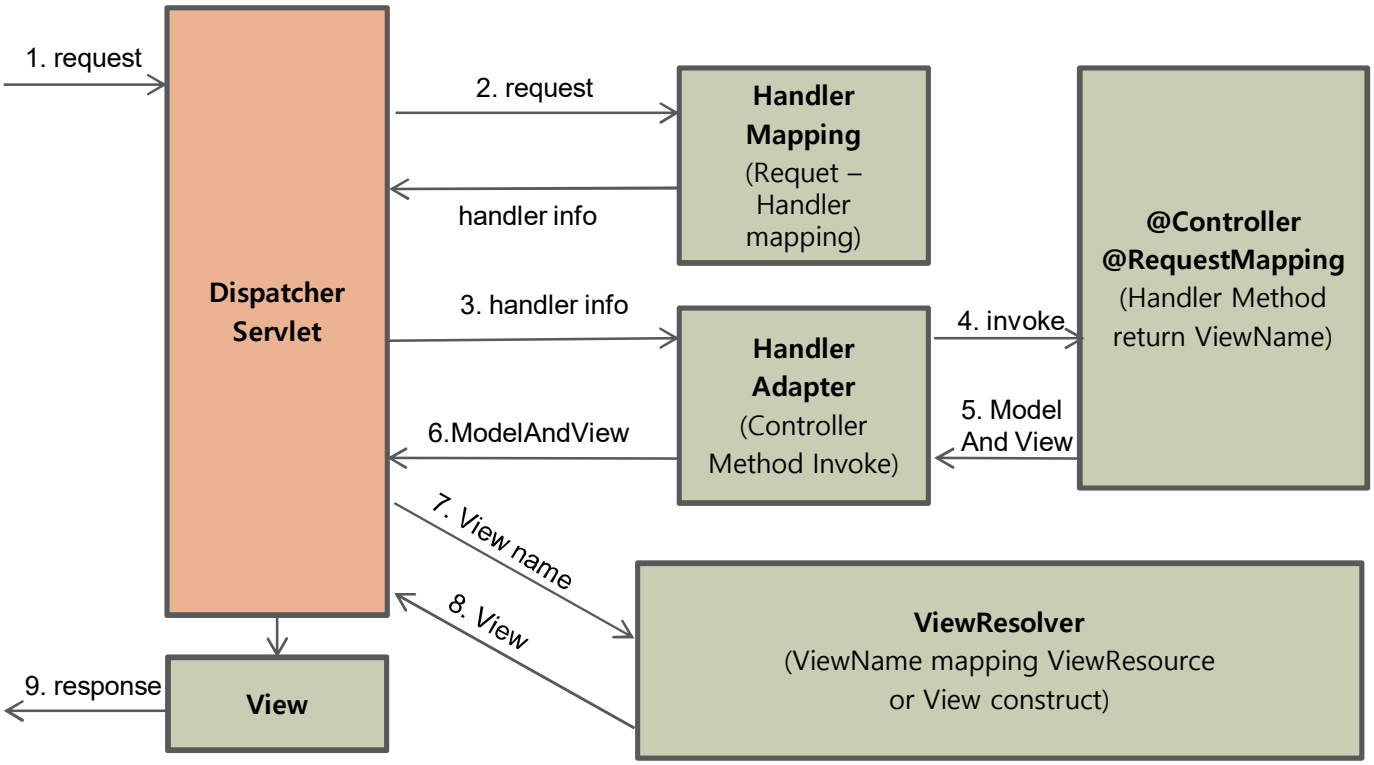
Spring Framework

Spring MVC

대덕인재개발원 : 최희연

❑ Spring MVC Architecture



❑ DispatcherServlet

- 클라이언트의 요청을 받아들이는 entry point(Front Controller)
- 하나의 요청을 처리하기 위하여 필요한 클래스들의 중계를 담당

❑ HandlerMapping

- 요청된 URL을 분석하여 해당 컨트롤러를 구함
- 해당 컨트롤러를 DispatcherServlet 에게 반환

❑ HandlerAdapter

- Handler mapping을 통해 확보한 핸들러의 호출을 담당.
- 핸들러 메소드의 파라미터를 준비하고, 핸들러 메소드의 리턴값에 대한 후처리를 담당하며, DispatcherServlet 에게 정제된 모델과 뷰에 대한 정보를 넘겨줌.

❑ Controller

- DispatcherServlet에게서 모든 권한을 위임 받아 실행됨
- 유효성 검증/ 비즈니스 로직/ 예외처리 등의 실질적인 작업을 한다.
- 스프링에서 제공하는 Controller 계열 클래스 중 하나를 상속받아서 구현한다.(2.x)
- @Controller 어노테이션을 사용하여 구현.(3.x)

❑ ModelAndView

- 비즈니스 로직의 결과물(Model) 과 보여줄 View(View객체 또는 논리적인 이름) 를 가지는 객체
- Controller 클래스에서 ModelAndView 생성하여 DispatcherServlet 에게 반환

❑ ViewResolver

- 화면에 어떤 식으로 표시할 지에 대하여 의뢰를 한다.

❑ View

- 결정되어진 표시할 방법으로 요청을 의뢰한 클라이언트의 브라우저로 응답이 전송되어 진다

❑ 순서 : 전체 예제는 Board_Tutorial 을 통해 확인할것.

1. 스프링 프로젝트 생성
2. DD파일(web.xml)에 DispatcherServlet 등록 및 인코딩 필터 등록
3. bean definition 파일 작성 : spring-servlet.xml
4. Controller 및 JSP파일 작성
5. 테스트

❑ 1. Spring MVC 프로젝트 생성:

- 이클립스에 sts 플러그인을 설치하거나, spring.io 에서 sts 를 다운.
- Spring MVC Project 생성
  - Project name: helloSpring
  - Base package : kr.co.sample

❑ 2. DispatcherServlet 등록

- web.xml 에 상위컨텍스트 생성을 위한 ContextLoaderListener를 등록.

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

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

- DispatcherServlet 설정파일은 기본적으로 [servlet-name]-servlet.xml
  - /WEB-INF/**appServlet-servlet.xml** 파일을 찾음
- 설정 파일의 이름을 다르게 지정하고자 경우에는 "contextConfigLocation" 파라미터를 설정할수 있으며, ant path 매핑 방식으로 설정을 여러 개의 xml로 분리할 수 있다.

```
<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>
```

- 인코딩 필터를 등록한다.

```
<filter>
	<filter-name>encodingFilter</filter-name>
	<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
	<init-param>
		<param-name>encoding</param-name>
		<param-value>UTF-8</param-value>
	</init-param>
</filter>

<filter-mapping>
	<filter-name>encodingFilter</filter-name>
	<url-pattern>/*</url-pattern>
</filter-mapping>
```

❑ 3. 빈 정의(bean definition) 파일 작성

- /WEB-INF/spring/appServlet/servlet-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"
	xmlns:mvc="http://www.springframework.org/schema/mvc"
	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">

<!-- MVC 패턴 구현에 필요한 기본 전략들을 자동 등록해주는 설정. -->
		<mvc:annotation-driven />

<!-- 스태틱 리소스들에 대한 관리 영역을 분리하거나 버전별로 관리하고 싶은 경우 유용한 설정, -->
<!-- 스태틱 리소스들에 대한 요청(context_path/resources/..)을 관리영역 하위의 리소스로
매핑시키는 컨트롤러 등록 -->
		<mvc:resources mapping="/resources/**" location="/resources/" />

<!-- DispatcherServlet 의 매핑 설정이 "/" 로 되어있는 경우, -->
<!-- WAS 의 default servlet 매핑이 무효가 되므로, -->
<!-- DispatcherServler 내에서 처리할수 없는 요청의 경우, -->
<!-- WAS 가 가진 default servlet 에게 위임하기 위한 설정. -->
		<mvc:default-servlet-handler />

<!-- view 컴포넌트를 특정영역으로 별도 관리하기 위한 ViewResolver 등록. -->
	<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
		<property name="prefix" value="/WEB-INF/views/" />
		<property name="suffix" value=".jsp" />
	</bean>
	<context:component-scan base-package="kr.co.sample" />
</beans>
```

❑ 4. Controller 및 JSP파일 작성

- org.springframework.web.servlet.mvc.Controller 를 구현(spring 2.X)하거나 구현된 컨트롤러를 선택하여 사용
  - 이 방법을 사용하는 경우 설정 파일 상에 명시적으로 컨트롤러를 등록해야 함.
- @Controller 와 @RequetMapping 어노테이션의 사용
  - Context:component-scan 설정에 의해 base-package 영역내의 컨트롤러 자동 등록.

```
package kr.co.sample;

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;

/**
 * 웹 어플리케이션의 welcome page 를 처리하기 위한 컨트롤러
 */
@Controller
public class HomeController {

	private static final Logger logger = LoggerFactory.getLogger(HomeController.class);

	@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 = DateFormat.getDateTimeInstance(DateFormat.LONG,
		                                                       DateFormat.LONG, locale);

		String formattedDate = dateFormat.format(date);

		model.addAttribute("serverTime", formattedDate );

		return "home";
	}

}
```

❑ 4. Controller 및 JSP파일 작성(계속)

- /WEB-INF/views/home.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" contentType="text/html; charset=UTF-8"%>
<html>
	<head>
		<title>Home</title>
	</head>
	<body>
		<h1>Hello Spring!</h1>

		<P>  현재 시각은 ${serverTime} 입니다.</P>
	</body>
</html>
```

❑ 5. 참고, spring 2.x의 경우 컨트롤러를 명시적으로 등록해야 함.

```
<!-- // handlerMapping을 지정하지 않을 경우 BeanNameUrlHandlerMapping 기본 -->
<bean id="beanNameHandler"
	class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />

<bean name="/" class="kr.co.sample.HomeController" />
```

❑ 6. 테스트

- `helloSpring 프로젝트 실행(http://localhost/sample)

❑ <mvc:annotation-driven /> 으로 등록되는 빈 종류. (3.1.2 ver.)

```
<bean id="handlerMapping"
        class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping“  p:order=“0”
/>
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"/>
<bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean"/>
<bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
<util:list id="messageConverters">
        <bean class="org.springframework.http.converter.ByteArrayHttpMessageConverter" />
        <bean class="org.springframework.http.converter.StringHttpMessageConverter"
                p:writeAcceptCharset="false" />
        <bean class="org.springframework.http.converter.ResourceHttpMessageConverter" />
        <bean class="org.springframework.http.converter.xml.SourceHttpMessageConverter" />
        <bean class="org.springframework.http.converter.xml.Jaxb2RootElementHttpMessageConverter" />
        <bean class="org.springframework.http.converter.support.AllEncompassingFormHttpMessageConverter" />
</util:list>
<bean id="handlerAdapter"
                class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
        <property name="webBindingInitializer">
                <bean class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer"
                        p:validator-ref="validator" p:conversionService-ref="conversionService"/>
        </property>
        <property name="messageConverters" ref="messageConverters" />
        <!-- <property name="customArgumentResolvers"> </property> -->
        <!-- <property name="customReturnValueHandlers"> </property> -->
</bean>
<bean class="org.springframework.web.servlet.handler.MappedInterceptor">
        <constructor-arg index="0">
                <null />
        </constructor-arg>
        <constructor-arg index="1">
                <bean class="org.springframework.web.servlet.handler.ConversionServiceExposingInterceptor">
                        <constructor-arg index="0" ref="conversionService" />
                </bean>
        </constructor-arg>
</bean>
<bean class="org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver">
        <property name="messageConverters" ref="messageConverters" />
        <property name="order" value="0" />
</bean>
<bean class="org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver"
                p:order="1"/>
<bean class="org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver"
                p:order="2"/>
```

❑ Spring @MVC 를 사용한 컨트롤러 구현

: 스프링 프레임워크는 2.5 버전부터 Java 5+ 이상이면 @Controller(Annotation-based Controller)를 개발할 수 있는 환경을 제공한다. 인터페이스 Controller를 구현한 SimpleFormController, MultiActionController 같은 기존의 계층형(Hierarchy) Controller 가 가지는 주요 차이점은 다음과 같다.

- 어노테이션을 이용한 설정 : XML 기반으로 설정하던 정보들을 어노테이션을 사용해서 정의한다.
- 유연해진 메소드 시그니쳐 : Controller 메소드의 파라미터와 리턴 타입을 좀 더 다양하게 필요에 따라 선택할 수 있다.
- POJO-Style의 Controller : Controller 개발시에 특정 인터페이스를 구현 하거나 특정 클래스를 상속해야 할 필요가 없다. 하지만, 폼 처리, 다중 액션등 기존의 계층형 Controller가 제공하던 기능들을 여전히 쉽게 구현할 수 있다.
- 어노테이션 기반의 편의 기능인 <context:component-scan/> 등을 사용하여 핸들러 객체들을 좀더 쉽게 컨테이너에 등록할 수 있다.
- 참고 : <context:component-scan /> 설정
  - @Component, @Service, @Repository, @Controller, @Configuration 으로 정의된 클래스들을 읽어들여 ApplicationContext, WebApplicationContext에 빈정보를 저장, 관리함.
  - <context:include-filter>나 <context:exclude-filter>를 사용하면 특정 조건에 해당하는 빈들의 메타데이터만 등록하는 것이 가능함.

```
<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:p="http://www.springframework.org/schema/p"
  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.xsd">

  <context:component-scan base-package="kr.co.sample">
    <context:include-filter type="annotation"
                            expression="org.springframework.stereotype.Controller"/>
    <context:exclude-filter type="annotation"
                            expression="org.springframework.stereotype.Service"/>
    <context:exclude-filter type="annotation"
                            expression="org.springframework.stereotype.Repository"/>
  </context:component-scan>

</beans>
```

❑ RequestMappingHandlerMapping (3.1 이전까지 DefaultAnnotationHandlerMapping이 사용됨.)

- @MVC 개발을 위한 HandlerMapping. 단 jdk1.5이상의 환경에서 사용가능.
- @RequestMapping에 지정된 url과 해당 Controller의 메소드 매핑
- 기본 HandlerMapping이므로 빈 설정 파일에 별도로 선언할 필요 없으나, 다른 HandlerMapping과 함께 사용한다면 적절한 HandlerAdapter 와 함께 선언해야함.
- mvc 네임스페이스로 지원되는 <mvc:annotation-driven > 태그를 통해 자동 설정되는 빈 들중에 포함되어 있어서, 기본 설정으로만 사용하는 경우, 별도로 등록할 필요가 없음.

```
<bean id="annotationMapper"
  class="org.springframework.web.servlet.mvc.annotation.RequestMappingHandlerMapping"
  p:order="1" />
<bean id="handlerAdapter"
  class="org.springframework.web.servlet.mvc.annotation.RequestMappingHandlerAdapter"
  p:order="1" />
```

❑ 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 해야 하는 부담과 번거로움이 될 수 있다.
- @MVC는 Controller 코드안에 어노테이션 지향형으로 요청 처리를 위한 매핑 설정이 가능하므로 좀 더 쉬운 프로그래밍이 가능함.
- @MVC에서 사용하는 주요 어노테이션은 아래와 같다

이 름	설 명
@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	핸들러 메소드의 파라미터나 특정 메소드의 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.

```
import org.springframework.stereotype.Controller;

@Controller
public class HelloController {
  //...
}
```

❑ **@RequestMapping**

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

이 름	타 입	설 명
value	String[]	요청-핸들러 매핑 조건으로 URL을 사용하기 위한 속성이다. ex) @RequestMapping(value="/hello.do") @RequestMapping(value={"/hello.do", "/world.do"}) @RequestMapping("/hello.do") 혹은 "/myPath/*.do"와 같이 Ant-Style의 패턴을 이용하거나 정규식 매핑을 사용하여 여러형식의 요청을 하나의 핸들러에서 처리하는 것도 가능하다.
method	RequestMethod[]	HTTP Request 메소드값을 매핑 조건으로 부여한다. HTTP 요청 메소드가 일치하는 경우에만 처리할 수 있도록 제한한다. ex) @RequestMapping(method = RequestMethod.POST) 값의 종류 : GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE
params	String[]	HTTP Request 파라미터를 매핑 조건으로 부여한다. ex) params="myParam=myValue" : HTTP Request URL중에 myParam이라는 파라미터가 있어야 하고 값은 myValue이어야 한다. params="!myParam" : myParam이라는 파라미터가 없어야 한다. @RequestMapping(params={"myParam1=myValue", "myParam2", "!myParam3"}) : HTTP Request에는 파라미터가 myParam1이 myValue값을 가지고 있고, myParam2 파라미터가 있어야 하고, myParam3라는 파라미터는 없어야 한다.
produces (spring 3.1)	String[]	Http Request 의 Accept헤더를 매핑 조건으로 부여한다. ex) produces={"application/json", "text/plain"} : 응답데이터의 type 이 json인 요청에 대한 처리를 위한 핸들러가 된다.
consumes (spring 3.1)	String[]	Http Request의 Content-Type 헤더를 매핑 조건으로 부여한다. ex) consumes={"application/json", "text/plain"} : 클라이언트로부터 json형식으로 작성된 요청 데이터가 넘어오는 경우 처리할 핸들러가 된다.

❑ @RequestMapping 메소드의 시그니쳐

- 기존의 계층형 Controller(SimpleFormController, MultiAction..)에 비해 유연한 메소드 파라미터, 리턴값을 갖는다.

❑ @RequestMapping 메소드 파라미터

- **Servlet API - ServletRequest, HttpServletRequest, HttpServletResponse, HttpSession 등의 Servlet API.**
- **org.springframework.web.context.request.WebRequest, ~.NativeWebRequest**
- **java.util.Locale** - LocaleResolver에 의해 결정된 현재 요청의 로케일 정보를 가진 객체
- **java.io.InputStream / java.io.Reader, java.io.OutputStream / java.io.Writer**
- **org.springframework.http.HttpMethod** – request method 를 나타내는 enum type
- **java.security.Principal** – 인증된 클라이이언트의 인증 정보
- **@RequestParam** - HTTP Request의 파라미터를 컨트롤러 메소드로 받기 위해 사용하는 어노테이션.
- **@RequestPart** – multipart/form-data 타입의 클라이언트 전송 데이터를 받기 위해 사용하는 어노테이션으로 Part 의 Content-Type 이 json이나 xml인 경우 해당 Part 의 content 받기위한 파라미터나 파일이 전송되는 경우 받기 위한 MultipartFile 타입의 파라미터에 사용하는 어노테이션
- **@RequestHeader** – 특정한 요청 헤더의 값을 받기위해 사용하는 어노테이션
- **@CookieValue** – 특정 쿠키의 값을 받기위해 사용하는 어노테이션
- **@RequestBody** – request body 영역의 데이터를 받기 위해 사용하는 어노테이션으로 적절한 messageConverter 나 propertyEditor 가 등록된 경우 특정 타입의 파라미터를 통해 request body를 받을수 있음.
- **HttpEntity<?>** - 메시지 컨버터에 의해 특정 타입으로 변환되기 전의 request 에 대한 정보를 가진 객체.
- **@PathVariable** – URI 의 특정 구간 substring 을 추출할때 사용.
- **@MatrixVariable** – URI중 일부 세그먼트에 포함된 name-value 쌍에 접근하기 위한 파라미터에 선언
- **java.util.Map / org.springframework.ui.Model / ~.ModelMap** - 뷰에 전달할 모델데이터.
- **org.springframework.web.servlet.mvc.support.RedirectAttributes** - 리다이렉션 하는 경우 플래시방식의 속성 전달을 위한 맵 객체
- **Command/form 객체** - HTTP Request로 전달된 parameter를 바인딩한 커맨드 객체, @ModelAttribute로 alias를 모델명을 변경할수 있고, @InitBinder 메소드를 통해 요청 파라미터 바인딩 및 검증 설정을 할수있음.
- **org.springframework.validation.Errors / ~.BindingResult** - 유효성 검사 후 결과 데이터를 저장한 객체.
- **org.springframework.web.bind.support.SessionStatus** – 세션범위의 폼 데이터 처리시에 해당 세션을 제거하기 위해 사용.

❑ @RequestMapping 메소드 리턴 타입

- **ModelAndView** - 커맨드 객체와 @ModelAttribute 메소드의 리턴값에 대한 접근방법과 view 에 대한 정보를 가진 객체.
- **Model**(또는 **ModelMap**) - 커맨드 객체, @ModelAttribute 메소드의 리턴값에 대한 접근방법을 가진 객체.
- **View** - 핸들러 메소드의 파라미터로 모델 데이터가 결정된 경우, 뷰에 대한 정보를 리턴값으로 넘김.
- **Map** - 커맨드 객체, @ModelAttribute 메소드의 리턴을 가진 Map 객체.
- **String** - 논리적인 View 이름. 파라미터로 모델 데이터에 대한 접근 방법이 제공되는 경우 사용하는 리턴 타입.
- **void** - 메소드가 ServletResponse / HttpServletResponse등을 사용하여 직접 응답을 처리하는 경우로, View 이름은 RequestToViewNameTranslator가 결정함.
- **HttpEntity<?> / ResponseEntity<?>** - response header 및 response content 를 모두 가진 객체로, Entity body 는 HttpMessageConverter 에 의해 응답데이터로 변환됨.
- **Callable<?>** - 비동기 리퀘스트에 대한 핸들러 메소드에서 사용.
- **DeferredResult<?>** - 비동기 리퀘스트 처리에서 병행처리를 위한 Callable 의 확장형
- **ListenableFuture<?>** - 비동기 리퀘스트 처리에서 병행처리를 위한 Future 의 확장형
- **@ModelAttribute** – 현재 메소드의 리턴값이 뷰로 전달될 모델 데이터임을 의미하는 어노테이션

❑ @RequestMapping 설정

- method level
  - /hello.do 요청이 오면 hello 메소드,
  - /helloForm.do 요청은 GET 방식이면 helloGet 메소드, POST 방식이면 helloPost 메소드가 수행된다.

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class HelloController {
  @RequestMapping("/hello.do")
  public String hello() {
    return "hello";
  }
  @RequestMapping(value="/helloForm.do", method = RequestMethod.GET)
  public String helloGet(){
    return "hello_get";
  }
  @RequestMapping(value="/helloForm.do", method = RequestMethod.POST)
  public String helloPost(){
    return "hello_post";
  }
}
```

❑ @RequestMapping 설정

- type + method level
  - 둘 다 설정할 수도 있는데, 이 경우엔 type level에 설정한 @RequestMapping의 value(URL)를 method level에서 재정의 할수 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(){
    return "hello_get";
  }
  @RequestMapping(method = RequestMethod.POST)
  public String helloPost(){
    return "hello_post";
  }
}
```

❑ @RequestMapping 메소드 파라미터에 사용되는 주요 어노테이션

- @RequestParam
  - @RequestParam은 Controller 메소드의 파라미터와 웹요청 파라미터와 맵핑하기 위한 어노테이션이다.
  - 관련 속성

이 름	타 입	설 명
value	String	파라미터 이름
required	boolean	해당 파라미터가 반드시 필수 인지 여부. 기본값은 true이다.
defaultValue	String	required=false 이고 해당 파라미터가 존재하지 않는 경우 사용할 기본값

  - 해당 파라미터가 Request 객체 안에 없을때 그냥 null값을 바인드 하고 싶다면, pageNo 파라미터 처럼 required=false로 명시해야 한다.
  - name 파라미터는 required가 true이므로, 만일 name 파라미터가 null이면 org.springframework.web.bind.MissingServletRequestParameterException이 발생한다

```
@RequestMapping(value="/hello.do")
 public String hello(@RequestParam("name") String name,
                     @RequestParam(value="pageNo", required=false)String pageNo) {
   return "hello";
 }
```

- @ModelAttribute
  - 관련 속성

이 름	타 입	설 명
value	String	바인드하려는 Model 속성 이름.

  - @ModelAttribute은 Controller에서 2가지 방법으로 사용된다.
    1. Model 속성(attribute)과 메소드 파라미터의 바인딩
       : 요청 파라미터를 바인딩할 커맨드 객체를 선언하고, 이 커맨드 객체가 뷰로 전달될때의 모델속성명을 결정하는데 사용됨.
       : 과거 SimpleController 를 사용할 당시 formBackingObject 를 통해 요청 파라미터를 바인딩하고 모델속성으로 전달하던 방식을 어노테이션 지향형으로 개선한 어노테이션.

```
@RequestMapping(value="/member/memberInsert.do")
public String memberInsert(@ModelAttribute("member") MemberBean member ) {
  //... Registration logic
  return "member/memberView";
}
```

    2. 입력 폼에 필요한 참조 데이터(reference data) 작성. - SimpleFormContrller의 referenceData 메소드와 유사하며 @ControllerAdvice 와 함께 범용 모델 데이터를 접근하는데 사용할 수 있다.

```
@RequestMapping(value="/member/memberList.do")
public String memberList(ModelMap map ) throws Exception {
  List<MemberBean> list = memberService.getMemberList();// DB에서 회원 데이터를 가져온다.
  map.addAttribute("list", list);    // 데이터를 모델 객체에 저장한다.
  return "member/memberList";
} // memberList
```

       : 메소드에서 비지니스 로직(DB 처리같은)을 처리한 후 결과 데이터를 ModelMap 객체에 저장하는 로직은 일반적으로 자주 발생한다.

@ModelAttribute를 메소드에 선언하면 해당 메소드의 리턴 데이터가 ModelMap 객체에 저장되며,
위 코드를 아래와 같이 변경할 수 있다.

```
@RequestMapping(value="/member/memberList.do")
public String memberList() {
  return "member/memberList";
} // memberList

@ModelAttribute("list")
public List<MemberBean> findByAll(){
  return memberService.getMemberList(); //DB에서 회원목록 데이터를  가져온다
}
```

/member/memberList.do 호출이 들어오면, memberList메서드가 실행 되기 전에 (@ModelAttribute가 선언된) findByAll 메서드를 실행하고, 결과를 ModelMap객체에 저장 및 관리한다.

- @PathVariable (다음 페이지 참조)
  - URI 자체만으로 일정데이터를 서버로 전송하는 경우, 일부 경로를 변수로 받기위한 어노테이션.
  - 정규식을 사용하여 정제된 데이터만을 경로변수로 받을 수 있음("/uppath/{varName:regexp}")
    : ex) @RequetMapping("/files/{fileId:[a-zA-Z]+₩₩d₩₩d}")
- @RequestHeader
  - 요청 헤더중 특정 헤더의 값을 파라미터로 받기 위해 사용하는 어노테이션
- @RequestPart
  - Servlet 3에서 지원되는 Part API 에 준하여 만들어진 어노테이션으로 multipart/form-data로 전송되는 컨텐츠를 파라미터로 받기위해 사용되는 어노테이션
  - 주로 content-type 이 JSON이나 XML 혹은 이진파일인 경우에 사용됨
    ex) handlerMtd(@RequestPart Metadata metadata,
    @ReqeustPart Part part, @RequestPart MultipartFile file)
- @RequestBody
  - HttpMessageConverter 를 사용한 요청데이터 바인딩을 위한 설정으로, 핸들러 메소드 시그니쳐의 파라미터 선언에 사용됨.
- @ResponseBody
  - HttpMessageConverter 를 사용한 응답데이터 변환에 사용되는 설정으로 핸들러 메소드의 리턴타입 선언에 사용됨.
  - (@RequestBody, @ResponseBody ;모두 해당 타입 변환을 위한 message converter 가 등록되어있을 때 동작 .)
- @SessionAttributes
  - @SessionAttributes는 model attribute를 session에 저장, 유지할 때 사용하는 어노테이션이다.
  - @SessionAttributes는 클래스 레벨(type level)에서 선언할 수 있다.
  - 관련 속성

이 름	타 입	설 명
value	String[]	session에 저장하려는 model attribute의 이름
types	Class[]	session에 저장하려는 model attribute의 타입

❑ 스프링을 이용한 파일업로드

- multipartResolver 의 종류
  - CommonsMultipartResolver : apache.commons-fileupload와 apache.commons-io 라이브러리 의존성 필요.
  - StandardServletMultipartResolver : Servlet 3.0 이후 사용, DispatcherServlet에 MultipartConfig 설정 필요.
- 핸들러에서 파일데이터를 받기위한 방법
  - @ReqeustPart 어노테이션으로 Part를 직접 받거나, MultipartFile타입 객체로 받기.
  - formbackingobject로 쓰이는 커맨드 객체의 프로퍼티로 MultipartFile 을 선언하여 받기.

❑ 웹컨텐츠 변환을 위한 MessageConverter 의 사용

- 스프링이 기본 지원하는 MessageConverter의 종류 : <mvc:annotation-driven >전략으로 기본 등록되는 컨버터 (org.springframework.http.converter 패키지)
  - ByteArrayHttpMessageConverter
  - StringHttpMessageConverter
  - ResourceHttpMessageConverter
  - xml.SourceHttpMessageConverter
  - xml.XmlAwareFormHttpMessageConverter
  - xml.Jaxb2RootElementHttpMessageConverter
  - json.MappingJackson2HttpMessageConverter
    : Spring 3.x까지 Jackson 1.x 라이브러리에 의존했으나, Sprig 4부터 2.x 의존성 필요
    (Spring 4부터 MappingJacksonHttpMessageConverter 는 deprecated 됐음.)
  - feed.AtomFeedHttpMessageConverter
  - feed.RssChannelHttpMessageConverter
    : RSS 피드 데이터 변환을 위한 feed패키지의 클래스들은 Spring 3.x까지는 ROME 라이브러리 1.0 버전에 대해 의존성을 갖지만, Spring 4 부터 ROME 1.5 에 의존함.
- Custom HttpMessageConverter : AbstractHttpMessageConverter의 하위구현체로 작성

```
public class CustomHttpMessageConverter extends AbstractHttpMessageConverter<SampleBean>{

}
```

- 웹컨텐츠를 MessageConverter로 변환하는 방법
  1. 적절한 형태의 message converter 를 handlerapdaper에 등록

```
<mvc:annotation-driven>
	<mvc:message-converters>
		<bean class="kr.co.sample.mvc.converters.CustomHttpMessageConverter" />
	</mvc:message-converters>
</mvc:annotation-driven>
```

2-1. 핸들러 메소드의 파라미터나 리턴값에 @RequestBody나 @ResponseBody 를 사용

```
@ResponseBody
public SampleBean testHandler(@RequestBody SampleBean sBean){
}
```

2-2. 핸들러 메소드의 파라미터나 리턴값에 HttpEntity객체를 직접 받아 Entity Body를 핸들링 하는 방법.

```
public HttpEntity<SampleBean> testHandler(HttpEntity<SampleBean> reqEnt){
	SampleBean body = reqEnt.getBody();
	HttpEntity<SampleBean> respEnt = new HttpEntity<SampleBean>(body);
	return respEnt;
}
```

❑ 컨트롤러 전/후의 처리를 위한 HandlerInterceptor의 사용

- 인증이나 인가 처리 혹은 요청 흐름 제어 및 응답데이터 필터링 등의 작업에 Decorator pattern 에 따라 javax.servlet.Filter 를 적용하면 웹 어플리케이션 내부의 로직에 영향을 주지않고 해당 작업들을 구현할수 있음. 다만, 스프링을 사용하는 경우 필터들이 웹컨테이너에 등록되어 관리되므로 스프링 컨테이너 내부의 빈들과 의존관계 형성에 어려움이 발생할 수 있다. 이를 감안하여 요청이나 응답에 대한 전/후처리, 즉 핸들러 메소드 호출 전이나 핸들러 메소드 호출 이후에 부가적인 처리가 필요한 경우, HandlerInterceptor를 사용함.
- Custom HandlerInterceptor : HandlerInterceptor 나 HandlerInterceptorAdapter의 하위 구현체로 작성

```
public class CustomHandlerInterceptor implements HandlerInterceptor{
	@Override
	public boolean preHandle(HttpServletRequest request,
		HttpServletResponse response, Object handler) throws Exception {
		return true;
	}
	@Override
	public void postHandle(HttpServletRequest request,
		HttpServletResponse response, Object handler,
		ModelAndView modelAndView) throws Exception {
	}
	@Override
	public void afterCompletion(HttpServletRequest request,
		HttpServletResponse response, Object handler, Exception ex)
		throws Exception {
	}
}
```

  - preHandler : 핸들러 메소드 전 호출되는 메소드로 리턴값이 false라면 핸들러 메소드는 호출되지 않고, DispatcherServlet 은 응답데이터가 이미 커밋된걸로 판단함.
  - postHandler : 핸들러 메소드 실행 이후 실제 View가 랜더링되기 전에 호출.
  - afterCompletion : 핸들러 메소드 실행 이후 실제 View가 랜더링되고 요청 처리가 완료된 다음 호출.
- HandlerInterceptor 등록 및 매핑

```
<mvc:interceptors>
	<beans:bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
		p:paramName="lang" />
	<mvc:interceptor>
		<mvc:mapping path="/filtering/**"/>
		<bean class="kr.co.sample.interceptors.CustomHandlerInterceptor" />
	</mvc:interceptor>
</mvc:interceptors>
```

❑ 컨테이너 내부의 필터bean을 사용하기 위한 DelegateFilterProxy 의 사용

- Spring-Security 에서 컨테이너 내부의 필터체인으로 필터링 기능을 위임하기 위해 사용했던 프록시 필터 방식으로 web.xml에 다음과 같이 선언하고, 컨테이너 내부의 bean name 을 filter-name 으로 지정하여 위임함. 필터빈은 루트 컨텍스트에 등록됨을 기본으로 하고, 특정 컨텍스트에 등록된경우, contextAttribute 파라미터로 지정할 수 있음.

```
<filter>
	<filter-name>targetBeanName</filter-name>
	<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
	<filter-name>targetBeanName</filter-name>
	<url-pattern>/*</url-pattern>
</filter-mapping>
```

❑ REST 아키텍처와 @PathVariable 어노테이션의 활용

- REST (Representational State Transfer) 구조 : 자원을 식별하고 접근하는 방법에 대한 정의이며, 자원에 대한 식별(Noun)과, 자원에 대한 행위 메소드(verb), 자원의 상태 (Representation) 로 구성된다.
  로이필딩이 자신이 설계한 Http 프로토콜의 장점을 활용하기 위해 제안한 네트워크 기반의 아키텍처로 기본은 Http URI + Http Method (+ Payload) 이며, "무엇(resource) 을 어떻게(Verb) " 로 정의된 구조이다.
  - 구성 요소 : Resource,Noun(URI) / Verb(Http Method) / Representation(Payload xml/json/yaml)
  - 특징 :
    1. Client / Server 구조로 분리된다.
    2. 각 요청에 대해 클라이언트의 컨텍스트가 서버에 저장되지 않는다(Stateless).
    3. 클라이언트가 응답을 캐싱할 수 있다(Cacheable).
    4. 클라이언트는 서버에 직접 연결되었는지 중간서버를 통해 연결되었는지 알수 없도록 하여, 로드밸런싱이나 공유 캐싱 등을 통해 시스템의 규모를 용이하게 한다(Layered System).
    5. 메시지 표현방식과 그 Parser 에 대한 정보까지 메시지에 포함하거나, 향후 상태 변이에 대한 부가 정보(LINK) 등을 포함하도록 하여, 자원 자체를 사용하기 전에 그 자원에 대한 사용방법을 확인할 수 있는 자기 서술적 (Self-descriptive) 메시지 특성을 가진다.
    6. 일관된 인터페이스로 자원에 대한 조작을 통일화하고, 구조를 단순화 시킴으로 C/S 의 각 파트가 독립적으로 개선될 수 있도록 한다.
  - 디자인 가이드
    1. URI 는 자원에 대한 식별자로만 사용한다(Noun).
       - 모든 자원은 서버에서 관리되고, 각 자원은 고유한 식별자(id)를 갖는다.
       - 자원은 /member/a001 과 같은 형태로 URI 를 이용하여 식별한다.
    2. 자원에 대한 조작은 Http Method 로 표현한다(Verb).
       - http method 를 통해 URI 로 식별된 자원에 대한 행위 즉 명령(CRUD)에 대한 정보를 전달한다.
         ex) /member/a001 (GET)
    3. 자원 요청에 대한 응답은 자원에 대한 표현의 형태로 범용 표현방식을 사용하여 전달된다.
       - JSON, XML, RSS, TEXT 등 대부분의 표현 방식이 활용될 수 있으나, 주로 json/xml 이 활용됨.
         ex) /member/a001/edit (PUT/PATCH) 라면 클라이언트에서 서버쪽으로 수정할 자원에 대한 데이터를 json/xml 형태로 표현하여 전달한다.
  - URI 설계시 참고사항
    1. "/"는 계층 관계에만 사용한다.
    2. URI 마지막에 "/"를 사용하지 않는다.
    3. "_" 대신 가독성을 높이기 위해 "-"을 사용한다.
    4. URI 경로에는 소문자를 사용한다(도메인을 제외한 경로는 대소문자를 구별함).
    5. 파일의 확장자는 포함하지 않는다.
  - RESTful URI 예

1.	GET	/members	회원목록 조회
2.	GET	/members/new	신규회원 등록 폼
3.	POST	/members/new	신규회원 등록
4.	GET	/members/{id}	회원정보 상세 조회
5.	GET	/members/{id}/edit	회원정보 수정 폼
6.	PUT	/members/{id}	회원정보 수정
7.	DELETE	/members/{id}	회원정보 삭제

- URI 를 이용한 요청 파라미터 전달 방식
  1. Query String : ex) /cars?type=compact&color=red (빨간색 경차 조회)
     쿼리 스트링에 포함된 멀티 섹션으로 전달하는 전통적인 데이터 전송 구조
     @RequestParam(name, required, defaultValue)
     ex) @RequestMapping("/cars")
         @RequestParam String type, String color
  2. Path Parameters : ex) /cars/compact/red (빨간색 경차 조회)
     URI 에 포함시켜 경로의 일부분으로 표현하는 데이터 전송 구조,
     @PathVariable(name, required, defaultValue)
     ex) @RequestMapping("/cars/{type}/{color}")
         @PathVariable String type, @PathVariable String color
  3. Matrix Parameters : /cars/daejon;gu=junggu/compact;color=red;
     (대전 중구에 등록된 빨간색 경차 조회)
     @MatrixVariable(name, pathVar, required, defaultValue)
     ex) @RequestMapping("/cars/{city}/{type}")
         @PathVariable String city, @MatrixVariable String gu,
         @MatrixVariable(pathVar="type") String color
- Rest 요청 처리 컨트롤러 예
  @RequestBody/@ResponseBody 어노테이션으로, 컨테이너 내부에 등록된 message converter 를 활용하는 구조.

```
@Controller
@RequestMapping(value="/cars", produces=MediaType. APPLICATION_JSON_UTF8_VALUE)
public class  CarController {
        @GetMapping
        @ResponseBody
        public List<Car> readCars(@MatrixVariable String type, @MatrixVariable String color) {
                return cars;
        }
        @GetMapping("{carId}")
        @ResponseBody
        public Car readCar(@PathVariable String carId) {
                return car;
        }
        @PutMapping("{carId}/edit")
        @ResponseBody
        public Car modifyCar(@RequestBody Car car) {
                return modifiedCar;
        }
        @ DeleteMapping("{carId}")
        @ResponseBody
        public Status removeCar(@PathVariable String carId) {
                return status;
        }
        @PostMapping
        @ResponseBody
        public Car registCar(@RequestBody Car car) {
                return registeredCar;
        }
}
```

❑ Spring 3.2 이후 지원되는 추가 어노테이션

- @RequestMapping 을 메타 어노테이션으로 갖는 복합 어노테이션 지원 (since 4.3)
  - @GetMapping : @RequestMapping(method=RequestMethod.GET) 복합 어노테이션
  - @PostMapping : @RequestMapping(method=RequestMethod.POST) 복합 어노테이션
  - @PutMapping : @RequestMapping(method=RequestMethod.PUT) 복합 어노테이션
  - @PatchMapping : @RequestMapping(method=RequestMethod.PATCH) 복합 어노테이션
  - @DeleteMapping : @RequestMapping(method=RequestMethod.DELETE) 복합 어노테이션

  @RequestMapping 이 갖는 속성들을 그대로 갖고, http method 에 따른 웹 요청 핸들러를 매핑할 때 사용되며, 주로 Rest 처리 방식에 활용됨.
- @ControllerAdvice (since 3.2)

  AOP 방법론을 사용하여, @Controller layer 를 대상으로 공통 기능을 적용할 때 활용할 수 있음.
  주로 @ExceptionHandler 를 이용한 공통 예외 처리나, @InitBinder 를 이용한 커맨드 오브젝트 공통 바인드 설정, @ModelAttribute 을 이용한 공통 모델 데이터 설정등에 활용되며, before weaving 과 after throw weaving 의 형태를 사용한다. 기본적으로 모든 @Controller 빈들에 위빙되지만, 타겟 컨트롤러를 제한할 수 있는 다음과 같은 선택자 역할의 속성들이 지원된다.

이 름	타 입	설 명
value	String[]	basePackages 속성에 대한 alias
basePackages	String[]	지정된 패키지들의 계층 구조 아래의 모든 컨트롤러를 타겟으로 하여 advice 가 위빙
basePackageClasses	Class<?>[]	지정된 클래스들이 포함된 패키지의 계층 구조 아래 모든 컨트롤러를 타겟으로 위빙되며, 포인트컷 용도로만 사용될 no-op 마커 클래스나 인터페이스를 활용할 것을 권장.
assingableTypes	Class<?>[]	특정 컨트롤러 클래스들만을 대상으로 위빙할때 활용
annotations	Class<? extends Annotation>[]	특정 어노테이션들을 가지고 있는 컨트롤러 클래스를 대상으로 위빙

- @RestController(since 4.0)

  스프링 4버전 부터 @ResponseBody 를 타입(클래스) 레벨에서 사용할 수 있도록 하여 추가된 어노테이션.
  @Controller 와 @ResponseBody 를 메타 어노테이션으로 갖는 복합 어노테이션으로 해당 컨트롤러의 모든 핸들러 메소드가 Rest 요청을 처리하기 위해 자원의 상태정보를 마샬링하고 있는 경우에 사용됨.

```
@RestController
@RequestMapping(value="/cars", produces=MediaType. APPLICATION_JSON_UTF8_VALUE)
public class  CarController {
        @GetMapping
        public List<Car> readCars(@MatrixVariable String type, @MatrixVariable String color) {
                return cars;
        }
}
```

- @RestControllerAdvice(since 4.3)

  @ControllerAdvice 와 @ResponseBody 를 메타 어노테이션으로 갖는 복합 어노테이션.

❑ Spring Java Configuration 을 이용한 웹어플리케이션 설정.

```
@Configuration
@EnableWebMvc //<mvc:annotation-driven />
//<context:component-scan />
@ComponentScan(basePackages="kr.co.ddit.**.controller", useDefaultFilters=false
 , includeFilters= @ComponentScan.Filter(type=FilterType.ANNOTATION, value=Controller.class))
public class WebAppConfig extends WebMvcConfigurerAdapter{
	@Override // <mvc:default-servlet-handler />
	public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
		configurer.enable();
	}
	@Override // <mvc:resources mapping="" handler="" >
	public void addResourceHandlers(ResourceHandlerRegistry registry) {
		registry.addResourceHandler("/resources/**").addResourceLocations("/resources/");
	}
	@Bean
	public LocaleResolver LocaleResolver(){
		CookieLocaleResolver resolver = new CookieLocaleResolver();
		resolver.setCookieName("localeCookie");
		resolver.setCookiePath("/");
		resolver.setCookieMaxAge(60*60*24*2);
		return resolver;
	}
	@Override // <mvc:interceptors >
	public void addInterceptors(InterceptorRegistry registry) {
		LocaleChangeInterceptor interceptor = new LocaleChangeInterceptor();
		interceptor.setParamName("lang");
		registry.addInterceptor(interceptor);
	}
	@Override // <mvc:message-converters>
	public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
		MappingJacksonHttpMessageConverter jsonConverter =
				new MappingJacksonHttpMessageConverter();
		List<MediaType> mediaTypes = new ArrayList<>();
		mediaTypes.add(MediaType.valueOf("application/json;charset=UTF-8"));
		jsonConverter.setSupportedMediaTypes(mediaTypes);
		converters.add(jsonConverter);
	}
	@Bean
	public InternalResourceViewResolver irvr(){
		InternalResourceViewResolver resolver = new InternalResourceViewResolver();
		resolver.setPrefix("/WEB-INF/views/");
		resolver.setSuffix(".jsp");
		return resolver;
	}
	@Bean
	public MessageSource messageSource(){
		ResourceBundleMessageSource msgSrc = new ResourceBundleMessageSource();
		msgSrc.setBasenames("kr.co.sample.msgs.message", "kr.co.sample.msgs.errorMessage");
		return msgSrc;
	}
}
```

❑ WebApplicationContext를 생성 및 관리.

: 기존의 web.xml 대신 WebApplicationInitializer 나 AbstractDispatcherServletInitializer 를 구현하여 상하위 WebApplicationContext 생성 및 관리를 설정함.

```
public class SpringMVCWebAppInitialization extends AbstractDispatcherServletInitializer{
	@Override
	protected Filter[] getServletFilters() {
		CharacterEncodingFilter encodingFilter = new CharacterEncodingFilter();
		encodingFilter.setEncoding("UTF-8");
		encodingFilter.setForceEncoding(true);
		return new Filter[]{encodingFilter};
	}

	@Override
	protected WebApplicationContext createServletApplicationContext() {
		AnnotationConfigWebApplicationContext context =
				new AnnotationConfigWebApplicationContext();
		context.register(WebAppConfig.class);
		return context;
	}

	@Override
	protected String[] getServletMappings() {
		return new String[]{"/", "/index.do"};
	}

	@Override
	protected WebApplicationContext createRootApplicationContext() {
		AnnotationConfigWebApplicationContext rootContext =
				new AnnotationConfigWebApplicationContext();
		rootContext.getEnvironment().setActiveProfiles("dev", "default");
		rootContext.register(RootWebAppConfig.class,
				DataSourceConfig.class, AopConfig.class);
		return rootContext;
	}
}
```

- createRootApplicationContext() : 상위 컨테이너의 config 클래스를 로딩하고 root WebApplicationContext 를 생성하기 위한 메소드로 웹어플리케이션 전체에 대해 특정 profile을 활성화하거나 커스텀 PropertySource 등을 설정하는데 사용됨. 위의 작업을 수행하기 위해 일반적으로 standalone 에서는 컨테이너 객체를 생성한 이후에 설정 작업을 하고, refresh 메소드를 호출하여 싱글턴 빈들을 인스턴스화하게 되지만, 웹어플리케이션의 경우 상위 WebApplicationInitializer 내에서 등록되는 ContextLoadListener 가 refresh를 담당함.
- createServletApplicationContext : 하위 컨테이너 객체의 config 클래스를 로딩하고 DispatcherServlet 을 등록한 후 getServletMappings 메소드의 리턴값에 따라 매핑 설정을 하는 메소드.
- getServletMappings : 등록될 DispatcherServlet에 요청 정보를 매핑하기 위한 url-pattern 지정.
- getServletFilters : 요청이나 응답들 필터링하기 위한 필터가 필요한 경우, Filter들을 등록해주면, 배열의 순서에 따라 FilterChain을 형성하게 됨.
- 참고 : 하위에 multi-context가 필요하고 각기 다른 매핑을 갖는 DispatcherServlet이 필요하다면, WebApplicationInitializer를 직접 구현하는 방식을 사용할 수 있음.