.springframework.validation.ValidationUtils;
import com.easycompany.domain.Department;

public class DepartmentValidator implements Validator {

    public boolean supports(Class clazz) {
        return Department.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        Department department = (Department) target;
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "deptname", "required");

        if (department.getDescription() == null || department.getDescription().length() < 10) { //부서설명 프로퍼티는 입력값의
            errors.rejectValue("description", "lengthsize", new Object[]{10}, "description's length must be
larger than 10.");
        }
    }
}
```

## 에러 메시지 설정

message 프로퍼티 파일에서 메시지 key인 "required", "lengthsize"에 대한 메시지 설정을 한다.

```
required=필수 입력값입니다.
lengthsize={0}자 이상 입력해야 합니다.
```

## Controller에서 validation

Controller에서 validation을 수행하는 코드를 적용해 보자.  
폼을 전송하는 순간에 유효성 검증을 하기 원한다면,  
com.easycompany.controller.annotation.UpdateDepartmentController 에서 onSubmit 메소드에 validation 코드를 추가한다.

```
package com.easycompany.controller.annotation;
...
import org.springframework.validation.BindingResult;
import com.easycompany.validator.DepartmentValidator;

@Controller
public class UpdateDepartmentController {

    //사용자가 데이터 수정을 끝내고 저장 버튼을 누르면 수정 데이터로 저장을 담당하는 서비스(DB)를 호출한다.
    //저장이 성공하면 부서리스트 페이지로 이동하고 에러가 있으면 다시 입력폼페이지로 이동한다.
    @RequestMapping(value = "/updateDepartment.do", method = RequestMethod.POST)
    public String onSubmit(@ModelAttribute("department") Department department, BindingResult bindingResult) {

        //validation code
        new DepartmentValidator().validate(department, bindingResult); //validation을 수행한다.
        if (bindingResult.hasErrors()) { //validation 에러가 있으면,
            return "modifydepartment"; //이 페이지로 이동.
        }

        try {
            departmentService.updateDepartment(department);
            return "redirect:/departmentList.do?depth=1";
        } catch (Exception e) {
            e.printStackTrace();
            return "modifydepartment";
        }
    }
}
```

## JSP

손쉬운 에러 메시지 표기를 위해 Spring 폼태그 <form:errors/>를 사용할 것을 권장한다.  
/easycompany/webapp/jsp/annotation/modifydepartment.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
...
<form:form commandName="department">
<table>
...
    <tr>
        <th>부서이름</th>
        <td><form:input path="deptname" size="20"/><form:errors path="deptname" /></td>
    </tr>
...
    <tr>
        <th>설명</th>
        <td><form:textarea path="description" rows="10" cols="40"/><form:errors path="description" /></td>
    </tr>
</table>
</form:form>
```

```
</table></tr>
</table>
</form:form>
...
```

## TEST

부서 이름값을 비우고, 부서설명 부분에 10자 이하로 입력한 후에 저장 버튼을 누르면, 다시 부서정보수정 페이지로 돌아와서 아래와 같이 에러 메시지가 출력될 것이다.

부서번호	1100
부서이름	<input type="text"/> 필수 입력값입니다.
상위부서	<input type="text" value="경영기획실"/>
설명	<div>매번   <input type="text"/></div> <div>10자 이상 입력해야 합니다.</div>

## 참고자료

- [Spring Framework API Documentation 2.5.6](#)

## 개요

🌐 화면처리: validation을 통해 검증방법을 알아보았다. 이전과는 다르게 JSR-303(Bean Validation) 스펙은 자동 검증 방식을 제공한다. @javax.validation.Valid 애노테이션을 사용하여 내부적으로(자동으로) 검증이 수행된다.

또한, 최근에 표준 스펙으로 인증받은 JSR-303 빈 검증방식을 이용하여 모델 오브젝트 필드에서 애노테이션을 이용해 검증을 진행할 수 있다.

- JSR-303
  - 개요
  - 설명
    - @Valid를 이용한 자동검증
    - JSR-303 빈 검증(bean validation) 기능
  - 참고자료

## 설명

### @Valid를 이용한 자동검증

기존의 검증 방식을 자동 검증 방식으로 변경하였으며, 방법은 컨트롤러 메소드의 @ModelAttribute 파라미터에 @Valid 애노테이션을 추가한다. 그러면 validate() 메소드를 실행하는 대신 바인딩 과정에서 자동으로 검증이 진행된다.

```
@Controller
public class ExampleController {

    @Autowired ExampleValidator validator;

    @InitBinder
    public void initBinder(WebDataBinder dataBinder){
        dataBinder.setValidator(this.validator);
    }

    @RequestMapping("/insertMember.do")
    public String insertMember(@ModelAttribute("memberVO") @Valid MemberVO memberVO, BindingResult
bindingResult, ..) {
        //..
    }
}
```

### JSR-303 빈 검증(bean validation) 기능

위의 방식은 기존의 검증방식을 자동 검증으로 변경한 방법이며 다음에 설명한 검증방법은 제약조건을 빈에 직접 설정하여 검증하는 방식이다.

먼저, 클래스패스에 의존 라이브러리를 추가해야 한다. 메이븐을 사용 중이라면 다음 의존 라이브러리를 프로젝트에 추가한다.

```
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
  <version>1.0.0.GA</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>4.0.0.GA</version>
</dependency>
```

다음은 JSR-303 제약조건 애노테이션이 적용된 모델오브젝트 예제이다.

```
public class MemberVO {

    @NotNull
    @Size(min = 1, max = 50, message="이름을 입력하세요.")
    private String name;

    @Pattern(regexp="^.+@.+\.[a-z]+$", message="이메일 형식이 잘못되었습니다.")
    private String email;

    //..
}
```

@NotNull은 빈문자열을 검증하지 못하기 때문에 @Size(min=1)을 사용하여 빈 문자열을 확인해야 한다.

위와같은 제약조건 애노테이션을 사용해 검증을 수행하기 위해서는 LocalValidatorFactoryBean을 빈으로 등록해 줘야 한다. LocalValidatorFactoryBean은 JSR-303의 검증기능을 스프링의 Validator처럼 사용할 수 있게 해주는 일종의 어댑터다. LocalValidatorFactoryBean을 빈으로 등록하면 컨트롤러에서 Validator타입으로 DI 받아서 @InitBinder에서 WebDataBinder에 설정하거나 코드에서 직접 Validator처럼 사용할 수 있다.

```
<bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean" />
```

빈 검증 기능을 validator로 사용하는 컨트롤러 예제이다.

```
@Controller
public class ExampleController {

    @Resource
    Validator validator;

    @InitBinder
    public void initBinder(WebDataBinder dataBinder){
        dataBinder.setValidator(this.validator);
    }

    @RequestMapping("/insertMember.do")
    public String insertMember(@ModelAttribute("memberVO") @Valid MemberVO memberVO, BindingResult
bindingResult, ..) {
        //..
    }
}
```

회원가입(\* 표시는 필수 입력 값 입니다.)

*ID	<input type="text"/>	영문 또는 숫자만 입력가능합니다. 아이디를 입력하세요.(1~20자)
*PASSWORD	<input type="password"/>	
*이름	<input type="text" value="관리자"/>	
*이메일	<input type="text" value="abc"/>	이메일 형식이 잘못되었습니다.
전화번호	<input type="text"/>	
핸드폰번호	<input type="text"/>	
우편번호	<input type="text"/>	
주소	<input type="text"/>	
상세주소	<input type="text"/>	

등록

초기화

참고자료

- Validator 예제



## 개요

Controller가 요청에 대한 처리를 하고, View 이름과 데이터(Model)를 ModelAndView에 저장해 DispatcherServlet에 반환(return)하면, DispatcherServlet은 View 이름을 가지고 ViewResolver에게서 실제 View 객체를 얻고, 이 View는 Controller가 저장한 Model 객체의 정보를 출력한다.

여기서는 View와 ViewResolver, 그리고 JSP에서 편리한 데이터 출력을 위해 스프링이 제공하는 Spring form tag library에 대해 설명한다.

- View
  - 개요
  - 설명
    - ViewResolver
    - View
    - Spring Tag Library
    - 전자정부프레임워크 Tag Library
  - 참고자료

- [ViewResolver](#)
- [View](#)
- [Spring Tag Library](#)
- [전자정부프레임워크 Tag Library](#)

## 설명

### ViewResolver

Controller는 코드내에서 실제 View 객체를 생성하지 않고 View 이름만을 결정할 수 있는데, 이로써 Controller와 View의 분리(decoupling)를 가능하게 한다.

```
package com.easycompany.controller.hierarchy;
...
public class EmployeeListController extends AbstractCommandController {
    @Override
    protected ModelAndView handle (HttpServletRequest request,
                                   HttpServletResponse response, Object command, BindException errors)
        throws Exception {
        ...
        List<Employee> employeeList = employeeService.getAllEmployees (commandMap);

        ModelAndView modelAndView = new ModelAndView ();
        modelAndView.addObject ("employeeList", employeeList);
        ...
        // 직접 View 객체를 생성하지 않고,
        // View view = new InternalResourceView ("/jsp/employeeList.jsp");
        // modelAndView.setView (view);
        // View 이름을 저장.
        modelAndView.setViewName ("employeeList");

        return modelAndView;
    }
}
```

이때, DispatcherServlet에 실제 View 객체를 구해주는건 Controller가 아니라 ViewResolver가 담당한다.

ViewResolver는 Controller가 반환한 ModelAndView 객체에 담긴 View 이름을 가지고 실제 View 객체를 반환하는 인터페이스이다.

Spring에서 제공하는 ViewResolver 구현 클래스는 아래와 같다.

ViewResolver	설명
XmlViewResolver	View이름과 View 클래스간의 매핑정보가 담긴 XML로 부터 View 이름에 해당하는 View를 구한다. 기본설정 파일은 /WEB-INF/views.xml이다.
ResourceBundleViewResolver	View이름과 View 클래스간의 매핑정보가 담긴 리소스 번들 (프로퍼티파일)로 부터 View 이름에 해당하는 View를 구한다. 기본설정 파일은 views.properties이다.
InternalResourceViewResolver/UrlBasedViewResolver	특정 디렉토리 경로의 JSP파일들을 호출할 때 편리하게 사용할수 있다. 기본적으로 사용하는 View 클래스는 InternalResourceView이며, View 이름이 곧 JSP 파일이름이 된다.
VelocityViewResolver /FreeMarkerViewResolver	Velocity/FreeMarker 연동시에 사용한다.

### InternalResourceViewResolver/UrlBasedViewResolver

비즈니스 로직 처리가 끝난 후 "/jsp/main/abc.jsp" 경로의 JSP 파일로 forwarding하는 Controller가 있다고 하면, InternalResourceViewResolver/UrlBasedViewResolver를 사용해서 아래와 같이 Controller를 작성하고, 빈 정의 파일에 설정할 수 있다.

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"/>
또는
<bean class="org.springframework.web.servlet.view.UrlBasedViewResolver "/>
```

```
@Controller
public class HelloController {
    @RequestMapping ("...")
    public String hello() {
        ... //비즈니스 로직 처리.
        return "/jsp/main/abc.jsp"; //뷰이름이 곧 JSP 파일의 경로.
    }
}
```

InternalResourceViewResolver/UrlBasedViewResolver의 프로퍼티 prefix, suffix를 사용하면 좀더 간단하게 처리할수 있는데,  
JSP가 특정 디렉토리 경로 아래에 있고, 예를 들어 /jsp/main 디렉토리 아래, 확장자는 .jsp 이라면,

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
      p:prefix="/jsp/main/" p:suffix=".jsp" />
또는
<bean class="org.springframework.web.servlet.view.UrlBasedViewResolver"
      p:prefix="/jsp/main/" p:suffix=".jsp" />
```

```
@Controller
public class HelloController {
    @RequestMapping ("...")
    public String hello() {
        ...
        return "abc"; //prefix와 suffix를 제외한 부분만 표기.
    }
}
```

간단히 뷰이름을 설정할 수 있다.

## View

Spring이 제공하는 View 클래스를 사용할 수도 있지만, UI Tool 등과의 연동등으로 인해 View 클래스를 직접 작성해야 하는 경우도 발생한다.

인터페이스 View를 직접 구현해서 View 클래스를 만들수도 있지만, AbstractView를 확장하여 구현해보자.  
renderMergedOutputModel 메소드를 구현하면 되는데, 아래와 같은 메소드 시그니처를 가지고 있다.

```
protected abstract void renderMergedOutputModel (Map model,
                                                  HttpServletRequest request,
                                                  HttpServletResponse response) throws Exception
```

AjaxTags란 Ajax 관련 오픈소스 사용을 위해 Model 객체의 데이터를 'text/xml' 형식으로 렌더링하는 View 클래스를 만들어 봤다.

```
package com.easycompany.view;

import java.io.PrintWriter;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.view.AbstractView;

public class AjaxXmlView extends AbstractView {

    @Override
    protected void renderMergedOutputModel (Map model,
                                             HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        response.setContentType ("text/xml");
        response.setHeader ("Cache-Control", "no-cache");
        response.setCharacterEncoding ("UTF-8");

        PrintWriter writer = response.getWriter();
        writer.write ((String) model.get ("ajaxXml"));
        writer.close();
    }
}
```

## Spring Tag Library

### message tag(<spring:message>)

스프링은 메시지 리소스 파일로 부터 메시지를 가져와 간편하게 출력할수 있도록, <spring:message> 태그를 제공한다.  
JSP 페이지의 타이틀을 <spring:message>를 이용해서 출력하는 예제를 만들어 보자.

빈 정의 파일에 리소스 번들 관련된 설정이 되어 있어야 한다.

```
<!-- Message Source-->
```

```
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource"
      p:basename="messages"/>
```

먼저 메시지 관련 리소스 파일에 코드값을 설정해준다. PropertiedEditor 같은 유틸의 도움을 받으면 편리하게 한글 입력-편집이 가능하다.

```
/easycompany/webapp/WEB-INF/classes/messages_ko.properties
```

```
...
# -- spring:message --
easaycompany.loginform.title=로그인페이지
easaycompany.employeeList.title=사원 정보 리스트 페이지
easaycompany.updateEmployee.title=사원 정보 수정 페이지
easaycompany.insertEmployee.title=사원 정보 입력 페이지
easaycompany.departmentList.title=부서 정보 리스트 페이지
easaycompany.updatedDepartment.title=부서 정보 수정 페이지
easaycompany.insertDepartment.title=부서 정보 입력 페이지
```

JSP 페이지에 커스텀 태그를 사용하기 위해 라이브러리 선언을 해줘야 한다. 그리고 <spring:message> 태그의 code 값에는 메시지 키값을 주면 된다.

```
...
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title><spring:message code="easaycompany.departmentList.title"/></title>
...
```

해당 화면의 타이틀이 “부서 정보 리스트 페이지”로 표기 될 것이다.

## form tag(<form:form>,<form:input>,...)

폼 관련 어플리케이션을 개발할 때는 스프링이 제공하는 폼 태그와 같이 사용하면 편리하다.

스프링 폼 태그는 Model 데이터의 커맨드 객체(command object)나 참조 데이터(reference data)들을 화면상에서 쉽게 출력하도록 도와 준다.

일단, 스프링 폼 태그를 사용하려면 페이지에 커스텀 태그 라이브러리를 선언해야 한다.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

스프링 폼 태그에는 아래와 같은 태그들이 있다.

<form:form>

<form:form>는 속성 commandName에 정의된 model attribute를 PageContext에 저장해서, <form:input>이나 <form:hidden>같은 태그들이 접근할 수 있도록 한다.

관련 속성은 아래와 같다.

- commandName : 참조하려는 model attribute 이름.
- action : 폼 전송할 URL
- method : 폼 전송시에 HTTP 메소드(GET,POST)
- enctype: 폼 전송시에 데이터 인코딩 타입.

<form:form> 태그를 사용하고 출력된 페이지에서 소스 보기로 HTML 코드를 열어보면 아래와 같이 HTML FORM 태그가 출력될 것 확인할 수 있을것이다.

```
<form:form commandName="department" action="http://myUrl..." method="post"> -> <form id="department"
action="http://myUrl..." method="post">
<form:form commandName="department"> -> <form id="department" action="현재페이지 URL" method="post">
```

<form:input>

HTML text타입의 input 태그에 commandName에 지정된 객체 프로퍼티를 바인딩하기 위해 사용한다.

path에 프로퍼티 이름을 적으면, text타입의 input 태그의 id, name 값이 프로퍼티 이름이 되고, value는 해당 프로퍼티의 값이 된다.

```
<form:form commandName="department">
  <tr>
    <th>부서이름</th>
    <td><form:input path="deptname" size="20"/></td>
  </tr>
</form:form>
```

아래와 같이 HTML로 출력된다.

```
<form id="department" action="/easycompany/updateDepartment.do?deptid=1100" method="post">
  <tr>
    <th>부서이름</th>
    <td><input id="deptname" name="deptname" type="text" value="회식매뉴혁신팀" size="20"/></td>
  </tr>
</form>
```

`<form:password>`

HTML password타입의 input 태그에 `commandName`에 지정된 객체 프로퍼티를 바인딩하기 위해 사용한다. 바인딩값을 표기하기 위해서는 `showPassword` 속성을 `showPassword="true"`로 지정해 주어야 한다.

`<form:hidden>`

HTML hidden타입의 input 태그에 `commandName`에 지정된 객체 프로퍼티를 바인딩하기 위해 사용한다.

`<form:select>`, `<form:options>`, `<form:option>`

HTML select, option 태그에 `commandName`에 지정된 객체 프로퍼티를 바인딩하기 위해 사용한다. 아래와 같이 `<form:select>`의 `path` 속성에 `commandName` 객체의 프로퍼티를 지정하고, `<form:options>`의 `items` 속성에 List, Map등의 Collection 객체를 값으로 주면,

```
<form:form commandName="department">
  <tr>
    <th>상위부서</th>
    <td>
      <form:select path="superdeptid">
        <option value="">상위부서를 선택하세요.</option>
        <form:options items="${deptInfoOneDepthCategory}" />
      </form:select>
    </td>
  </tr>
</form:form>
```

아래와 같이 HTML로 출력된다. `<form:select>`의 `path` 속성값과 일치하는 option 값이 있으면 `selected="selected"` 된다.

```
<form id="department" action="/easycompany/updateDepartment.do?deptid=1100" method="post">
  <tr>
    <th>상위부서</th>
    <td>
      <select id="superdeptid" name="superdeptid">
        <option value="">상위부서를 선택하세요.</option>
        <option value="5000">금융사업부</option>
        <option value="3000">IT연구소</option>
        <option value="4000">공공사업부</option>
        <option value="1000" selected="selected">경영기획실</option>
        <option value="2000">경영지원실</option>
      </select>
    </td>
  </tr>
</form>
```

`<form:checkboxes>`

`<form:checkbox>`

**error tag(`<form:errors>`)**

## 전자정부프레임워크 Tag Library

`<ui:pagination/>`

페이징 처리의 편의를 위해 `<ui:pagination/>` 태그를 제공한다.

`<ui:pagination/>`의 주요 속성은 아래와 같다.

이름	설명	필수여부
paginationInfo	페이징리스트를 만들기 위해 필요한 데이터. 데이터 타입은 <code>egovframework.rte.ptl.mvc.tags.ui.pagination.PaginationInfo</code> 이다.	yes
type	페이징리스트 렌더링을 담당할 클래스의 아이디. 이 아이디는 빈설정 파일에 선언된 프로퍼티 <code>rendererType</code> 의 key값이다.	yes
jsFunction	페이지 번호에 걸리게 될 자바스크립트 함수 이름. 페이지 번호가 기본적인 argument로 전달된다.	yes

ui 태그에 대한 라이브러리 선언을 해주고 페이징 리스트가 위치할 곳에 아래와 같이 사용하면 된다.

`paginationInfo` 속성에는 Controller에서 Model 객체에 저장한 `PaginationInfo`의 attribute name을 적어 주면 되고, `jsFunction` 속성은 페이징 리스트의 각 페이지 번호에 걸릴 링크인 자바스크립트 함수명을 적어 주면 된다.

`type` 속성은 빈 설정시에 `rendererType` 프로퍼티의 entry key값을 적어준다. 렌더링 타입을 태그에서 결정하는 것이다.

1. 관련 클래스 빈 설정한다.

```

<!-- For Pagination Tag -->
<bean id="imageRenderer" class="com.easycompany.tag.ImagePaginationRenderer"/>
<bean id="textRenderer" class="egovframework.rte.ptl.mvc.tags.ui.pagination.DefaultPaginationRenderer"/>
<bean id="paginationManager"
class="egovframework.rte.ptl.mvc.tags.ui.pagination.DefaultPaginationManager">
    <property name="rendererType">
        <map>
            <entry key="image" value-ref="imageRenderer"/>
            <entry key="text" value-ref="textRenderer"/>
        </map>
    </property>
</bean>

```

2.JSP에서 라이브러리를 선언한 후 사용한다.

```

<%@ taglib prefix="ui" uri="http://egovframework.gov/ctl/ui"%>
...
<script type="text/javascript">
    function linkPage(pageNo){
        location.href = "/easycompany/employeeList.do?pageNo="+pageNo;
    }
</script>
<body>
...
    <ui:pagination paginationInfo = "${paginationInfo}"
        type="image"
        jsFunction="linkPage" />
...
</body>

```

## 참고자료

- The Spring Framework - Reference Documentation 2.5.6
- Spring Framework API Documentation 2.5.6

## 개요

Spring 3.0에서 처음 소개된 스프링 전용 표현식 언어로 강력하고 유연하게 사용된다. SpEL은 빈 오브젝트에 직접 접근할 수 있는 표현식을 이용해서 프로퍼티 값을 능동적으로 가져오는 방법이며 가장 기본적인 방법이다. 또한 jsp에서 `<spring:eval>` 태그를 사용하여 SpEL을 적용 할 수도 있다.

- SpEL
  - 개요
  - 설명
    - 빈 설정파일을 사용하여 SpEL적용
    - JSP에서 SpEL적용
  - 참고자료

## 설명

### 빈 설정파일을 사용하여 SpEL적용

빈 프로퍼티에 값을 설정하면, 다른 빈이나 프로퍼티에 접근 가능하다.

- 다음의 빈에서 접근하는 예제이다.

```
<bean id="springTest" ..>
  <property name="test" value="Sample" />
</bean>

<bean id="testNames">
  <property name="name" value="#{springTest.test}" />
</bean>
```

- 다음은 `<util:properties>` 를 사용하여 프로퍼티에 접근하는 예제 이다.

globals.properties

```
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:1623/EASYCOMPANY
username=tex
password=texAdmin
```

context-datasource.xml

```
<util:properties id="dbprops" location="classpath:/egovframework/property/globals.properties" />

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="#{dbprops['driverClassName']}" />
  <property name="url" value="#{dbprops['url']}" />
  <property name="username" value="#{dbprops['username']}" />
  <property name="password" value="#{dbprops['password']}" />
</bean>
```

- 다음은 `<util:properties>` 를 사용하여 변수로 직접 주입하는 예제 이다.

```
@Value("#{dbprops.driverClassName}")
private String driverClassName;
```

또는

```
@Value("#{dbprops}")
private Dbproperties dbprops;
```

### JSP에서 SpEL적용

JSP의 EL대신에 Spring 3.0의 SpEL을 사용해서 값을 출력할 수 있다. JSP에서 SpEL을 사용하려면 태그 라이브러리를 추가해야 한다.

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
```

`<spring:eval>` 태그를 사용하여 JSP에서 SpEL을 사용한다.

모델 오브젝트를 직접 사용할 수 있다.

```
<spring:eval expression="sampleVO.money" />
```

메소드의 리턴값이 스트링일 경우, 메소드 자체를 호출할 수 있다.

```
<spring:eval expression="sampleVO.toString()" />
```

또한, @NumberFormat, @DateTimeFormat과 같은 컨버전 서비스에 등록되는 포맷터를 자동으로 적용 할 수 있다.

다음은 sampleVO의 일부이다.

```
/** 잔액 */  
@NumberFormat(pattern = "###,##0")  
private Integer money;
```

```
<spring:eval expression="sampleVO.money"/>
```

위와 같이 적용하면 입력 값에 따라 3자리마다 쉼표(.)가 출력된다.

**입력값: 1234000 출력값: 1,234,000**

모델에 직접 어노테이션으로 설정하지 않아도 new를 이용해 SpEL을 적용할 수 있다.

```
<spring:eval expression='new java.text.DecimalFormat("###,##0").format(price)'/>
```

### SpEL 예제

이름:

통장 잔액:  (숫자만 입력하세요)



### SpEL Sample

SpEL 적용 데이터1 : Admin님의 통장 잔고는 1,246,000원 입니다.

toString 적용 데이터 : 이름은 Admin, 현재 잔액은 1246000원

SpEL 적용 데이터2 : SampleProduct의 가격은 1,234,500원 입니다.

## 참고자료

- Spring Framework - Reference Document / 7. Spring Expression Language (SpEL)
- SpEL예제

Table of Contents
▪ Ajax 지원 서비스
▪ 개요
▪ 설명
▪ 설치
▪ AjaxTags Tag Reference
▪ 공통적인 개발 작업
▪ 예제
▪ 참고자료

개요

일반적으로 Ajax 기능은 javascript 언어로 개발하나, server-side 구현에 익숙한 J2EE 개발자들에게는 쉽지 않은 작업이 될 수 있다. Ajax 지원 서비스에서는 Ajax를 이용해 자주 사용되는 기능을 custom tag형태로 제공한다. 기능은 오픈소스 라이브러리인 AjaxTags를 이용한다.

설명

설치

시스템 환경 및 필요 라이브러리

- JDK 1.5
- Servlet container running Servlets 2.4+ and JSP 2.0+ (jsp-api 2.0, servlet-api 2.4)
- AjaxTags 라이브러리

설치 순서

1. AjaxTags Download 사이트에 가서 해당 라이브러리를 download한 후 WEB-INF/lib에 위치시킨다.
2. web.xml 설정.

```
<servlet>
    <servlet-name>sourceloader</servlet-name>
    <servlet-class>net.sourceforge.ajaxtags.servlets.SourceLoader</servlet-class>

    <init-param>
        <param-name>prefix</param-name>
        <param-value>/ajaxtags</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>sourceloader</servlet-name>
    <url-pattern>/ajaxtags/js/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>sourceloader</servlet-name>
    <url-pattern>/ajaxtags/img/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>sourceloader</servlet-name>
    <url-pattern>/ajaxtags/css/*</url-pattern>
</servlet-mapping>
```

AjaxTags Tag Reference

ajax:autocomplete

자동완성기능. 보통 검색 입력창에 prefix 문자를 입력하면 해당 추천 검색어를 보여주는 방식으로 이용.

파라미터	설명	필수여부
baseUrl	자동완성기능을 위한 결과 데이터를 보내주는 server-side 액션을 위한 URL.	yes
source	추천 검색어 리스트를 보여줄 텍스트 필드 이름. 입력 필드에 추천 검색어리스트를 보여준다면 target과 source를 동일하게 입력한다.	yes
target	사용자가 입력하는 텍스트 필드 이름.	yes
parameters	baseUrl에 추가할 파라미터들.여러개일 경우 comma로 구별한다.	yes
className	추천 검색어리스트에 적용할 CSS 클래스이름	yes
indicator	Ajax 요청중일때 보여줄 표시.	no



minimumCharacters	Ajax 요청을 위한 최소 입력값.	no
preFunction	Ajax 요청이 시작되기 전에 동작하는 function 이름.	no
postFunction	Ajax 요청이 완료된 후에 동작하는 function 이름.	no
errorFunction	Ajax 요청 error시에 동작하는 function 이름.	no

## ajax:select

하나의 선택트박스에서 값을 변경하면 다른 선택트박스에 연관된 값으로 리스트를 구성. Linked SelectBox.

파라미터	설명	필수여부
baseUrl	자동완성기능을 위한 결과 데이터를 보내주는 server-side 액션을 위한 URL.	yes
source	추천 검색어 리스트를 보여줄 텍스트 필드 이름. 입력 필드에 추천 검색어리스트를 보여준다면 target과 source를 동일하게 입력한다.	yes
target	사용자가 입력하는 텍스트 필드 이름.	yes
parameters	baseUrl에 추가할 파라미터들.여러개일 경우 comma로 구별한다.	no
eventType		no
executeOnLoad	응답 데이터로 select box를 구성하는 중일때 구성중인지를 별도 표시를 할지 여부.[default=false]	no
defaultOptions	Ajax 응답값이 없을때 보여줄 기본 리스트. comma로 구별하여 작성한다.	no
preFunction	Ajax 요청이 시작되기 전에 동작하는 function 이름.	no
postFunction	Ajax 요청이 완료된 후에 동작하는 function 이름.	no
errorFunction	Ajax 요청 error시에 동작하는 function 이름.	no
parser	응답 데이터에 대한 parser.[default=ResponseHtmlParser]	no

## ajax:tabPanel

탭으로 구성된 페이지들 새로 고침 없이 보여 줄때.

파라미터	설명	필수여부
id	tabPanel의 ID	yes
preFunction	Ajax 요청이 시작되기 전에 동작하는 function 이름.	no
postFunction	Ajax 요청이 완료된 후에 동작하는 function 이름.	no
errorFunction	Ajax 요청 error시에 동작하는 function 이름.	no
parser	응답 데이터에 대한 parser.[default=ResponseHtmlParser]	no

## others

이외에도 여러 기능이 있다. AjaxTags의 Tag 레퍼런스 및 사용법은 아래 [AjaxTags 사이트](#)에서 확인할 수 있다.

## 공통적인 개발 작업

AjaxTags의 어떤 태그를 사용하던지, 아래의 작업은 공통적으로 발생한다.

## JSP

### 태그 라이브러리 선언

```
<%@ taglib prefix="ajax" ri="http://ajaxtags.sourceforge.net/tags/ajaxtags" %>
```

### JavaScript, CSS 선언

```
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/prototype.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/scriptaculous/scriptaculous.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/overlibmws/overlibmws.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/ajaxtags.js"></script>
<link type="text/css" rel="stylesheet" href="<%=request.getContextPath()%>/ajaxtags/css/ajaxtags.css" />
<link type="text/css" rel="stylesheet" href="<%=request.getContextPath()%>/ajaxtags/css/displaytag.css" />
```

## Controller

AjaxTags를 사용하기 위해서는 결과 데이터가 AjaxTags에서 데이터 형식(XML style)을 갖추어야 한다.

이를 위해 AjaxTags는 AjaxXmlBuilder라는 데이터 가공을 위한 API를 제공한다.

결과 데이터를 AjaxXmlBuilder를 이용해서 변환하는 작업을 View에서 할 수도 있지만, View의 갯수가 기능 단위로 추가될 수도 있으므로, Controller에서 변환한 후에 Model 객체에 담아서 View로 보내고 View는 공통으로 하나를 사용하기를 권한다.

### org.ajaxtags.helpers.AjaxXmlBuilder

#### Collection 타입의 결과를 한번에 추가

```
List<Department> deptList = departmentService.getDepartmentList(param);
String result = new AjaxXmlBuilder().addItem(deptList, "deptname", "deptid").toString();
```

```
model.addObject("ajaxXml", ajaxXmlBuilder.toString());
```

## 한건씩 추가하기

```
List<Department> deptList = departmentService.getDepartmentList(param);
AjaxXmlBuilder ajaxXmlBuilder = new AjaxXmlBuilder();
for (Iterator iter = deptList.iterator(); iter.hasNext();) {
    Department dept = (Department) iter.next();
    ajaxXmlBuilder.addItem(dept.getDeptname(), dept.getDeptid());
}
model.addObject("ajaxXml", ajaxXmlBuilder.toString());
```

결과 데이터는 동일하다.

```
<?xml version="1.0" encoding="UTF-8"?>
<ajax-response>
  <response>
    <item>
      <name>점심메뉴기획팀</name>
      <value>1200</value>
    </item>
    <item>
      <name>야근금지역량팀</name>
      <value>1300</value>
    </item>
    ...
  </response>
</ajax-response>
```

## View

JSP 페이지에 프린트되는 일반적인 응답방식이 아니므로, 응답 처리를 위한 공통 View를 만들어야 한다.  
결과데이터의 형식(XML)을 응답 객체에 설정한다.  
Controller에서 보낸 Model객체의 결과데이터를 꺼내 write한다.

```
package com.easycompany.view;

import java.io.PrintWriter;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.view.AbstractView;

public class AjaxXmlView extends AbstractView {

    @Override
    protected void renderMergedOutputModel(Map model,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        response.setContentType("text/xml");
        response.setHeader("Cache-Control", "no-cache");
        response.setCharacterEncoding("UTF-8");

        PrintWriter writer = response.getWriter();
        writer.write((String) model.get("ajaxXml")); //Model Attribute 이름은 공통으로 사용하는 것으로...
        writer.close();
    }
}
```

## 예제

### ajax:autocomplete

사원 정보 조회 페이지에서, 조회 조건중에 하나인 이름 필드에 자동완성기능(autocomplete)을 적용해 보자.  
검색하려는 이름을 입력하기 시작하면, 입력값에 해당하는 prefix를 가진 이름들이 추천 리스트로 나온다.

사원정보 사원정보추가 부서정보 실적집계

사원번호 :	<input type="text"/>	부서번호 :	<input type="text"/>	이름 :	<input type="text" value="김"/>	<input type="button" value="검색"/>
--------	----------------------	--------	----------------------	------	--------------------------------	-----------------------------------

	사원번호	부서번호	이름	나이	이메일
	<a href="#">1</a>	1200	김길동	28	kkd@e
	<a href="#">2</a>	1100	김길수	39	kks@easycompany.com
	<a href="#">3</a>	1200	강감찬	17	kkc@easycompany.com

## JSP

/easycompany/webapp/WEB-INF/jsp/employeeelist.jsp

```
<%@ taglib prefix="ajax" uri="http://ajaxtags.sourceforge.net/tags/ajaxtags" %>
<!-- Ajax Tags -->
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/prototype.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/scriptaculous/scriptaculous.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/overlibmws/overlibmws.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/ajaxtags.js"></script>
```

```

<link type="text/css" rel="stylesheet" href="<%=request.getContextPath()%/ajaxtags/css/ajaxtags.css" />
<link type="text/css" rel="stylesheet" href="<%=request.getContextPath()%/ajaxtags/css/displaytag.css" />
...
<form:form commandName="searchCriteria" action="/easycompany/employeeList.do">
<table width="50%" border="1">
<tr>
<td>사원번호 : <form:input path="searchEid"/> </td>
<td>부서번호 : <form:input path="searchDid"/> </td>
<td>이름 : <form:input path="searchName"/>
</td>
<td><input type="submit" value="검색" onclick="this.disabled=true,this.form.submit();" /></td>
</tr>
</table>
</form:form>

<ajax:autocomplete
baseUrl="<${pageContext.request.contextPath}/suggestName.do"
source="searchName"
target="searchName"
className="autocomplete"
minimumCharacters="1" />
...

```

## Controller

com.easycompany.controller.annotation.AjaxController

```

package com.easycompany.controller.annotation;
import net.sourceforge.ajaxtags.xml.AjaxXmlBuilder;
import com.easycompany.view.AjaxXmlView;

@Controller
public class AjaxController {

    @Autowired
    private EmployeeService employeeService;

    @Autowired
    private DepartmentService departmentService;

    @RequestMapping("/suggestName.do")
    protected ModelAndView suggestName(@RequestParam("searchName") String searchName) {

        ModelAndView model = new ModelAndView(new AjaxXmlView());
        List<String> nameList = employeeService.getNameListForSuggest(searchName);

        AjaxXmlBuilder ajaxXmlBuilder = new AjaxXmlBuilder();

        for (String name : nameList) {
            ajaxXmlBuilder.addItem(name, name, false);
        }
        model.addObject("ajaxXml", ajaxXmlBuilder.toString());
        return model;
    }
}

```

## 한글처리설정

위 예제에서 보면 '/suggestName.do?searchName=김' 같이, 사원 이름 prefix값이 파라미터로 전달되는데, 파라미터 값이 한글인 경우 제대로 처리되기 위해서는, {Tomcat DIR}/conf/server.xml에 인코딩 처리를 해줘야 한다. UTF-8 인코딩을 한다면, <Connector/> 태그에 URIEncoding="utf-8"을 추가하면 된다.

```

...
<Connector port="8080" maxHttpHeaderSize="8192"
maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
enableLookups="false" redirectPort="8443" acceptCount="100"
connectionTimeout="20000" disableUploadTimeout="true" URIEncoding="utf-8"/>
...
<Connector port="8009"
enableLookups="false" redirectPort="8443" protocol="AJP/1.3" URIEncoding="utf-8" />

```

## ajax:select

사원 정보 수정(입력) 페이지에서, 상위 부서 정보 select box에서 한 부서를 선택하면, 하위 부서 정보 select box는 해당 상위 부서에 속한 하위 부서 정보들로 옵션을 구성한다.

사원번호	6
부서번호	<div> <div>경영기획실</div> <div>마은금지역합팀</div> </div>
이름	재기찬
비밀번호	6
주민번호	-
나이	34
이메일	jackie@easycompany.com

## JSP

/easycompany/webapp/WEB-INF/jsp/addemployee.jsp, /easycompany/webapp/WEB-INF/jsp/modifyemployee.jsp

```

<%@ taglib prefix="ajax" uri="http://ajaxtags.sourceforge.net/tags/ajaxtags" %>
...

```

```

<!--Ajax Tags-->
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/prototype.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/scriptaculous/scriptaculous.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/overlibmws/overlibmws.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/ajaxtags.js"></script>
<link type="text/css" rel="stylesheet" href="<%=request.getContextPath()%>/ajaxtags/css/ajaxtags.css" />
<link type="text/css" rel="stylesheet" href="<%=request.getContextPath()%>/ajaxtags/css/displaytag.css" />
...
<form:form commandName="employee">
...
    <tr>
        <th>부서번호</th>
        <td>
            <form:select path="superdeptid">
                <option value="">상위부서를 선택하세요.</option>
                <form:options items="${deptInfoOneDepthCategory}" />
            </form:select>
            <form:select path="departmentid">
                <option value="">근무부서를 선택하세요.</option>
                <form:options items="${deptInfoTwoDepthCategory}" />
            </form:select>
        </td>
    </tr>
</form:form>
<ajax:select
    baseUrl="<%=pageContext.request.contextPath%>/autoSelectDept.do"
    parameters="depth=2,superdeptid={superdeptid}"
    source="superdeptid"
    target="departmentid"
    emptyOptionName="Select model"/>
...

```

## ajax:tabPanel, ajax:tab

부서정보 페이지에서 각 상위부서에 속한 하위부서리스트를 보여줄때, tab으로 처리해서 보여준다.

경영기획실

경영지원실

IT연구소

공공사업부

금융사업부

	부서번호	부서명	상위부서명
	<a href="#">1100</a>	회식메뉴혁신팀	경영기획실
	<a href="#">1200</a>	점심메뉴기획팀	경영기획실
	<a href="#">1300</a>	야근금지역량팀	경영기획실
	<a href="#">1400</a>	사랑의 짝대기팀	경영기획실

## JSP

/easycompany/webapp/WEB-INF/jsp/departmenlist.jsp

```

<%@ taglib prefix="ajax" uri="http://ajaxtags.sourceforge.net/tags/ajaxtags" %>
<!--Ajax Tags-->
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/prototype.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/scriptaculous/scriptaculous.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/overlibmws/overlibmws.js"></script>
<script type="text/javascript" src="<%=request.getContextPath()%>/ajaxtags/js/ajaxtags.js"></script>
<link type="text/css" rel="stylesheet" href="<%=request.getContextPath()%>/ajaxtags/css/ajaxtags.css" />
<link type="text/css" rel="stylesheet" href="<%=request.getContextPath()%>/ajaxtags/css/displaytag.css" />
...
    <ajax:tabPanel id="departmentTab">
        <c:forEach items="${departmentlist}" var="departmentinfo" varStatus="status">
            <c:choose>
                <c:when test="${status.first}">
                    <ajax:tab caption="${departmentinfo.deptname}"
                        baseUrl="/easycompany/subDepartmentList.do?superdeptid=${departmentinfo.deptid}&depth=2"
                        defaultTab="true"/>
                </c:when>
                <c:otherwise>
                    <ajax:tab caption="${departmentinfo.deptname}"
                        baseUrl="/easycompany/subDepartmentList.do?superdeptid=${departmentinfo.deptid}&depth=2"/>
                </c:otherwise>
            </c:choose>
        </c:forEach>
    </ajax:tabPanel>
...

```

## 참고자료

 AjaxTags Web Site

개요

전자정부 표준 프레임워크에서는 Spring MVC 에서 제공하는 LocaleResolver를 이용한다.

우리는 여기서 LocaleResolver를 알아보고 적용하는 설정과 다국어가 적용된 message resource 를 가져와 활용하는 것을 보도록 하겠다.

Spring MVC 는 다국어를 지원하기 위하여 아래와 같은 종류의 LocaleResolver 를 제공하고 있다.

- CookieLocaleResolver : 쿠키를 이용한 locale정보 사용
- SessionLocaleResolver : 세션을 이용한 locale정보 사용
- AcceptHeaderLocaleResolver : 클라이언트의 브라우저에 설정된 locale정보 사용

Bean 설정 파일에 정의하지 않을 경우 AcceptHeaderLocaleResolver 를 default로 적용된다.

설명

3가지의 LocaleResolver

CookieLocaleResolver

CookieLocaleResolver 를 설정하는 경우 사용자의 쿠키에 설정된 Locale 을 읽어들인다.  
sample-servlet.xml

```
<...  
<bean id="localeResolver"  
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver" >  
  <property name="cookieName" value="clientlanguage"/>  
  <property name="cookieMaxAge" value="100000"/>  
  <property name="cookiePath" value="web/cookie"/>  
</bean>  
<...>
```

다음과 같은 속성을 사용할 수 있다.

속성	기본값	설명
cookieName	classname + locale	쿠키명
cookieAge	nteger.MAX_INT	-1 로 해두면 브라우저를 닫을 때 없어짐
cookiepath	/	Path 를 지정하면 해당하는 Path와 그 하위 Path 에서만 참조

SessionLocaleResolver

request가 가지고 있는 session으로 부터 locale 정보를 가져온다.  
sample-servlet.xml

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

AcceptHeaderLocaleResolver

사용자의 브라우저에서 보내진 request 의 헤더에 accept-language 부분에서 Locale 을 읽어들인다. 사용자의 브라우저 의 Locale 을 나타낸다.  
sample-servlet.xml

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver" />
```

XML 설정

Web 을 통해 들어오는 요청을 Charset UTF-8 적용한다.

CharacterEncodingFilter 을 이용하여 encoding 할 수 있도록 아래와 같이 세팅한다.

web.xml

```
...
<filter>
  <filter-name>encoding-filter</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>encoding-filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
...
```

## Spring Configuration

사용자의 브라우저의 Locale 정보를 이용하지 않고 사용자가 선택하여 언어를 직접 선택할 수 있도록 구현하려 한다면 CookieLocaleResolver 나 SessionLocaleResolver 를 이용한다. 먼저 다국어 지원을 하므로 메시지를 MessageSource 로 추출하여 구현해야 한다.

자세한 MessageSource 내용은 [Resource](#) 를 참고하길 바란다.

messageSource는 아래와 같이 설정하였다.

sample-servlet.xml

```
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>classpath:/message/message</value>
    </list>
  </property>
</bean>
```

message properties 파일은 아래와 같다.

locale에 따라 ko, en 으로 구분하였다.

message\_ko.properties

```
view.category=카테고리
```

message\_en.properties

```
view.category=category
```

ResourceBundleMessageSource 는 beanname 명으로 messages 을 받아오는데 디폴트의 경우는 messages.properties 에서 message를 받아오고 locale 이 한국어일 경우는 messages\_ko\_KR.properties 에서 받아오고 영어일 경우는 messages\_en\_US.properties 에서 받아온다. 아래와 같이 localeResolver 과 localeChangeInterceptor 를 등록하고 Annotation 기반에서 동작할 수 있도록 **DefaultAnnotationHandlerMapping** 에 interceptor 로 등록을 해준다.

sample-servlet.xml

```
<!-- 세션을 이용한 Locale 이용시 -->
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>

<!-- 쿠키를 이용한 Locale 이용시 -->
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
  <property name="paramName" value="language"/>
</bean>

<bean id="annotationMapper"
class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">
  <property name="interceptors">
    <list>
      <ref bean="localeChangeInterceptor"/>
    </list>
  </property>
</bean>
```

SessionLocaleResolver 를 이용하여 위와 같이 하였을 경우 Locale 결정은 **language** 로 Request Parameter 로 넘기게 된다. 카테고리 용어가 영어와 한글로 바뀌는 것을 아래와 같이 볼 수 있다.

리스트를 보여주는 화면에 예를 보자면 Spring 메시지 태그를 이용하여

```
<spring:message code="view.category" />
```

으로 표현한다.

```
<%@ taglib prefix="spring" uri=http://www.springframework.org/tags %>
<form:form commandName="message" >
....
    <table border="1" cellspacing="0" class="boardList" summary="List of Category">
        <thead>
            <tr>
                <th scope="col">No.</th>
                <th scope="col">
                    <input name="checkAll" type="checkbox" class="inputCheck"
                    title="Check All" onclick="javascript:fnCheckAll();" />
                </th>
                <th scope="col"><spring:message code="view.category" /> ID</th>
                <th scope="col"><spring:message code="view.category" /> 명</th>
                <th scope="col">사용여부</th>
                <th scope="col">Description</th>
                <th scope="col">등록자</th>
            </tr>
        </thead>
    </table>
....
```

화면상으로 해당 페이지를 실행해보면 아래와 같다.

한글인 경우 :

http://localhost:8080/sample-web/sale/listCategory.do?language=ko

No.	<input type="checkbox"/>	카테고리 ID	카테고리 명	사용여부	
1	<input type="checkbox"/>	0000000001	Sample Test	Y	This is initial test data,
2	<input type="checkbox"/>	0000000002	test Name	Y	test Desc

Manage CheckedList All

EndPrev1112

영어인 경우 :

http://localhost:8080/sample-web/sale/listCategory.do?language=en

No.	<input type="checkbox"/>	category ID	category 명	사용여부	
1	<input type="checkbox"/>	0000000001	Sample Test	Y	This is
2	<input type="checkbox"/>	0000000002	test Name	Y	test De

Manage CheckedList All

Er

## Java 소스내에서 locale 적용 메세지 가져오기

참고로 MessageSource 는 아래와 같은 메소드로 이루어져 있다.(실제로 여기서의 구현체는 ResourceBundleMessageSource 임.)

```
MessageSource
- getMessage(String, Object[], String, Locale)
- getMessage(String, Object[], Locale)
- getMessage(MessageSourceResolvable, Locale)
```

```
String msg = messageSource.getMessage(messageKey, messageParameters, defaultMessage, locale);
```

## 참고자료



개요	▪Security 서비스 ▪개요 ▪설명 ▪참고자료
----	--------------------------------------

인증, 권한 같은 일반적인(통상적인) 개념의 Security 서비스는 Spring Security를 활용한 공통기반 레이어에서 제공한다. 화면 처리 레이어의 Security 서비스는 입력값 유효성 검증 기능을 제공한다. 입력값 유효성 검증(validation)을 위한 기능은 Valang, Jakarta Commons, Spring 등에서 제공하는데, 여기서는 기반 오픈소스로 Jakarta Commons Validator를 선택했다. MVC 프레임워크인 Spring MVC와 Jakarta Commons Validator의 연계와 활용방안 설명한다.

설명

Jakarta Commons Validator는 필수값, 각종 primitive type(int,long,float...), 최대-최소길이, 이메일, 신용카드번호등의 값 체크 등을 할 수 있도록 Template이 제공된다. 또한 client-side, server-side의 검증을 함께 할 수 있으며, Configuration과 예러메시지를 client-side, server-side 별로 따로 하지 않고 한곳에 같이 쓰는 관리상의 장점이 있다. 자세한 설명은 아래의 문서를 참조하라.

- [Spring Framework에서 Commons Validator 사용](#): 입력값을 Jakarta Commons Validator를 이용하여 client-side, server-side 검증(validation)함.
- [Commons Validator에 validation rule 추가](#) : Commons Validator에 주민등록번호 validation rule을 추가해본다.

참고자료

## 개요

전자정부 표준프레임워크와 UI 솔루션(Rich Internet Application) 연동에 대해 살펴 본다.

UI Adaptor를 적용하는 방식은 특정한 하나의 방법을 표준화하기 어렵다.  
보통 Web Framework 과 UI 솔루션과의 연동을 하는 방법중 가장 많이 사용하는 방식은 Controller 역할을 수행하는 Servlet 객체에서 업무 로직을 호출전 데이터를 DTO 형태로 변화하여 업무로직으로 넘기는 방식이다.


전자정부 표준프레임워크에서는 Spring MVC Annotation 기반으로 개발시 요청되는 URI 와 Controller 클래스내의 메소드를 매핑하고 있다.

따라서 메소드의 파라미터로 넘어오는 객체가 request 객체가 아닌 업무용 DTO 클래스로 넘어올수 있도록 가이드 하는 방식을 선택했다.

(사실 @ModelAttribute 를 이용하는 것과 같다.)

하지만 프로젝트 별로 비기능 요구사항의 특성을 고려하여 적합한 구조를 정의하여 적용하는 것이 필요하다.

UI 솔루션 업체별 상세 가이드

-  마이플랫폼\_연동\_샘플\_설명서포함.zip - written by 고석률(MiPlatform)
- [토마토시스템즈](#) - 토마토시스템즈
- [SHIFT](#) - SHIFT(Gauce)

## 설명

중점적으로 우리가 살펴볼 내용은 Controller 앞단에서 UI 솔루션으로부터 넘어온 데이터를 DTO 로 변화하는 과정이다.  
데이터 변화를 위해 우리는 **ArgumentResolver** 를 이용한 방법을 살펴보도록 하겠다.

UI 솔루션으로 넘어오는 데이터 객체는 request 객체에 포함되어 넘어온다. 우리가 필요한 것은 업무용 DTO 클래스이다.  
업무용 DTO 클래스는 URI(@RequestMapping)와 매핑된 Controller 메소드의 파라미터로 존재하게 된다.  
Controller 메소드의 파라미터에 설정된 클래스(여기서는 DTO)를 AnnotationMethodHandlerAdapter 에서 그에 해당하는 ArgumentResolvers(customArgumentResolvers포함) 를 호출해준다.

따라서 우리는 ArgumentResolver를 확장하여 CustomRiaArgumentResolver 개발하여 AnnotationMethodHandlerAdapter에 등록한다.

CustomRiaArgumentResolver 에서 리턴되는 객체는 Contorller 단의 메소드의 파라미터로 이용된다.

\* 참고 : AnnotationMethodHandlerAdapter 는 URI 와 매핑되는 Contorller 의 메소드를 실행시 파라미터로 존재하는 객체타입에대한 ArgumentResolver를 실행하여 가져온다.

그리고 Controller단의 실행 결과는 ViewResovler를 통해 RiaView 로 전송되며 RiaVeiw는 결과물인 DTO 를 UI 솔루션 데이터 타입으로 변환하여 response로 보내어 진다.

다시 핵심적인 내용을 정리하자면 다음과 같다.

1. AnnotationMethodHandlerAdapter 의 CustomRiaArgumentResolver 등록 ⇒ **CustomRiaArgumentResolver**
2. UIAdaptor 등록 ⇒ **UIAdaptorImpl**
3. RiaView 등록 ⇒ **RiaView**

사용되는 DTO 를 아래 클래스로 생각하고 살펴보겠다.

UdDTO.java(샘플)

```
...
public class UdDTO implements Serializable {

    private Map variableList ;
    private Map dataSetList ;
    private Map Objects ;

    public void setVariableList (Map variableList) {
        this.variableList= variableList;
    }

    public void setDataSetList (Map dataSetList) {
        this.dataSetList= dataSetList;
    }

    public Map getVariableList () {
        return variableList;
    }

    public Map getDataSetList () {
```

```

        return dataSetList;
    }

    public void setObjects(Map objects) {
        Objects = objects;
    }

    public Map getObjects() {
        return Objects;
    }
}
...

```

## UI솔루션데이터에서 DTO로 변환

다음은 UTO 를 가져와 UI 솔루션에 의해 들어오는 객체로부터 UTO 로 변환하는 부분을 설명하겠습니다.  
변환을 담당하는 UIAdaptorImpl 객체는 AnnotationMethodHandlerAdapter 의 **CustomRiaArgumentResolver** 에 설정된 다.

### 설정정보(CustomRiaArgumentResolver)

```

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
    <property name="webBindingInitializer">
        <bean class="egovframework.rte.fdl.web.common.EgovBindingInitializer" />
    </property>
    <property name="customArgumentResolvers">
        <list>
            <bean class="egovframework.rte.fdl.sale.web.CustomRiaArgumentResolver">
                <property name="uiAdaptor">
                    <ref bean="riaAdaptor" />
                </property>
            </bean>
        </list>
    </property>
</bean>

```

WebArgumentResolver의 구현체인 CustomRiaArgumentResolver 는 uiAdaptor 에 세팅된 Adaptor 를 실행해준다.

### CustomRiaArgumentResolver.java

```

public class CustomRiaArgumentResolver implements WebArgumentResolver {
    private UiAdaptor uiA;

    public void setUiAdaptor(UiAdaptor uiA) {
        this.uiA = uiA;
    }

    public Object resolveArgument(MethodParameter methodParameter, NativeWebRequest webRequest) throws Exception
    {
        Class<?> type = methodParameter.getParameterType();
        Object uiObject = null;

        if (uiA == null)
            return UNRESOLVED;

        //Controller의 실행되는 메소드의 파라미터타입 정보가 MethodParameter를 통해 넘어온다.
        //설정한 UIAdaptor 구현체에 등록되어 있는 UTO 와 비교한다.
        if (type.equals(uiA.getModelName())) {
            HttpServletRequest request = (HttpServletRequest) webRequest.getNativeRequest();
            // 여기서 데이터 만들어 넘긴다.
            uiObject = (UdDTO) uiA.convert(request);
            return uiObject;
        }

        return UNRESOLVED;
    }
}
...

```

UiAdaptor 의 구현체는 아래와 같다. 여기서는 Miplatform 의 예를 들어 코드 작성하였다.  
UI 솔루션의 객체에서 DTO 로 데이터를 옮기는 역할은 converte4In 메소드에서 수행된다.

### RiaAdaptorImpl.java(MiPlatform => UdDTO)

```

public class RiaAdaptorImpl implements UiAdaptor {
    protected Log log = LoggerFactory.getLog(this.getClass());

    //resolveArgument 메소드에서 호출하는 메소드
    public Object convert(HttpServletRequest request) throws Exception {
        PlatformRequest platformRequest = null;

        try {
            platformRequest = new PlatformRequest(request, PlatformRequest.CHARSET_UTF8);
            platformRequest.receiveData();
        } catch (IOException ex) {
            ex.printStackTrace();
            // throw new IOException("PlatformRequest error");
        }

        //UI 솔루션 데이터 에서 DTO 객체로 변환
        UdDTO dto = converte4In(platformRequest);
        return dto;
    }

    private UdDTO converte4In(PlatformRequest platformRequest) {
        UdDTO dto = new UdDTO();
    }
}

```

```

        //... DTO 또는 VO 값 채우기
        .....
        return dto;
    }

    public Class getModelName() {
        return UdDTO.class;
    }
}

```

## Controller 메소드 구현

UdDTO 클래스는 CustomRiaArgumentResolver 에서 만들어져 Controller 의 메소드의 parameter 형태로 가져온다. 아래 예는 Controller 단의 메소드 이다.

### XXCategoryController.java

```

...
@RequestMapping("/sample/miplatform.do")
public ModelAndView selectCategoryList4Mi(UdDTO miDto, Model model) throws Exception {

    ModelAndView mav = new ModelAndView("riaView");

    //조희조건이 있을 경우 사용될 Map
    Map<String, String> smp = new HashMap<String, String>();
    try {

        //Biz Layer 를 호출 한다.
        List resultList = categoryService.selectCategoryList(smp);

        //결과값을 모델에 저장
        mav.addObject("MiDTO", resultList);

    } catch (Exception ex) {
        log.info(ex.getStackTrace(), ex);
    }
    return mav;
}

```

## BeanNameViewResolver 설정

모델 객체의 이름이 riaView 이다. 이것은 Bean Name을 직접 명시 한것으로 아래와 같은 설정(BeanNameViewResolver)이 필요하다.

```

<bean class="org.springframework.web.servlet.view.BeanNameViewResolver" p:order="0" />
<bean class="org.springframework.web.servlet.view.UrlBasedViewResolver"
    p:order="1" p:viewClass="org.springframework.web.servlet.view.JstlView"
    p:prefix="/WEB-INF/jsp/" p:suffix=".jsp" />
<bean id="riaView" class="egovframework.rte.fdl.sale.web.RiaView" />

```

## RiaView 구현

RiaView 의 코드는 아래와 같다. DTO 를 업체에 맞춰 다시 가져 나가기 위해 convert 하는 모듈이다. 여기서는 Miplatform 객체로 변환한다. egovDs 라는 DataSet 으로 객체화 한후 stream 형태로 보내는 로직이다. 따라서 업체별 데이터 형태로 변환하여 보내도록 수정하면 된다.

### RiaView.java(DTO ⇒ MiPlatform)

```

.....
public class RiaView extends AbstractView {

    protected Log log = LogFactory.getLog(this.getClass());

    @SuppressWarnings("unchecked")
    @Override
    protected void renderMergedOutputModel(Map model, HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        VariableList miVariableList = new VariableList();
        DatasetList miDatasetList = new DatasetList();
        PlatformData platformData = new PlatformData(miVariableList, miDatasetList);

        List list = (List) model.get("MiDTO");

        Iterator<Map> iterator = list.iterator();
        Iterator<Map> dataIterator = list.iterator();

        Dataset dataset = new Dataset("egovDs");

        while (iterator.hasNext()) {
            // Header 세팅
            Map<String, Object> record = iterator.next();
            Iterator<String> si = record.keySet().iterator();

            while (si.hasNext()) {
                String key = si.next();
                dataset.addColumn(key, ColumnInfo.COLUMN_TYPE_STRING, (short) 255);
            }
        }

        while (dataIterator.hasNext()) {
            Map<String, Object> record = dataIterator.next();

```

```

        Iterator<String> si = record.keySet().iterator();
        // Header 세팅
        while (si.hasNext()) {
            String key = si.next();
            dataset.addColumn(key, ColumnInfo.COLUMN_TYPE_STRING, (short) 255);
        }

        // Value 세팅
        int row = dataset.appendRow();
        Iterator<String> si2 = record.keySet().iterator();
        while (si2.hasNext()) {
            String key = si2.next();
            String value = (String) record.get(key);

            System.out.println("key = " + key + " , value = " + value);
            dataset.setColumn(row, key, value);
        }
        miDatasetList.add(dataset);
    }
    try {
        new PlatformResponse(response, PlatformConstants.CHARSET_UTF8).sendData(platformData);
    } catch (IOException ex) {
        if (log.isDebugEnabled()) {
            log.error("Exception occurred while writing xml to MiPlatform Stream.", ex);
        }
        throw new Exception();
    }
}

```

## 참고자료