



Spring Web MVC

강사 : 강병준

- 
1. 이클립스 file – new – project – spring project – spring MVC project
 2. tooLevelPackage com.ch.helloworld
 3. 실행명령 ; <http://localhost:8181/helloworld/>
- 

스프링에서 하나의 요청에 대한 Life Cycle

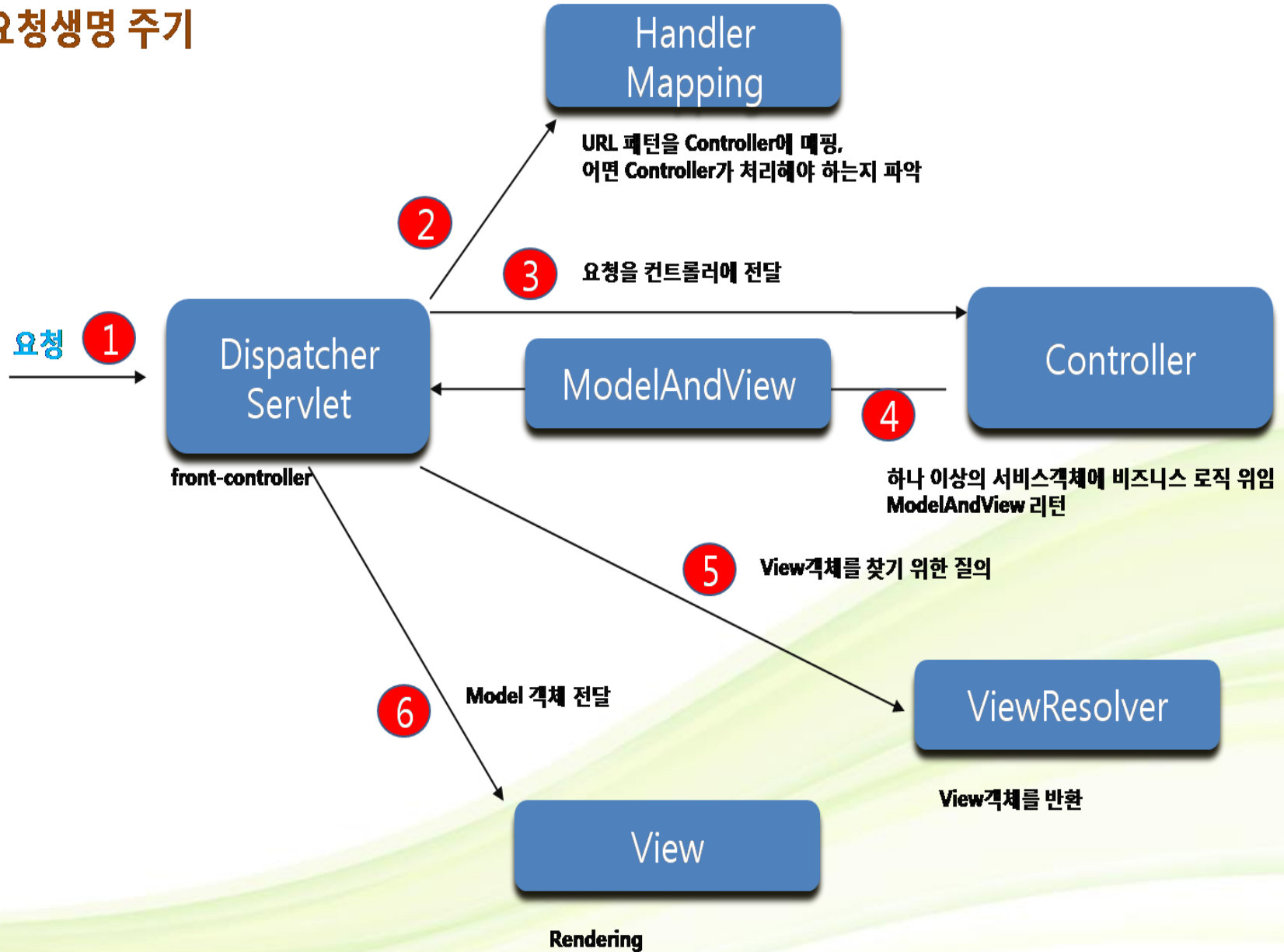
클라이언트의 요청에 대한 최초 진입 지점은 DispatcherServlet이 담당한다. DispatcherServlet은 Spring Bean Definition에 설정되어 있는 Handler Mapping 정보를 참조하여 해당 요청을 처리하기 위한 Controller를 찾는다. DispatcherServlet은 선택된 Controller를 호출하여 클라이언트가 요청한 작업을 처리한다.

Controller는 Business Layer와의 통신을 통하여 원하는 작업을 처리한 다음 요청에 대한 성공유무에 따라 ModelAndView 인스턴스를 반환한다. ModelAndView 클래스에는 UI Layer에서 사용할 Model데이터와 UI Layer로 사용할 View에 대한 정보가 포함되어 있다.

DispatcherServlet은 ModelAndView의 View의 이름이 논리적인 View 정보이면 ViewResolver를 참조하여 이 논리적인 View 정보를 실질적으로 처리해야 할 View를 생성하게 된다.

DispatcherServlet은 ViewResolver를 통하여 전달된 View에게 ModelAndView를 전달하여 마지막으로 클라이언트에게 원하는 UI를 제공할 수 있도록 한다. 마지막으로 클라이언트에게 UI를 제공할 책임은 View 클래스가 담당하게 된다.

요청생명 주기



/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
<!-- The definition of the Root Spring Container shared by all Servlets and Filters -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
</context-param>
<!-- Creates the Spring Container shared by all Servlets and Filters -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<!-- Processes application requests -->
<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

web.xml

web.xml 파일은 웹애플리케이션이 처음 로딩될 때 실행되는 것으로 아주 중요한 파일이다. listener 태그안에 있는 ContextLoaderListener는 웹애플리케이션의 컨텍스트를 생성하는데 DispatcherServlet이 이러한 역할을 할수도 있다.

"/"요청이 들어오면 "appServlet"이 처리하는데 이파일은 실제 springframework.web.servlet.DispatcherServlet 파일이며 읽어 올 설정파일은 /WEB-INF/spring/appServlet/servlet-context.xml 이다. ContextLoaderListener와 DispatcherServlet은 둘다 웹애플리케이션 컨텍스트를 생성하지만 ContextLoaderListener가 ROOT 컨텍스트이면 DispatcherServlet은 자식 컨텍스트이다.

브라우저에서 <http://localhost:8181/helloworld/> 라는 요청을 보내면 DispatcherServlet이 어떤 컨트롤러가 처리할 지를 servlet-context.xml 파일을 읽어 아는 것이다.

web.xml 서블릿 설정에서 <load-on-startup>

<load-on-startup> 엘리먼트의 사용의 목적

웹 어플리케이션 구동시 자동으로 서블릿 클래스를 초기화(init 메서드 호출) 시키기 위한 목적

<load-on-startup> 값이 0이거나 양수인 경우

해당 서블릿이 속한 웹 어플리케이션이 실행될 때, 초기화(init 메서드 호출) 한다.

즉, WAS(컨테이너)가 실행되는 시점에 서블릿이 초기화된다.

가지고 있는 값에 따라 초기화 순서가 결정되며, 값이 같을 경우 WAS 임의로 순서를 정해서 초기화한다.

<load-on-startup> 값이 음수이거나 해당 엘리먼트가 없는 경우

해당 서블릿이 실행되는 시점에 초기화(init 메서드 호출) 한다.

웹 어플리케이션이 시작되었더라도, 해당 서블릿에 대한 요청이 없다면 서블릿의 초기화(init 메서드의 호출)가 되지 않을 수도 있음 (WAS가 임의의 시점에 호출할 수도 있음)

/WEB-INF/spring/appServlet/servlet-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
<!-- DispatcherServlet Context: defines this servlet's request-processing infrastructure -->
<!-- Enables the Spring MVC @Controller programming model -->
<annotation-driven />
<!-- Handles HTTP GET requests for /resources/** by efficiently serving up static resources in
the ${webappRoot}/resources directory -->
<resources mapping="/resources/**" location="/resources/" />
<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-
INF/views directory -->
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<beans:property name="prefix" value="/WEB-INF/views/" />
<beans:property name="suffix" value=".jsp" />
</beans:bean>
<context:component-scan base-package="com.ch.helloworld" />
</beans:beans>
```


/WEB-INF/spring/root-context.xml

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

```
<!-- Root Context: defines shared resources visible to all other web components -->
```

```
</beans>
```

```
%
```

```
<context-param>
```

```
<param-name>contextConfigLocation</param-name>
```

```
<param-value>/WEB-INF/spring/root-context.xml</param-value>
```

```
</context-param>
```

@Controller

Controller를 생성할 때 interface없이 선언만으로 가능
설정 XML 에 기술해서 해당 컨트롤러 역할을하는 프로그램을 찾아 가던 것을
스프링 버전이 올라가면서 Annotation 만으로 해결이 가능해졌다.
Controller 클래스 내부에는 각 URL 요청을 담당하는 메소드가 존재한다.

@RequestMapping

어노테이션을 통해 등록하며,
value 는 URL 경로를,
method 는 HTTP Method 를 등록하면 된다.

Home Controller 는 / 에 대한 요청밖에 없지만,
User Controller 라면 /users /users/51041 등을 처리할 수 있는
여러 메소드가 있을 수 있다.

Controller 의 @RequestMapping 메소드는

Model 이라는 파라미터를 받는데,
이건 return 문에서 지정한 Home 이라는
이름의 View 에 적용할 **Model Attribute** 다

기본 HandlerMapping 구현 클래스인 BeanNameUrlHandlerMapping 클래스 사용
ViewResolver 인터페이스도 설정파일에 구현 클래스를 정의하지 않으면 기본인
InternalResourceViewResolver를 사용

BeanNameUrlHandlerMapping 클래스

스프링 설정 파일에 컨트롤러를 정의하면서 지정한 name 속성의 값과 웹 요청 URL을
매핑하는 HandlerMapping 구현 클래스.

MVC 기본 ViewResolver인 InternalResourceViewResolver 클래스를 암묵적 사용.
명시적으로 정의하면 뷰 정보에 prefix 프로퍼티나 suffix를 추가하거나 뷰의 구현 클래스 지정

스프링 설정파일 상 BeanNameUrlHandlerMapping 예

```
<!-- ViewResolver -->  
<bean id="InternalResourceViewResolver" class  
="org.springframework.web.servlet.view.InternalResourceViewResolver">  
  <property  
name="viewClass"> <value>org.springframework.web.servlet.view.JstlView</value>  
</property>  
<property name="suffix"> <value>.jsp</value> </property>  
</bean>
```

HomeController.java

```
package com.ch.helloworld;
import java.text.DateFormat;import java.util.Date;
import java.util.Locale;import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;import
org.springframework.web.bind.annotation.RequestMapping;import
org.springframework.web.bind.annotation.RequestMethod;
/** * Handles requests for the application home page. */
@Controller
public class HomeController {
    private static final Logger logger = LoggerFactory.getLogger(HomeController.class);
    /**      * Simply selects the home view to render by returning its name.      */
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String home(Locale locale, Model model) {
        logger.info("Welcome home! The client locale is {}. ", locale);
        Date date = new Date();
        DateFormat dateFormat =
                                // 날짜 Type      시간 Type      지역
        DateFormat.getDateInstance(DateFormat.LONG, DateFormat.LONG, locale);
        String formattedDate = dateFormat.format(date);
        model.addAttribute("serverTime", formattedDate );
        return "home";
    }
}
```

HomeController.java

클래스 정의 윗부분 @Controller 애노테이션 때문에 이 클래스가 컨트롤러가 되는 것이며

@RequestMapping(value = "/",...) 부분에 의해 "/"요청이 들어오면 home method가 실행되는 것이다.

마지막 return "home"을 하게 되면

다시 이 클래스를 처음 요청한 DispatcherServlet이 받는데

디스패처서블릿은 뷰리졸버 한테 정의된 클래스에게

"home"이라는 뷰 이름을 해석시키는데

뷰리졸버는 prefix, suffix를 붙여 뷰 이름을 "/WEB-INF/view/hello.jsp"로

해석하여 해당 JSP가 실행 되는 것이다.

/WEB-INF/views/home.jsp

```
<%@ taglib uri= "http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session= "false" %>
<html>
  <head>
    <title>Home</title>
  </head>
<body>
  <h1>
    Hello world!
  </h1>

  <P> The time on the server is ${serverTime}. </P>
</body>
</html>
```

ContextLoaderListener, DispatcherServlet란?

web.xml파일에보면 ContextLoaderListener와 DispatcherServlet이 보이는데 이들의 개념에 대해서 명확히 알 필요가 있다.

우선 각각 WebApplicationContext의 인스턴스를 생성하는데 ContextLoaderListener가 생성한 컨텍스트가 ROOT Context가되고 DispatcherServlet이 생성한 인스턴스는 ROOT Context를 부모로 하는 자식 컨텍스트가 된다.

root-context.xml 을 Root Spring Container로 지정해 모든 필터와 서블릿이 공유

Spring의 DispatcherServlet은 자체가 서블릿이므로 1개 이상의 DispatcherServlet이 설정되어 기동되는 것이 가능하다. 예를 들어 큰 시스템에 인사관리, 영업관리가 있는데 각각 별도의 디스패처서블릿으로 설정하고 다른 설정 파일을 정의할 수 있다. 다음과 같은 모양이 된다.

```
<!--인사관리 -->
<servlet>
  <servlet-name>insaServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/insa-context.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```



```
<servlet-mapping>
  <servlet-name> insaServlet </servlet-name>
  <url-pattern>/insa</url-pattern>
</servlet-mapping>

<!--영업관리 -->
<servlet>
  <servlet-name> saleServlet </servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/sale-context.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name> saleServlet </servlet-name>
  <url-pattern>/sale</url-pattern>
</servlet-mapping>
```

/insa 요청이 오면 insaServlet으로 정의된 DispatcherServlet이 요청을 처리하며 설정파일로는 insa-context.xml 을 사용한다고 설정하는 것이다. 영업쪽도 마찬가지. 이 경우 인사쪽과 영업쪽은 서로 다른 Context에서 독립적으로 운영되므로 서로 공통 자바빈 같은 것을 공유할 수가 없게 되므로 추후 문제가 될 수 있다. 만약 두 컨트롤러가 공통 자바빈 등을 사용한다면 다음과 같이 ContextLoaderListener를 이용하여 정의할 수 있다. 또한 ContextLoaderListener는 contextConfigLocation을 명시하지 않으면 /WEB-INF/applicationContext.xml 파일을 설정파일로 사용한다.

```
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/_insa-context.xml /WEB-INF/_sale-context.xml</param-value>
</context-param>

<!--인사관리 -->
<servlet>
  <servlet-name>insaServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name> insaServlet </servlet-name>
  <url-pattern>/insa</url-pattern>
</servlet-mapping>

<!--영업관리 -->
<servlet>
  <servlet-name>saleServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
```

```
<servlet-mapping>  
    <servlet-name> saleServlet </servlet-name>  
    <url-pattern>/sale</url-pattern>
```

```
</servlet-mapping>
```

또한 DispatcherServlet은 Front Controller로서 서블릿 컨테이너에서 HTTP 프로토콜을 통해 들어오는 모든 요청을 프리젠테이션 계층의 제일 앞에 뒤서 처리할 수 있는 컨트롤러이다.

즉 DispatcherServlet은 서블릿 컨테이너가 생성하고 관리하는 오브젝트로서 스프링이 관여하는 오브젝트가 아니므로 직접 DI를 해줄 방법이 없고 대신 web.xml에서 설정한 웹 어플리케이션컨텍스트를 참고하여 필요한 전략을 DI하여 사용할 수 있다

XML설정 파일이 아닌 @Configuration클래스를 이용해서 설정정보를 작성했다면 contextClass를 추가 설정

```
<init-param>  
    <param-name>contextConfigLocation</param-name>  
    <param-value>spring4.chap07.config.SampleConfig</param-value>  
</init-param>  
<init-param>  
    <param-name>contextClass</param-name>  
    <param-value>  
org.springframework.web.context.support.AnnotationConfigWebApplicationContext  
    </param-value>  
</init-param>
```

문제

우측의 그림과 같이 입력하고
이름과 주소를 받아서
하단과 같이 출력하도록 하시오

참고

HomeController수정

입력 프로그램

addr.html작성

출력프로그램

addr.jsp작성

이름과 주소를 입력하세요

이름 :

주소 :

확인

홍길공님 서울 강남에 사시는군요

방가!방가!

@ExceptionHandler를 이용한 에러처리

Controller에 포함

```
@ExceptionHandler(ArithmeticException.class)
private String errhandler() {
    return "arithError";
}
```

arithError.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html> <head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    으로 나누면 안되지
</body>
</html>
```

@ControllerAdvice를 이용한 예러 처리

= 앞에서 사용되었던 ^{복사} @ExceptionHandler는 하나의 컨트롤러에서 하나의 익셉션을 처리할 때 사용되었다.

@ControllerAdvice는 하나의 컨트롤러가 아닌 여러 컨트롤러에서 하나의 익셉션을 처리할 때 사용된다,

```
package com.ch.hello;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.servlet.ModelAndView;
@ControllerAdvice
public class CommonExceptionHandler {
    @ExceptionHandler(Exception.class)
    private ModelAndView errorMode(Exception e) {
        ModelAndView mav = new ModelAndView();
        mav.addObject("ex",e);
        mav.setViewName("error_common");
        return mav;
    }
}
```

@ControllerAdvice를 이용한 예러처리

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html> <head> <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    <H2>${ex.getMessage()}</H2>
    <ul>
        <c:forEach var="st" items="${ex.getStackTrace()}">
            <li>${st.toString()}</li>
        </c:forEach>
    </ul>
</body>
</html>
```


REST

인터넷 업계는 OpenAPI의 열풍이 불고 있다. 너도나도 OpenAPI를 공개하고 있고 사용자들에게 다양한 방식의 사용을 기대하고 있다. 최근 이 OpenAPI와 함께 거론되는 기술을 당연 REST이다. 구글, 아마존, 네이버 모두가 OpenAPI를 REST 방식으로 지원한다

그렇다면 REST란 과연 어떤것일까? W3C 표준이 아님에도 불구하고 업계의 사랑을 받는 이유는 무엇일까? 이 궁금증을 풀기위해 본 포스트를 작성하고자 한다.

1) 정의

- REST는 ROA를 따르는 웹 서비스 디자인 표준이다.
- ROA : Resource Oriented Architecture

2) 주요 특징

- REST 방식의 웹서비스는 잘 정의된 Cool URI로 리소스를 표현한다.
무분별한 파라미터의 남발이 아니라, 마치 오브젝트의 멤버 변수를 따라가듯이

예를 들면 아래와 같다.

<http://www.iamcorean.net/user/mk/age/32>

기존의 서블릿을 이용한 URI는 대부분 이랬다.

<http://www.iamcorean.net/finduser.jsp?user=mk&age=32>

REST

REST 방식의 웹서비스는 세션을 쓰지 않는다는 거다.

기존의 서블릿 개발에서는 세션을 이용해서 인증 정보들을 가지고 다닌다.
또한 요청간의 전후 관련성이 생기기 때문에 한 세션의 일련의 요청은 무조건 하나의 서버가 처리해야 한다. 그래서 로드발란싱을 위해 고가의 로드발란싱 서버가 필요하게 된다.
하지만 **REST**는 세션을 사용하지 않기 때문에 각각의 요청을 완벽하게 독립적이다. 따라서 각각의 요청은 이전 요청과는 무관하게 어떠한 서버라도 처리할 수 있게 된다. 즉! 로드발란싱이 간단해 질 것이라는가? (물론 인증 관련해서는 복잡한 문제가 생긴다.)

3) ROA의 정의

+ ROA는

- 웹의 모든 리소스를 **URI**로 표현하고
- 모든 리소스를 구조적이고 유기적으로 연결하여
- 비 상태 지향적인 방법으로
- 정해진 **method**만을 사용하여 리소스를 사용하는 아키텍처 이다.

+ 이는 4가지의 고유한 속성과 연관되어 진다.

- **Addressability**
- **Connectedness**
- **Statelessness**
- **Homogeneous Interface**

+ 여기서 잠깐! 정리하자면 REST란 위에 언급한 4가지 속성을 지향하는 웹서비스 디자인 표준이다.

REST

4) RESTful 웹 서비스 속성 (ROA 속성)

+ Addressability (주소로 표현 가능함)

- 제공하는 모든 정보를 URI로 표시할 수 있어야 한다.
- 직접 URI로 접근할 수 없고 HyperLink를 따라서만 해당 리소스에 접근할 수 있다면 이는 RESTful하지 않은 웹서비스이다.

+ Connectedness (연결됨)

- 일반 웹 페이지처럼 하나의 리소스들은 서로 주변의 연관 리소스들과 연결되어 표현(Presentation)되어야 한다.
- 예를 들면,

<user>

<name>MK</name>

</user> 는 연결되지 않은 독립적인 리소스이다.

<user>

<name>MK</name>

<home>MK</home>

<office>MK</office>

</user> 는 관련 리소스(home, office)가 잘 연결된 리소스의 표현이다.

REST

+ Statelessness (상태 없음)

- 현재 클라이언트의 상태를 절대로 서버에서 관리하지 않아야 한다.
- 모든 요청은 일회성의 성격을 가지며 이전의 요청에 영향을 받지 말아야 한다.
- 세션을 유지 하지 않기 때문에 서버 로드 발란싱이 매우 유리하다.
- URI에 현재 state를 표현할 수 있어야 한다. (권장사항)

+ Homogeneous Interface (동일한 인터페이스)

- HTTP에서 제공하는 기본적인 4가지의 method와 추가적인 2가지의 method를 이용해서 리소스의 모든 동작을 정의한다.
- 리소스 조회 : GET
- 새로운 리소스 생성 : PUT, POST (새로운 리소스의 URI를 생성하는 주체가 서버이면 POST를 사용)
- 존재하는 리소스 변경 : PUT
- 존재하는 리소스 삭제 : DELETE
- 존재하는 리소스 메타데이터 보기 : HEAD
- 존재하는 리소스의 지원 method 체크 : OPTION
- 대부분의 리소스 조작은 위의 method를 이용하여 대부분 처리 가능하다. 만일 이것들로만 절대로 불가능한 액션이 필요할 경우에는 POST를 이용하여 추가 액션을 정의할 수 있다. (되도록 지양하자)

RestController

1. REST는 Representational State Transfer 약어로 URI는 하나의 고유한 리소스를 대표한다는 설계 개념.
2. 서버에 접근하는 기기가 공통으로 데이터를 처리할 수 있는 규칙
3. /boards/123은 게시물 중에서 123번 이라는 의미
4. Restful : Open API에서 많이 사용되는 REST방식으로 제공하는 외부연결 URI를 REST API이라고 하고 REST방식의 서비스 제공이 가능한 것을 Restful
5. 객체를 json으로 반환할 때 RestController

@RestController

1. 스프링 4.0부터 지원
2. JSP뷰가 아닌 REST방식의 데이터 처리 이용

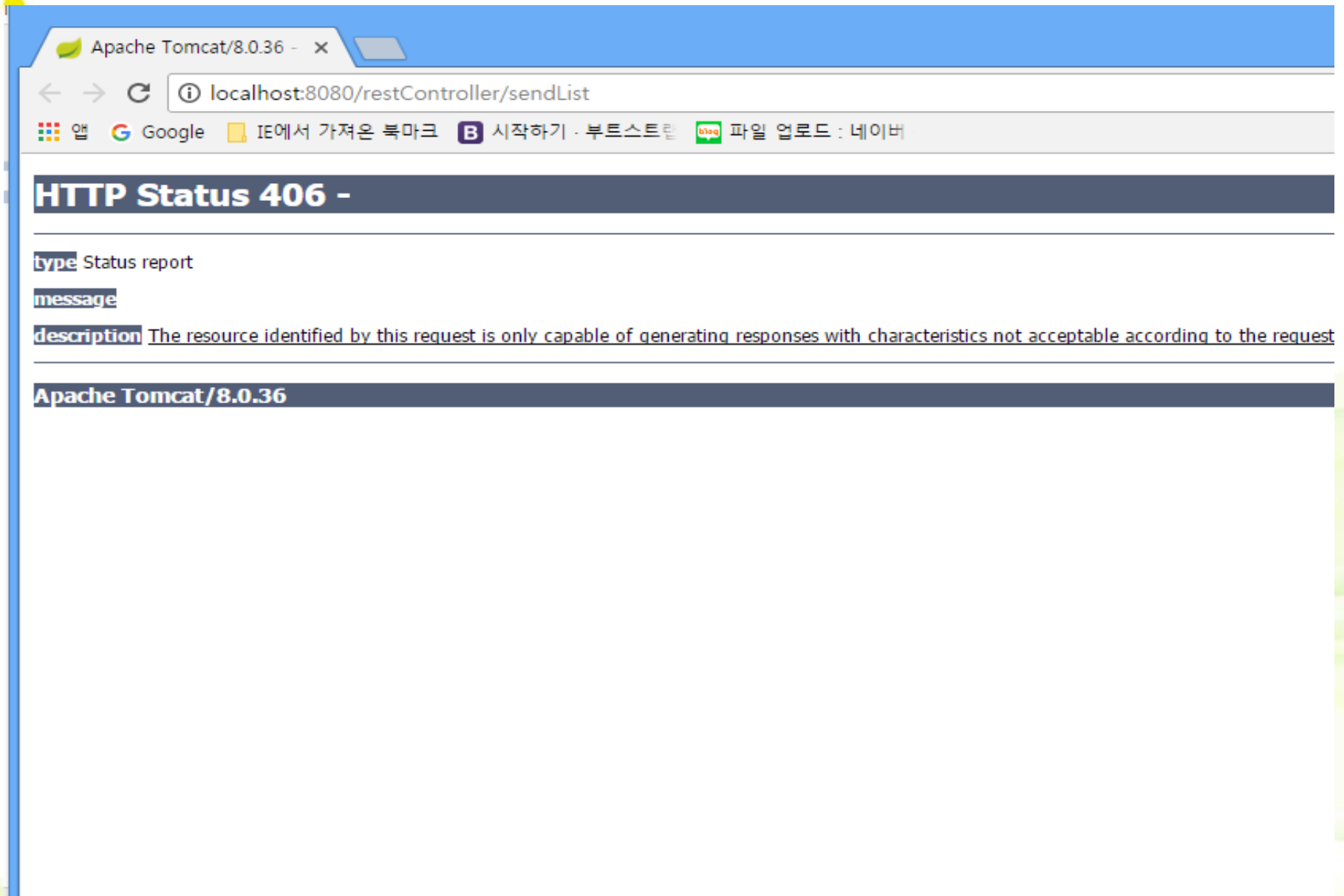
```
package test.model;
public class SampleVO {
    private Integer mno;    private String firstName;    private String lastName;
    public Integer getMno() {    return mno;    }
    public void setMno(Integer mno) {    this.mno = mno;    }
    public String getFirstName() {    return firstName;    }
    public void setFirstName(String firstName) {    this.firstName = firstName;    }
    public String getLastName() {    return lastName;    }
    public void setLastName(String lastName) {    this.lastName = lastName;    }
    public String toString() {
        return "SampleVO [mno=" + mno + ", firstName=" + firstName +
            ", lastName=" + lastName + "];"
    }
}
```

@RestController

```
package test.controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import test.model.SampleVO;
@RestController
public class SampleController {
    @RequestMapping("/sendVO")
    public SampleVO sendVO() {
        SampleVO vo = new SampleVO();
        vo.setFirstName("길동");
        vo.setLastName("홍");
        vo.setMno(123);
        return vo;
    }
}
```

결과 : {"mno":123,"firstName":"길동","lastName":"홍"}

@RestController



@RestController

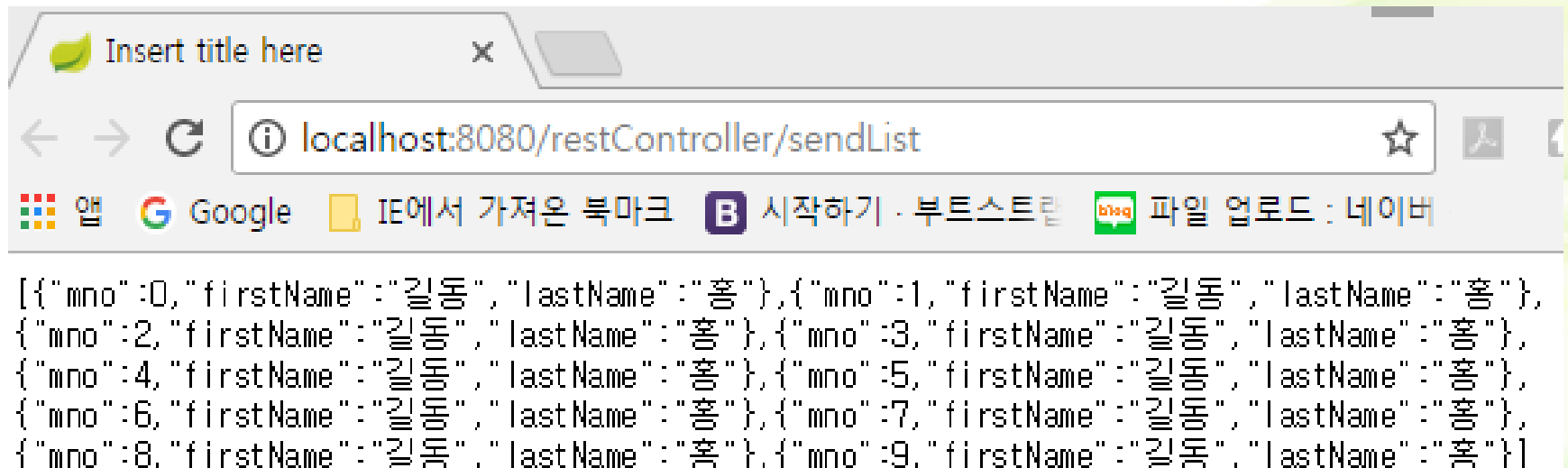
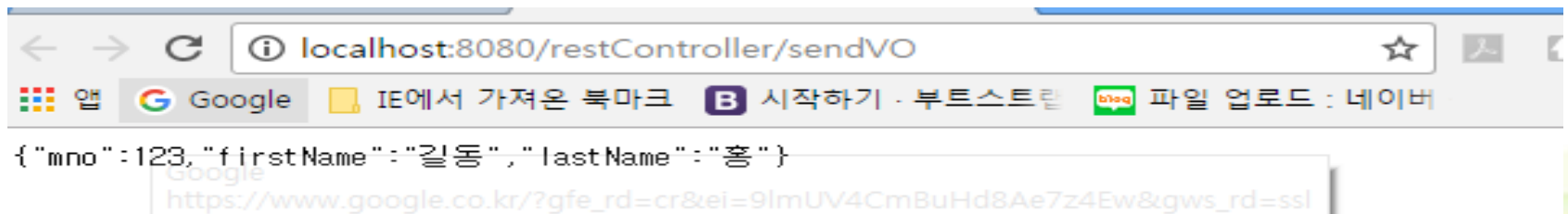
<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.5.4</version>

</dependency>



컬렉션 타입의 객체를 반환하는 경우

```
@RequestMapping("/sendList")
```

```
public List<SampleVO> sendList() {
```

```
    List<SampleVO> list = new ArrayList<>();
```

```
    for (int i = 0; i < 10; i++) {
```

```
        SampleVO vo = new SampleVO();
```

```
        vo.setFirstName("길동");    vo.setLastName("홍");
```

```
        vo.setMno(i);                list.add(vo);
```

```
    }
```

```
    return list;
```

```
}
```

결과

```
[ {"mno":0,"firstName":"길동","lastName":"홍"}, {"mno":1,"firstName":"길동","lastName":"홍"},  
  {"mno":2,"firstName":"길동","lastName":"홍"}, {"mno":3,"firstName":"길동","lastName":"홍"},  
  {"mno":4,"firstName":"길동","lastName":"홍"}, {"mno":5,"firstName":"길동","lastName":"홍"},  
  {"mno":6,"firstName":"길동","lastName":"홍"}, {"mno":7,"firstName":"길동","lastName":"홍"}, {"mno":8,"firstName":"길동",  
  "lastName":"홍"}, {"mno":9,"firstName":"길동","lastName":"홍"}]
```

@Valid 어노테이션

1. Bean Validation API(JSR 303)에 정의
2. 커맨드 객체에 검증 규칙 설정
3. LocalValidatorFactoryBean 클래스를 이용하여 JSR 303 프로바이더를 스프링 Validator로 등록
4. 컨트롤러가 두 번째에서 생성한 빈을 Validator로 사용하도록 설정

<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>5.1.3.Final</version> </dependency>

<dependency>

애너테이션 정의 ; Bean Validation

- @NotNull ; null검증
- @Max ; 지정한 수치 이하인지 검증
- @Min ; 지정한 수치 이상인지 체크
- @Size ; 지정한 범위의 크기인지 검증
- @AssertTrue @AssertFalse ; true 또는 false인지 검증
- @Pattern 정규표현과 일치여부
- @NotEmpty
- @NotBlank
- @Length @Length(max = 20)
- @Email ; 이메일 형식 여부
- @Range
- @ControlCardNumber
- @URL ; URL형식여부

• Free mail

Hot mail(MSN)이나 Yahoo! 메일 등 메일 주소를 무료로 얻을 수 있는 [서비스](#). 일반적인 [전자 우편](#)과는 달리 [웹 브라우저](#)를 사용하여 메일을 열람, 송신하는 [웹 메일](#)이라는 방법을 이용

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>5.1.3.Final</version>

</dependency>

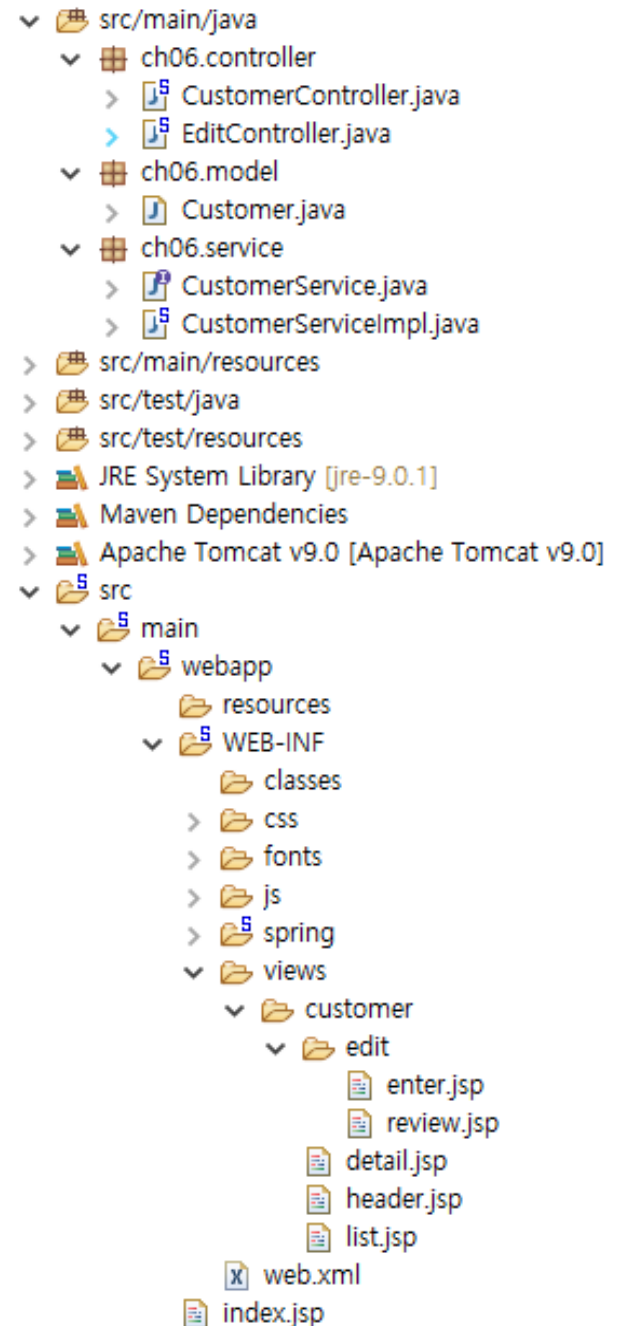
bean-biz.xml에 validator object설정

Spring Web MVC

1. 레이어와 패키지 구조

sample.customer

- biz	비즈니스 로직
- domain	도메인 Customer
- service	서비스 CustomerService
	DataNotFoundException
	MockCustomerService
- web	프리젠테이션
- controller	컨트롤러 CustomerEditController CustomerListController CustomerRestController FileUploadController



Customer

```
package ch06.model;
import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.Length;
import org.hibernate.validator.constraints.NotBlank;
public class Customer {
    private int id;
    @NotBlank
    @Length(max=20)
    private String name;
    @NotBlank
    @Length(max=100)
    private String address;
    @Email
    private String emailAddress;
    public Customer() {
    }
    public Customer(String name,String address,String emailAddress) {
        this.name = name;
        this.address=address;
        this.emailAddress = emailAddress;
    }
}
```


Customer

```
public int getId() {  
    return id;  
}  
public void setId(int id) {  
    this.id = id;  
}  
public String getName() {  
    return name;  
}  
public void setName(String name) {  
    this.name = name;  
}  
public String getAddress() {return address;}  
public void setAddress(String address) {  
    this.address = address;  
}  
public String getEmailAddress() {return emailAddress;}  
public void setEmailAddress(String emailAddress) {  
    this.emailAddress = emailAddress;  
}  
}
```

CustomerService

```
package ch06.service;  
import java.util.List;  
import ch06.model.Customer;  
public interface CustomerService {  
    List<Customer> list();  
    Customer select(int id);  
    void update(Customer customer);  
}
```

@PostConstruct 어노테이션 과 @PreDestroy 어노테이션

@PostConstruct 어노테이션 과 @PreDestroy 어노테이션은 라이프 사이클의 초기화 및 제거 과정을 제공한다.

@PostConstruct 는 의존하는 객체를 설정한 이후에 초기화 작업을 수행할 메서드에 적용되며,

@PreDestroy 어노테이션은 컨테이너에서 객체를 제거하기 전에 호출 될 메서드에 적용된다.

즉, 스프링 설정 파일에서 init-method 속성과 destroy-method 속성을 이용해 명시한 메서드와 동일한 시점에 실행된다.

```
public class HomeController{  
    @PostConstruct  
    public void init(){  
        //초기화 처리  
    }  
    @PreDestroy  
    public void close(){  
        // 자원 반환 등 종료 처리  
    }  
}
```

CustomerServiceImpl

```
package ch06.service;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javax.annotation.PostConstruct;
import org.springframework.stereotype.Service;
import ch06.model.Customer;

@Service
public class CustomerServiceImpl implements CustomerService {
    private Map<Integer, Customer> map =
        new HashMap<Integer, Customer>();
    private int nextId;
    public List<Customer> list() {
        return new ArrayList<Customer>(map.values());
    }
}
```

CustomerServiceImpl

@PostConstruct

```
public void init() {  
    regist(new Customer("길동","강남","k1@k.com"));  
    regist(new Customer("연산군","한양","k2@k.com"));  
    regist(new Customer("장녹수","대동강","k3@k.com"));  
}  
public Customer regist(Customer customer) {  
    customer.setId(++nextId);  
    map.put(customer.getId(), customer);  
    return customer;  
}  
public Customer select(int id) {  
    return map.get(id);  
}  
public void update(Customer customer) {  
    map.put(customer.getId(), customer);  
}  
}
```

CustomerController

```
package ch06.controller;
@Controller
public class CustomerController {
    @Autowired
    private CustomerService cs;
    @RequestMapping(value="/",method=RequestMethod.GET)
    public String init() {
        return "forward:/customer";
    }
    @RequestMapping(value="/customer",method=RequestMethod.GET)
    public String list(Model model) {
        List<Customer> customers = cs.list();
        model.addAttribute("customers", customers);
        return "/customer/list";
    }
    @RequestMapping("/customer/{id}")
    public String detail(@PathVariable int id, Model model) {
        Customer ct = cs.select(id);
        model.addAttribute("customer", ct);
        return "/customer/detail";
    }
}
```

@Controller 의 메소드 파라미터로 올 수 있는 것들

파라미터(Parameter)	간단 설명(simple contents)
HttpServletRequest, HttpServletResponse	서블릿 관련~
HttpSession	서블릿 관련~
WebRequest, NativeWebRequest	HttpServletRequest 의 요청정보를 대부분 그대로 갖고 있는, 서블릿 API 에 종속적이지 않은 오브젝트 타입이다.(서블릿과 포틀릿 환경 양쪽에 모두 적용 가능한 범용적인 핸들러 인터셉터를 만들 때 활용하기 위해 만들어졌다.
Locale	java.util.Locale 타입으로 DispatcherServlet 의 지역정보 리졸버(Locale Resolver)가 결정한 Locale 오브젝트를 받을 수 있다. Locale 은 예를들어 ko_KR, jp_JP(?) 암튼 한국어, 일본어, 영어 등등 이라고 보면 된다.
InputStream, Reader	HttpServletRequest 의 getInputStream() 을 통해서 받을 수 있는 콘텐츠 스트림 또는 Reader 타입 오브젝트를 제공 받을 수 있다.
OutputStream, Writer	HttpServletRequest 의 getOutputStream() 을 통해서 받을 수 있는 콘텐츠 스트림 또는 Writer 타입 오브젝트를 제공 받을 수 있다.
@PathVariable	<p>@RequestMapping의 URL에 {}로 들어가는 패스 변수(path variable)를 받는다.</p> <p>http://acet.pe.kr/post/?id=10 이라고 하는걸 http://acet.pe.kr/post/10 으로 만들 수 있다.</p> <p>ex) @RequestMapping("/post/{id}") public String view(@PathVariable</p>

@RequestParam	public String view(@RequestParam("id") int id){....}
@CookieValue	HTTP 요청과 함께 전달된 쿠키 값을 메소드 파라미터에 넣어주도록 @CookieValue 를 사용 할 수 있다.
@RequestHeader	요청 헤더정보를 메소드 파라미터에 넣어주는 어노테이션이다. 헤더정보에 있는 Server 정보, Content-Type 등의 정보를 이용하여 어디에서 접근 하는지 알아내서 view 를 다르게 해준다던지 하는 것들을 구현 할 수 가 있다.
Map, Model, ModelMap	다른 어노테이션이 붙어있지 않다면
@ModelAttribute	요청 파라미터를 메소드 파라미터에서 1:1 로 받으면 @RequestParam 이고, 도메인 오브젝트나 DTO, VO 의 프로퍼티에 요청 파라미터를 바인딩해서 한번에 받으면 @ModelAttribute 라고 볼 수 있다. ex) @RequestMapping("/acet/search") public String search(@ModelAttribute UserSearch userSearch){ List<User> list = userService.search(userSearch); model.addAttribute("userList", list); }

<p>Errors, BindingResult</p>	<p>변환작업 시 오류가 있을 때 @ModelAttribute를 사용 했을 때는 @RequestParam과는 달리 HTTP 400 이 발생하지 않고 작업은 계속 진행 되며, 단지 타입 변환 중에 발생한 예외가 BindException 타입의 오브젝트에 담겨서 컨트롤러로 전달 된다. 즉, 예외처리에 대한 기회를 주어야 하므로 org.springframework.validation.Errors or org.springframework.validation. BindingResult 타입의 파라미터를 같이 사용 한다.</p> <p>ex) @RequestMapping(value="add", method=RequestMethod.POST)</p> <p>public String add(@ModelAttribute User user, BindingResult bindingResult) { }</p> <p>BindingResult 대신 Errors를 사용해도 되며, 사용시 주의 사항으로 는 반드시 @ModelAttribute 뒤에 나와야 한다.@ModelAttribute 검증작업에서 나온 오류만을 전달하기 때문이다.</p>
<p>SessionStatus</p>	<p>컨트롤러가 제공하는 기능 중에 모델 오브젝트를 세션에 저장했다가 다음 페이지 에서 다시 활용하게 해주는 기능이 있다.</p> <p>org.springframework.web.bind.support.SessionStatus 오브젝트이며, 필요없어지면 확실하게 제거해줘야 한다.</p>
<p>@RequestBody</p>	<p>Http 요청의 본문(body) 부분이 그대로 전달 된다.</p> <p>일반적인 GET/POST의 요청 파라미터라면 사용 할 일이 없다.</p> <p>XML이나 JSON 기반의 메시지를 사용하는 경우에는 이 방법이 매우 유용하다.</p> <p>또한 메시지컨버터가 필요하다</p>

@Value	<p>SpEL을 이용해 클래스의 상수를 읽어오거나 특정 메소드를 호출한 결과값, 조건식 등을 넣을 수 있다.</p> <p>ex) @RequestMapping(...) public String hello(@Value("#{systemProperties['os.name']}") String osName){...} 필드 주입이라는 것도 있다.</p>
@Valid	<p>@Valid는 JSR-303의 빈 검증기를 이용해서 모델 오브젝트를 검증하도록 지시하는지시자다.</p> <p>서버 Valid는 나오며, @Validate도 있다!~</p>

※ Model Interface (<http://docs.spring.io/spring/docs/3.2.x/javadoc-api/> 참조)

Model(모델) 오브젝트는 정보를 보여주는 View에 의해 사용될 정보들을 제공한다.

이러한 오브젝트들은 Spring Framework에서 추상화로 제공된다. 어떤 종류의 View 기술도 프레임워크에 쉽게 플러그인 될 수 있다. 예를 들자면 excel, pdf, xslt, Free maker, html, velocity 등 웹 프레임워크를 지원한다. 모델 오브젝트(org.springframework.ui.ModelMap에서 제공)는 정보를 저장하기 위해 내부적으로 Map으로 유지된다.

org.springframework.ui에 있는 Interface Model

Model addAttribute (String name, Object value)	value 객체를 name 이름으로 추가
Model addAttribute (Object value)	value 를 추가. value 의 패키지 이름을 제외한 단순 클래스 이름을 모델 이름으로 사용(첫 글자는 소문자). value 가 배열이거나 콜렉션인 경우 첫 번째 원소의 클래스 이름 뒤에 " List "를 붙인 걸 모델 이름으로 사용(첫 글자는 소문자)
Model addAllAttributes (Collection<?> values)	addAttribute(Object value) 메소드를 이용해서 콜렉션에 포함된 객체들을 차례대로 추가
Model addAllAttributes (Map<String, ?>, attributes)	Map 에 포함된 <키, 값>에 대해 키를 모델 이름으로 사용해서 값을 모델로 추가
Model mergeAttributes (Map<String, ?>, attributes)	Map 에 포함된 <키, 값>을 현재 모델에 추가. 단 키와 동일한 이름을 갖는 모델 객체가 존재하지 않는 경우에만 추가
boolean containsAttributes (String name)	지정한 이름의 모델 객체를 포함하고 있는 경우 true 리턴

@SessionAttributes 이 어노테이션은 스프링에서 상태유지를 위해 제공되는 어노테이션인데 대충 객체의 위치가 뷰와 컨트롤러의 사이에 존재한다고 생각하면 좋다.

우선 @SessionAttributes는 항상 클래스 상단에 위치하며 해당 어노테이션이 붙은 컨트롤러는 @SessionAttributes("세션명")에서 지정하고 있는 "세션명"을 @RequestMapping으로 설정한 모든 뷰에서 공유하고 있어야 한다는 규칙을 갖고 있다. 예를 들어 @SessionAttributes("command") 라는 어노테이션이 붙은 클래스라면 하위의 종속되어있는 모든 뷰가 "command"라는 모델 값을 공유하고 있어야 한다는 것이다. 만약 이 조건을 충족하지 못하면 다음과 같은 에러가 발생하게 된다.

org.springframework.web.HttpSessionRequiredException:**Expected session attribute 'command'**

```
RequestMapping(value="/", method=RequestMethod.GET)
public String home(Model model) {
    model.addAttribute("command", new Command());
}
```

// 이런 방식으로 SessionAttributes를 이용하는 것은 옳지 않다.

이 문제를 해결하기 위해 사용할 수 있는 방법은 2가지 인데 첫째는 해당 컨트롤러에서 맨 처음 읽어들이는 것으로 예상되는 뷰의 Model 객체를 통해 수동적으로 "command"란 파라미터를 보내주는 것이다. 이 방식은 클라이언트가 해당 클래스로 뷰어에 접근할 때 반드시 첫번째로 해당 뷰를 통해야만 한다는 제약조건을 갖게 되며 그렇지 않을 경우 또다시 위의 에러가 발생할 수 있으므로 결코 추천할 수 없는 방식이다.

@SessionAttributes("command")

@Controller

public class Controller {

```
    @ModelAttribute("command")  
    public Command command() {  
        return new Command();  
    }
```

```
    @RequestMapping(value="/", method=RequestMethod.POST)  
    public String home(@ModelAttribute Command command) {  
        ...  
    }
```

```
}
```


에러를 보고 싶지 않다면 @ModelAttribute를 붙인 메서드를 이용할 것을 권장한다. 예를 보면 @ModelAttribute가 붙은 command()메서드를 볼 수 있는데 이 메서드는 해당 컨트롤러로 접근하려는 모든 요청에 @ModelAttribute가 붙은 메서드의 리턴 값을 설정된 모델명으로 자동 포함해주는 역할을 담당해준다. 물론 이미 동일한 이름의 모델이 생성 되어있다면 위의 메서드 값은 포함되지 않으며 오로지 설정한 모델명과 일치하는 객체가 존재하지 않는 경우에만 메서드의 리턴 값을 서버의 응답과 함께 클라이언트에게 전송하는 역할을 담당한다.



← → ↺ 🏠 localhost:8080/Spring/

Java Design Spring MVC

Hello world!

The time on the server is 2012년 3월 12일 (월) 오전 3

Happenstantial 사랑해요LG

해당 URL과 매핑되는 @RequestMapping에는 위와 같은 데이터를 내보낸 적이 없지만 \${command.username}, \${command.password} 값이 위와 같이 발생한다.

```
import com.billions.spring.domain.Command;

/**
 * Handles requests for the application home page.
 */
@Controller
public class HomeController {

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String home() {
        return "home";
    }

    @ModelAttribute("command")
    public Command command() {
        Command command = new Command();
        command.setUsername("Happenstantial");
        command.setPassword("사랑해요LG");
        return command;
    }
}
```


해당 컨트롤러로 클라이언트가 접근할 때 반드시 @ModelAttribute가 붙은 메서드의 리턴 값을 보장받는 다는 소리다. 지금은 단순히 return new Command(); 정도로 마무리 지었지만 원한다면 해당 객체에 기본 값을 포함할 수도 있다.

@SessionAttributes의 기본 충족조건을 이해했으므로 이제 사용 용도에 대해 조금 생각해보자. 필자가 생각하는 @SessionAttribute의 사용 용도는 다음과 같다.

1. 스프링에서 제공하는 form 태그라이브러리를 이용하고 싶을 때.
2. 몇 단계에 걸쳐 완성되는 폼을 구성하고 싶을 때
3. 지속적으로 사용자의 입력 값을 유지하고 싶을 때

아마 첫번째 이유가 가장 절실할 것 같다. 스프링에서 제공하는 폼태그를 자주 활용하는데 이 태그라이브러리를 활용하면 폼 작성이 정말 쉬워지는데다 검증 바인딩 기술은 아주 쉽다

@SessionAttributes는 해당 어노테이션에 설정한 값과 동일한 이름의 모델객체를 발견하면 이를 캐치하여 세션 값으로 자동 변경시켜준다. 그리고 해당 모델객체가 세션 값으로 대체되면 앞으로 세션 값을 지우기 전까지 해당 이름의 모델명 호출할 때 세션에 저장된 값을 불러오게 된다

@Controller

@SessionAttributes("command")

```
public class Controller {  
    @ModelAttribute("command")  
    public Command command() {  
        return new Command();  
    }  
    @RequestMapping(value="/", method=RequestMethod.GET)  
    public String home() {  
        return "home";  
    }  
    @RequestMapping(value="/", method=RequestMethod.POST)  
    public String home(Model model, @ModelAttribute Command command) {  
        model.addAttribute("command", command);  
        return "home";  
    }  
}
```

위와 같은 소스를 예로 들어 설명한다면 "/"란 경로로 POST 방식을 통해 클라이언트가 파라미터를 보낼 경우 서버는 해당 값을 세션에 저장되어 있는 **"command"객체에 저장시켜 해당 세션을 종료하기 전까지 값을 유지해준다.**

이제 세션 값이 더이상 필요 없어질 경우 이를 지우는 방법도 알아야 하겠다. 세션 값을 제때 지우지 않고 계속 쌓아둔다면 메모리에 무리가 생길 수 있으므로 불필요해질 경우 제거해주는 것도 중요하다. 제거 법은 매우 간단한데

```
@RequestMapping(value="/", method=RequestMethod.POST)
public String home(Model model, @ModelAttribute Command command,
SessionStatus session) {
    model.addAttribute("command", command);
    session.setComplete();
    return "home";
}
```

위와 같이 종료가 필요한 URL매핑 메서드에 SessionStatus란 세션관리 인자를 전달받아 종료시켜주면 된다.

2. 화면표시

- 1) @RequestMapping 애노테이션 ; URI와 Controller메소드 매핑하는 설정
 - URL/customer을 매핑할 때
@RequestMapping("/customer") 또는 @RequestMapping(value="/customer")
 - value ; URL @RequestMapping(value={"/foo", "bar"})
 - method ; HttpMethod @RequestMapping(method=HttpMethod.POST)
 - params ; 요청파라미터 @RequestMapping(params="action=new")
 - headers ; 요청헤더 @RequestMapping(headers="myHeader=myValue")
 - consumes ; 요청미디어 타입 @RequestMapping(consumes="text/html")
 - produces ; 응답미디어타입 @RequestMapping(produces="text/html")

```
import static org.springframework.web.bind.annotation.RequestMethod.GET;
@Controller
public class CustomerListController {
    .....
    @RequestMapping(value = "/customer", method = GET)
    public String showAllCustomers(Model model) {
        return "customer/list";
    }
    .....
}
```

```
@RequestMapping(value = "/enter", params = "_event_proceed", method = POST)
    public String verify(@Valid @ModelAttribute("editCustomer") Customer customer,
                        Errors errors) {
        if (errors.hasErrors()) {
            return "customer/edit/enter";
        }

        return "redirect:review";
    }
```

```
@RequestMapping(value = "/review", params = "_event_confirmed", method = POST)
    public String edit(@ModelAttribute("editCustomer") Customer customer,
                      RedirectAttributes redirectAttributes, SessionStatus sessionStatus)
        throws DataNotFoundException {
        customerService.update(customer);
        // retrun "redirect:edited";
        redirectAttributes.addFlashAttribute("editedCustomer", customer);

        sessionStatus.setComplete();

        return "redirect:/customer";
    }
```

3. REST(Representational State Transfer) : URI로 웹상의 리소스 특징

@RequestMapping(value = "/{customerId}", method = *GET*)

```
public ResponseEntity<Customer> findById(@PathVariable int customerId)  
throws DataNotFoundException {  
    Customer customer = customerService.findById(customerId);  
    HttpHeaders headers = new HttpHeaders();  
    headers.setContentType(  
        new MediaType("text", "xml", Charset.forName("UTF-8")));  
    headers.set("My-Header", "MyHeaderValue");  
return new ResponseEntity<Customer>(  
        customer, headers, HttpStatus.OK);  
}
```

복수

@RequestMapping(value = "/company/{companyId}/user/{userId}", method = *GET*)

사용자 Id는 a부터 z사이 문자열이 한글자 이상일때만 메소드 사용

@RequestMapping(value = "user/{userId:[a-z]+}", method = *GET*)

4. Controller 메소드의 인수

- Model 오브젝트

- URI 템플릿 변수 (@PathVariable 애노테이션을 설정한 변수)

URI 템플릿 형식으로 지정한 URL의 변수값 : *foo(@PathVariable("userId") int userId)*

HTTP 요청 파라미터값 ; *foo(@RequestParam("userId") int userId)*

업로드 파일 : *foo(@RequestParam("uploadFile") MultipartFile uploaded)*

HTTP 요청의 헤더값 ; *foo(@RequestHeader("User-Agent") String userAgent)*

쿠키값 ; *foo(@Cookie("jsessionid") String sessionId)*

HTTP 요청의 메시지 바디 ; *foo(@RequestBody User user)*

HttpEntity 오브젝트 ; *foo(HttpEntity<User> user)*

Model 오브젝트 ; *foo(Model model)*

ModelAttribute 오브젝트 ; *foo(ModelAttribute("editedUser") User user)*

Session 관리 오브젝트 ; *foo(SessionStatus sessionStatus)*

오브젝트 ; *foo(Errors errors)*

WebRequest 오브젝트 ; *foo(WebRequest req)*

Servlet API의 각종 오브젝트 ; *foo(HttpServletRequest req, HttpServletResponse res)*

로케일 ; *foo(Locale local)*

요청 응답에 액세스 위한 스트림 오브젝트 ; *foo(Reader reader, Writer writer)*

인증 오브젝트 ; *foo(Principal principal)*

5. Controller 메소드 반환값

- View이 이름
- View이름의 접두사
 - . redirect 지정된 URL로 리다이렉트
 - redirect:/user
 - redirect:http://www.springsource.org/
 - . forward
 - forward:/user

@PathVariable

```
@RequestMapping(value={"/campaigns/delete/{id}"},  
method=RequestMethod.POST)
```

```
@PathVariable long id
```

```
@PathVariable(value = "id") long resellerId,
```

: 파라메타로 받은 {id}를 변수화

: id에 대입

: id로 받아서 resellerId에 대입

@RequestParam 과 @PathVariable 비교

1. @RequestParam

"/board/list_controller?currentPage=1"형태의 요청을 처리

```
@RequestMapping(value="/board/list_controller")
```

```
public String getAllBoards(@RequestParam(value="currentPage", required=false,  
defaultValue="1")
```

```
int currentPage,Model model){
```

```
    model.addAttribute("list",boardService.selectAll(currentPage));
```

```
    return "board_list";
```

```
}
```

2. @RequestMapping

"/board/list_controller/1"형태의 요청을 처리

```
@RequestMapping(value="/board/list_controller/{currentPage}")  
public String getAllBoards(@PathVariable(value="currentPage") int  
currentPage, Model model){  
    model.addAttribute("list",boardService.selectAll(currentPage));  
    return "board_list";  
}
```

3. 두개이상의 파라미터 처리

/board/list_controller/1/test/루피

```
@RequestMapping(value="/board/list_controller/{currentPage}/test/{name}")  
public String getAllBoards(@PathVariable(value="currentPage") int currentPage,  
    @PathVariable(value="name") String name, Model model){  
  
    return "view페이지";  
}
```

EditController.java

```
package ch06.controller;
@Controller
@RequestMapping("/customer/{id}")
@SessionAttributes("editCustomer")
public class EditController {
    @Autowired
    private CustomerService cs;

    @RequestMapping(value="/edit",method=RequestMethod.GET)
    public String edit(@PathVariable int id, Model model) {
        Customer ct = cs.select(id);
        model.addAttribute("editCustomer", ct);
        return "redirect:enter";
    }
    @RequestMapping(value="/enter",method=RequestMethod.GET)
    public String enter(@ModelAttribute("editCustomer") Customer
                        customer) {
        return "customer/edit/enter";
    }
}
```

```
@RequestMapping(value="/enter",params="_event_proceed")
public String confirm(@Valid @ModelAttribute("editCustomer")
    Customer customer, Errors errors) {
    if (errors.hasErrors()) return "customer/edit/enter";
    else return "redirect:review";
}
```

```
@RequestMapping(value="/review",method=RequestMethod.GET)
public String showReview(@ModelAttribute("editCustomer") Customer
    customer) {
    return "customer/edit/review";
}
```

```
@RequestMapping(value = "/review", params = "_event_revise",
    method=RequestMethod.POST)
public String revise() {
    return "redirect:enter";
}
```

```
@RequestMapping(value = "/review", params = "_event_confirmed",
    method=RequestMethod.POST)
public String edit(@ModelAttribute("editCustomer") Customer customer,
    RedirectAttributes redirectAttributes, SessionStatus sessionStatus){
    cs.update(customer);
    /*redirectAttributes.addFlashAttribute("editedCustomer", customer); */
    sessionStatus.setComplete();
    return "redirect:/customer";
}
}
```

header.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="spring"
    uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form" %>
<c:set var="path" value="${pageContext.request.contextPath }"/>
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link href="${path}/css/bootstrap.min.css" rel="stylesheet">
<script src="${path}/js/jquery.js"></script>
<script src="${path}/js/bootstrap.min..js"></script>
<style>.err { color: red; font-size: 20px; } </style>
```

list.jsp

```
<%@ include file= "header.jsp" %>
<body>
<div class= "container" align="center">
  <h2 class= "text-primary">고객 목록</h2>
  <table class= "table table-hover">
    <tr> <th>아이디</th> <th>이름</th> <th>주소</th> <th>이메일</th>
    <th>상세</th> <th>편집</th> </tr>
    <c:forEach var= "ct" items= "${customers }">
      <tr> <td> ${ct.id }</td> <td> ${ct.name }</td>
      <td> ${ct.address }</td> <td> ${ct.emailAddress}</td>
      <td> <!-- <c:url var="url1" value="/customer/${ct.id}"/>
      <a href= "${url1}">상세</a> --%>
      <a href= "${path}/customer/${ct.id}">상세</a> </td>
      <td> <c:url var= "url2" value= "/customer/${ct.id}/edit"/>
      <a href= "${url2}">편집</a> </td> </tr>
    </c:forEach>
  </table>
</div>
</body>
</html>
```


detail.jsp

```
<%@ include file= "header.jsp" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html> <head> <meta http-equiv= "Content-Type" content= "text/html;
charset=UTF-8">
<title>Insert title here</title> </head> <body>
<div class= "container" align= "center">
    <h3 class= "text-primary">고객 상세정보</h3>
    <table class= "table table-striped">
        <tr> <th>아이디</th> <td>${customer.id }</td> </tr>
        <tr> <th>이름</th> <td>${customer.name }</td> </tr>
        <tr> <th>주소</th> <td>${customer.address }</td> </tr>
        <tr> <th>이메일</th> <td>${customer.emailAddress }</td> </tr>
        <tr> <th colspan= "2"> <c:url var= "url" value= "/customer"/>
        <a href= "${url}"> 목록</a> </th> </tr>
    </table>
</div>
</body>
</html>
```

edit/enter.jsp

```
<%@ include file = "../header.jsp" %>
<body> <div class = "container" align = "center">
    <h2 class = "text-primary">고객내용 변경</h2>
    <form:form modelAttribute = "editCustomer">
        <table class = "table table-striped">
            <tr> <th>이름</th> <td> <form:input path = "name"/>
                <form:errors path = "name"> </form:errors> </td> </tr>
            <tr> <th>주소</th> <td> <form:input path = "address"/>
                <form:errors path = "address"> </form:errors> </td> </tr>
            <tr> <th>이메일</th> <td> <form:input path = "emailAddress"/>
                <form:errors path = "emailAddress"> </form:errors> </td> </tr>
            <tr> <th colspan = "2"> <button type = "submit"
                name = "_event_proceed" value = "process">다음</button> </th>
            </tr>
        </table>
    </form:form>
</div>
</body>
</html>
```

edit/review.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ include file="../header.jsp" %>
<body> <div class="container">
    <h1 class="text-primary">확인 화면</h1>
    <form method="post">
        <table class="table table-hover">
            <tr> <th>이름</th> <td>${editCustomer.name}</td> </tr>
            <tr> <th>주소</th> <td>${editCustomer.address}</td> </tr>
            <tr> <th>이메일 주소</th>
                <td>${editCustomer.emailAddress}</td> </tr>
        </table>
        <button type="submit" name="_event_confirmed">갱신</button>
        <button type="submit" name="_event_revise">재입력</button>
    </form>
</div> </body>
</html>
```

오류메세지 ; message.properties

- errors.datanotfound.customer=지정된 ID고객이 없습니다
errors.ngemail=이 메일은 주소는 사용할 수 없습니다.
- org.hibernate.validator.constraints.NotBlank.message = {0}은 필수입니다.
- org.hibernate.validator.constraints.Length.message = {0}은 {max}이하여야 합니다.
- org.hibernate.validator.constraints.Email.message = {0}은 올바른 형식으로 입력해야 합니다.
- name = 이름
- address = 주소
- emailAddress = 이메일

1. 스프링 MVC와 REST(Representational State Transfer)

- 웹상의 정보를 하나의 리소스로 파악하고 그 식별자로서 URI를 할당하게 고유하게 특정할 수 있음
- 예를들어 uri로 <http://foo.bar.baz>도메인으로 복수의 사용자 정보관리 <http://foo.bar.baz/user/{사용자 ID}>를 할당 즉 /user/1, usr/2, usr/3 등

구성요소	개요
DispatcherServlet	브라우저로부터 송신된 Request를 일괄적으로 관리한다.
HandlerMapping	RequestURL과 Controller 클래스의 맵핑을 관리한다. (묵시적 사용)
Controller	비즈니스 로직을 호출하여 처리 결과의 ModelAndView 인스턴스를 반환한다.
ViewResolver	Controller 클래스로부터 반환된 View명을 기본으로 이동처가 되는 View 인스턴스를 해결한다. (묵시적 사용)
View	프레젠테이션층으로의 출력 데이터를 설정한다. (묵시적 사용)

2. 비즈니스 로직의 Bean정의 파일 ; beans-biz.xml

```
<?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:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.1.xsd">
<!-- 사용할 서비스를 자동으로 DI컨테이너에 등록하는 설정 -->
    <context:component-scan base-package="sample.customer.biz.service"/>
<!-- 빈의 검증 설정-->
    <bean id="validator"
        class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
        <property name="validationMessageSource" ref="messageSource"/>
    </bean>
<!-- 메시지 관리 설정 -->
    <bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="basename" value="classpath:/META-INF/messages"/>
    </bean>
</beans>
```


3. 스프링 MVC의 Bean정의 파일 ; beans-webmvc.xml

```
<?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:context="http://www.springframework.org/schema/context"
xmlns:mvc="http://www.springframework.org/schema/mvc"
xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.1.xsd
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd">
<!-- Controller클래스를 자동으로 DI컨테이너에 등록하는 설정 -->
<context:component-scan base-package="sample.customer.web.controller"/>
<!-- Spring MVC애니메이션 이용 실적 -->
<mvc:annotation-driven
    validator="validator">
    <!-- for REST API
    <mvc:message-converters>
        <bean
class="org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConverter"/>
    </mvc:message-converters>
    -->
</mvc:annotation-driven>
```


<!-- Static Resource설정 DispatcherServlet을 경유해 정적 리소스 파일(html, 이미지, css, javascript파일 등)에 액세스하기 위한 설정 -->

```
<mvc:resources mapping="/resources/**" location="/WEB-INF/resources/" />
```

```
<bean id="multipartResolver"
```

```
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>
```

<!-- View설정-->

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

```
    <property name="prefix" value="/WEB-INF/views/" />
```

```
    <property name="suffix" value=".jsp" />
```

```
</bean>
```

```
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
```

```
    <property name="exceptionMappings">
```

```
        <props>
```

```
            <prop key="java.lang.Exception">error</prop>
```

```
        </props>
```

```
    </property>
```

```
</bean>
```

```
</beans>
```

정적 리소스 파일(html, 이미지, css, javascript파일 등)에 액세스 설정

```
<mvc:resources mapping="/image/**" location="/WEB-INF/image/" />
```

```
<mvc:resources mapping="/css/**" location="/WEB-INF/css/" />
```

```
<mvc:resources mapping="/js/**" location="/WEB-INF/js/" />
```

4. 뷰 리졸버 설정 ; beans-webmvc.xml

```
<bean class="org.springframework.web.servlet.view.ResourceViewResolver">
    <property name="basename" value="/WEB-INF/spring/views/" />
    <property name="order" value="0" />
</bean>
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
    <property name="order" value="1" />
</bean>
```

1. *ResourceViewResolver* 와 *InternalResourceViewResolver*을 설정
2. *order*에 0, 1을 설정하여 *DispatcherServlet*이 View Object를 가져올 때 우선 순위 지정
3. *InternalResourceViewResolver*는
접두사, 접미사를 설정할 수 있음

HTML 특수 문자 처리 방식 설정

- JSP를 뷰 기술로 사용 - 커스텀 태그를 이용해서 메시지 출력.

```
<title> <spring:message code="login.form.title"/> </title>
```

- 커스텀 태그 출력 값이 '<입력폼>'일 경우 '<'와 '>'는 특수문자이므로 '<'나 '>'와 같은 엔티티 레퍼런스로 변환해주어야 함.

- 스프링에서의 특수 문자 처리

- defaultHtmlEscape 컨텍스트 파라미터를 통해서 지정.

```
<context-param>
```

```
    <param-name>defaultHtmlEscape</param-name>
```

```
    <param-value>false</param-value>
```

```
</context-param>
```

```
...
```

```
</web-app>
```

- defaultHtmlEscape 컨텍스트 파라미터 값

- true 로 지정 : 스프링이 제공하는 커스텀 태그나 Velocity 매크로는 HTML의 특수 문자의 엔티티 레퍼런스로 치환.
- false 로 지정 : 특수 문자를 그대로 출력.
- 기본값 : true

5. web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
  <display-name>Sample MVC</display-name>
  <context-param> 5)
    <param-name>defaultHtmlEscape</param-name>
    <param-value>true</param-value>
  </context-param>
  <context-param> 1)
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:/META-INF/spring/beans-biz.xml</param-value>
  </context-param>
  <listener> 2)
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <filter> 6)
    <filter-name>characterEncodingFilter</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>characterEncodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
<servlet> 3)
  <servlet-name>dispatcherServlet</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:/META-INF/spring/beans-webmvc.xml
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping> 4)
  <servlet-name>dispatcherServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```



- 1)과 2) SMS DI컨테이너를 웹컨테이너 상에 작성하는 컨테이너
 - 2)는 컨테이너를 웹컨테이너 상에 작성하는 리스너
 - 1)은 어느 Bean정의 파일을 바탕으로 DI컨테이너를 작성할지 지정하는 파라미터
- 3)과 4)는 스프링의 중심이 되는 DispatcherServlet정의
 - 3)에서 DispatcherServlet을 정의하고
 - 4)에서 DispatcherServlet 경로 설정
- 5)는 스프링 MVC뷰에서 html의 이스케이프 실시에 대한 설정 jsp로 말하면 defaultHtmlEscape를 true로 설정하고 스프링 MVC가 준비한 태그를 사용해 데이터를 표시하면 < , > 등의 문자를 이스케이프 처리해 준다
- 6)은 CharacterEncodingFilter설정
 - 이 Filter를 설정해 두면 자동으로 HttpServlet의 setCharacterEncoding메소드를 실행해 적절한 문자코드를 지정한다

mvc:annotation-driven

<mvc:annotation-driven>은 애노테이션 방식의 @MVC를 사용시 필요한 몇 가지 빈들을 자동으로 등록해 줍니다. 자동으로 등록되는 빈이 어떤 것인지 기억해두고 이를 다시 <bean>태그로 등록하지 않게 해야 합니다. AnnotationMethodHandlerAdapter와 DefaultAnnotationHandlerMapping등의 설정을 변경해야 할 때는 <mvc:annotation-driven>을 사용할 수는 없고 이때는 직접 필요한 빈을 등록하고 프로퍼티를 설정해줘야 합니다.

아래는 자동으로 등록되는 빈들의 내용이고 몰라도 일단 사용상의 문제는 없겠습니다.

1. DefaultAnnotationHandlerMapping

@RequestMapping을 이용한 핸들러 매핑 전략 등록, 이것이 가장 우선이 되므로 다른 핸들러 매핑은 자동등록 되지 않는다.

2. AnnotationMethodHandlerAdapter

디폴트 핸들러 어댑터, 이것이 가장 우선이 되므로 다른 핸들러 어댑터는 자동등록 되지 않는다.

3. ConfigurableWebBindingInitializer

모든 컨트롤러 메소드에 자동으로 적용되는 WebDataBinder 초기화용 빈을 등록하고 AnnotationMethodHandlerAdapter의 프로퍼티로 연결해 준다.

기본 등록 빈은 FormattingConversionServiceFactoryBean, LocalValidatorFactoryBean 이고 LocalValidatorFactoryBean기능이 적용되려면 JSR-303지원 라이브러리가 클래스패스에 등록 되어 있어야 한다.

4. 메시지 컨버터

네개의 디폴트 메시지 컨버터와 Jaxb2RootElementHttpMessageConverter, MappingJacksonHttpMessageConverter가 추가로 등록된다 (JAXB2와 Jackson 라이브러리를 클래스패스에 있어야만 등록됨)

5. <spring:eval>을 위한 컨버전 서비스 노출용 인터셉터

기본적으로 표준 컨버터를 이용해서 모델의 프로퍼티 값을 JSP에 출력할 문자열로 변환 하지만 <mvc:annotation-driven>을 등록해주면 ConfigurableWebBindingInitializer에 등록되는 것과 동일한 컨버전 서비스를 인터셉터를 이용해서 <spring:eval>태그에서 사용할 수 있게 해준다.

6. validator

자동등록되는 ConfigurableWebBindingInitializer의 validator 프로퍼티에 적용할 Validator 타입의 빈을 직접 지정할 수 있다.

```
<mvc:annotation-driven validator="myValidator"/>
<bean id="myValidator" class="MyLocalValidatorFactoryBean">
    // property 설정
</bean>
```

7. conversion-service

자동등록되는 ConfigurableWebBindingInitializer의 conversionService 프로퍼티에 설정될 빈을 지정할 수 있다.

```
<mvc:annotation-driven conversion-service="myConversionService"/>
<bean id="myConversionService" class="FormattingConversionServiceFactoryBean">
    <property name="converters">
        ...
    </property>
</bean>
```