

스프링 MVC

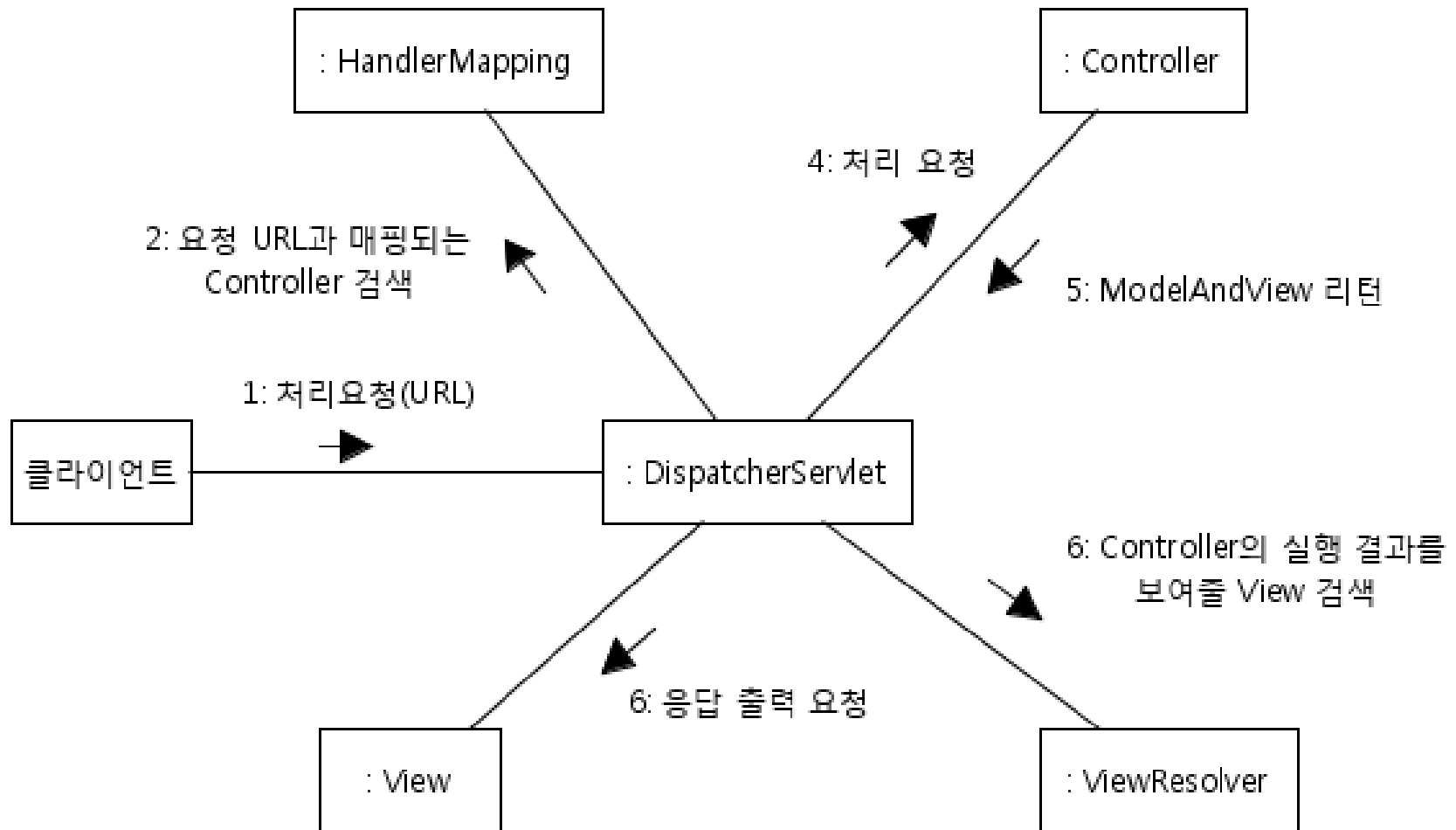
# 목차

- 스프링 MVC의 주요 구성요소 및 처리흐름
- DispatcherServlet과 ApplicationContext
- 컨트롤러 구현
- 폼 값 검증 및 에러 메시지
- 파일 업로드
- HandlerInterceptor
- 예외처리

# 스프링 MVC의 주요 구성 요소

구성요소	설명
DispatcherServlet	클라이언트의 요청을 전달받아 컨트롤러에게 클라이언트의 요청을 전달하고 컨트롤러가 리턴한 결과 값을 View에 전달하여 알맞은 응답을 생성하도록 한다.
HandlerMapping	클라이언트의 요청 URL을 어떤 컨트롤러가 처리할지를 결정한다.
컨트롤러(Controller)	클라이언트의 요청을 처리한 뒤, 그 그결과를 DispatcherServlet에게 알려준다. 스트럿츠의 Action과 동일한 역할을 수행한다.
ModelAndView	컨트롤러가 처리한 결과 정보 및 뷰 선택에 필요한 정보를 담는다.
ViewResolver	컨트롤러의 처리 결과를 생성할 뷰를 결정한다.
뷰(View)	컨트롤러의 처리 결과 화면을 생성한다.

# 스프링 MVC의 클라이언트 요청 처리 과정



# 스프링 MVC Hello World

- 클라이언트의 요청을 받을 DisptcherServlet을 web.xml에 설정한다.
- 클라이언트의 요청을 처리할 컨트롤러를 작성한다.
- ViewResolver를 설정한다. ViewResolver는 컨트롤러가 전달한 값을 이용해서 응답 화면을 생성할 뷰를 결정한다.
- JSP를 이용하여 뷰 영역의 코드를 작성한다.
- 실행

# DispatcherServlet 설정 및 컨텍스트 설정

- 클라이언트의 요청을 전달받을 DispatcherServlet 설정
- 공통으로 사용할 어플리케이션 컨텍스트 설정

```
<servlet>  
  <servlet-name>dispatcher</servlet-name>  
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
</servlet>
```

```
<servlet-mapping>  
  <servlet-name>dispatcher</servlet-name>  
  <url-pattern>*.do</url-pattern>  
</servlet-mapping>
```

\*.do로 들어오는 클라이언트의 요청을 DispatcherServlet이 처리하도록 설정

**dispatcher-servlet.xml** 파일을 설정 파일로 사용하게 됨.

MVC의 구성요소인 Controller, ViewResolver, view 등의 빈을 설정한다.

# 컨트롤러 구현 및 설정 추가

@Controller

→ 해당 클래스가 스프링 MVC의 컨트롤러라는 것을 지정

```
public class HelloController {
```

```
    @RequestMapping("/hello.do")
```

```
    public ModelAndView hello()
```

```
    {
```

```
        ModelAndView mav = new ModelAndView();
```

```
        mav.setViewName("hello");
```

```
        mav.addObject("greeting", getGreeting());
```

```
        return mav;
```

```
    }
```

→ 요청 경로를 처리할 메서드를 설정, /hello.do로 요청했을 때 hello()가 처리하게 됨.

→ 컨트롤러의 처리 결과를 보여줄 뷰와 뷰에서 출력할 모델을 지정

```
private String getGreeting() {
```

```
    int hour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
```

```
    if(hour >= 6 && hour <= 10){
```

```
        return "좋은 아침입니다.";
```

```
    } else if(hour >= 12 && hour <= 15){
```

```
        return "점심 식사는 하셨나요?";
```

```
    } else if(hour >= 18 && hour <= 22){
```

```
        return "좋은 밤 되세요";
```

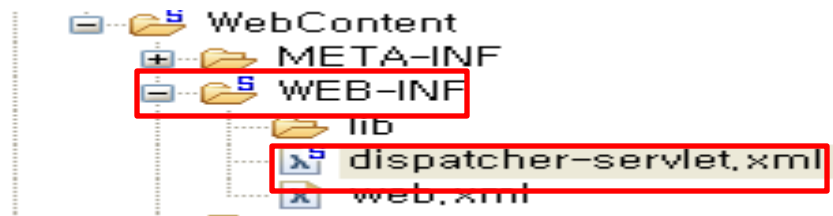
```
    }
```

```
    return "안녕하세요";
```

```
}
```

```
}
```

# 설정파일의 작성

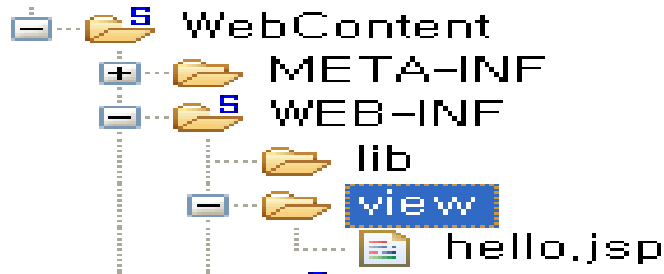


```
<bean id="helloController"  
      class="exam.test.HelloController"/>
```

```
<bean id="viewResolver"  
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
    <property name="prefix" value="/WEB-INF/view/" />  
    <property name="suffix" value=".jsp" />  
</bean>
```



# 뷰 코드 구현

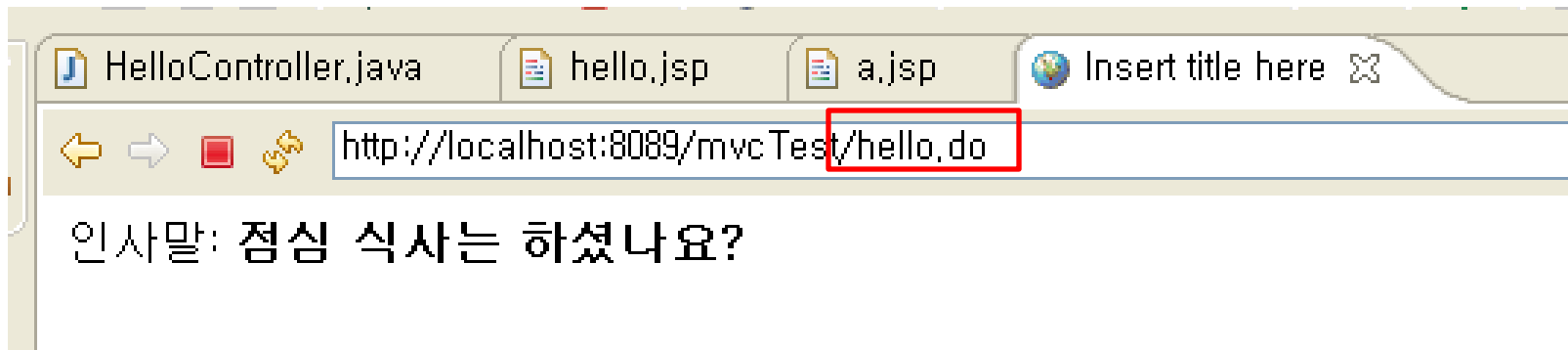


```
- 컨트롤러
@RequestMapping("/hello.do")
public ModelAndView hello() {
    ModelAndView mav = new ModelAndView();
    mav.setViewName("hello");
    mav.addObject("greeting", getGreeting());
    return mav;
}
```

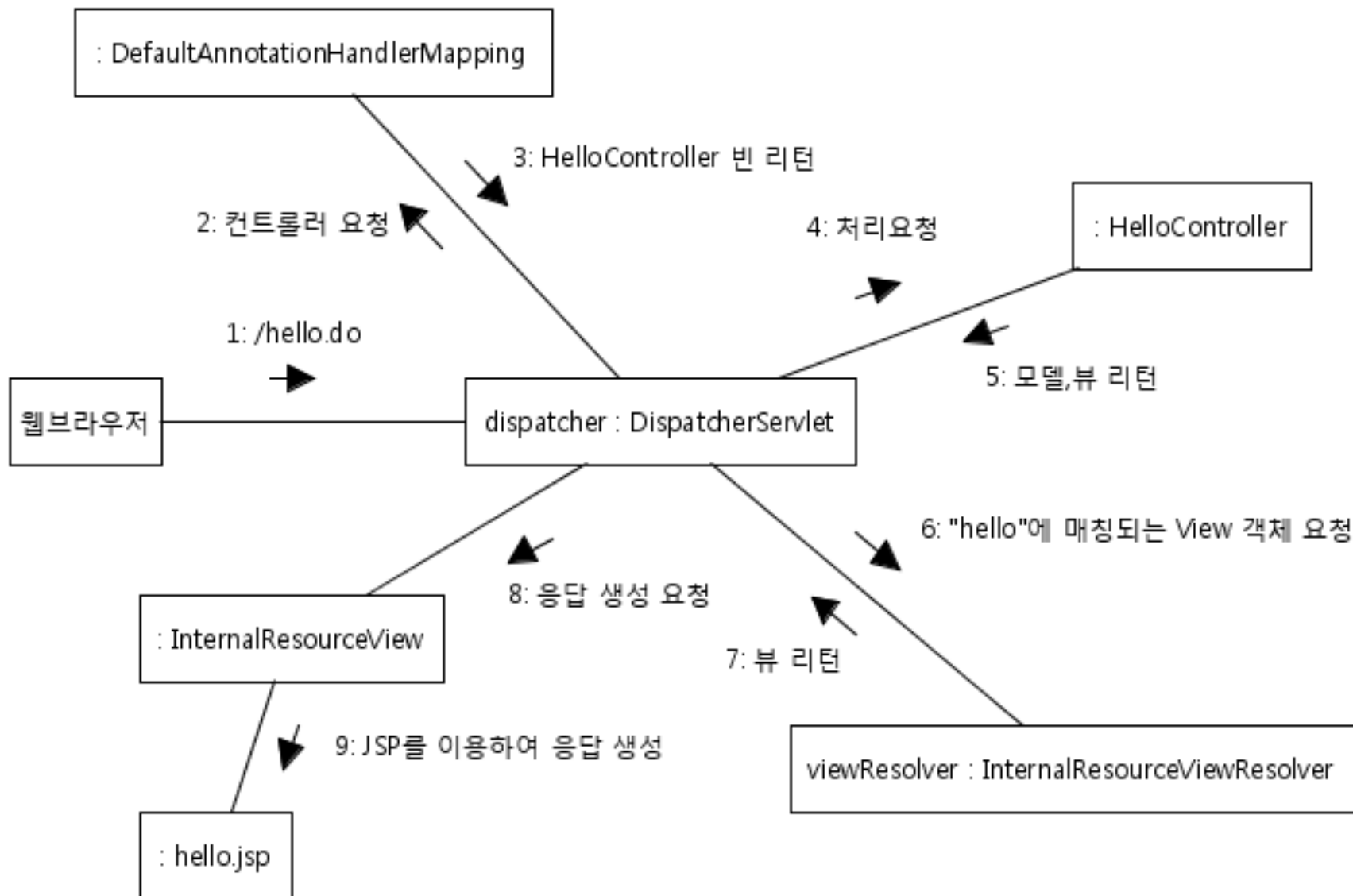
```
- 뷰JSP
<body>
인사말: <strong>${greeting}</strong>
</body>
```

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html" />
<title>Insert title here</title>
</head>
<body>
    인사말: <strong>${greeting}</strong>
</body>
</html>
```

# 실행



# 실행 흐름 정리



## DispatcherServlet 설정과 ApplicationContext의 관계

- DispatcherServlet
  - 클라이언트의 요청을 중앙에서 처리하는 스프링 MVC의 핵심 구성요소
  - web.xml에 한 개 이상의 DispatcherServlet을 설정
  - 각 DispatcherServlet은 한 개의 WebApplicationContext를 갖게 된다.

# DispatcherServlet의 설정

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
```

등록한 서블릿명-servlet.xml 파일로 부터 스프링 설정 정보를 읽어 온다.

# 다른 이름의 설정 파일 사용

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

설정파일의 구분은  
coma, 공백문자, 탭, 줄바꿈, 세미콜론으로 한다.

# 웹 어플리케이션을 위한 ApplicationContext 설정

- DispatcherServlet은 한 개 이상의 DispatcherServlet을 설정 하는 것이 가능하다.

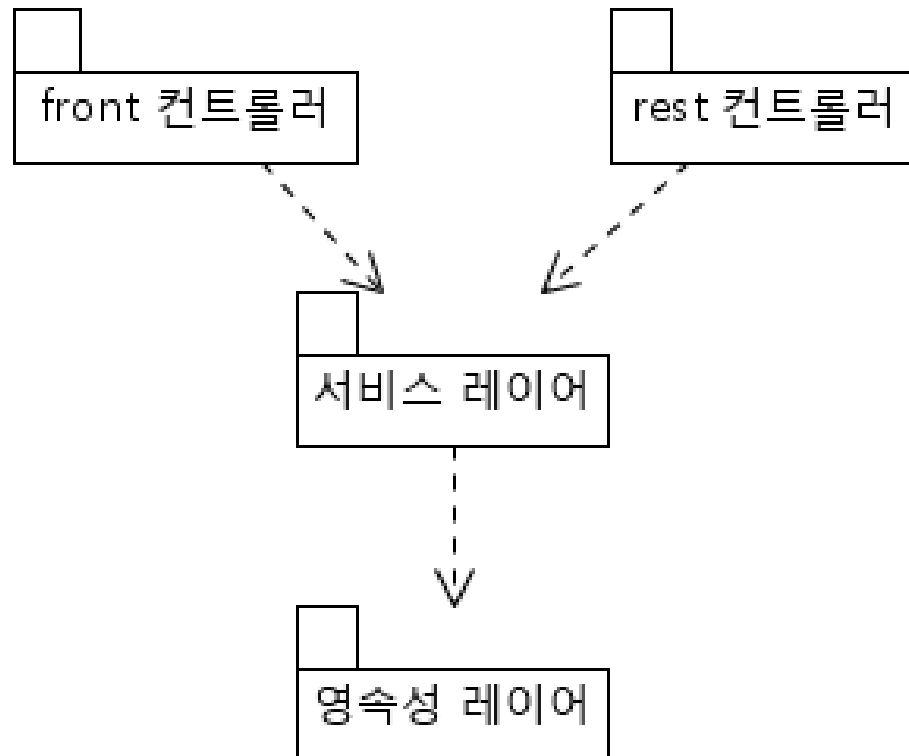
```
<servlet>
  <servlet-name>front</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>

<servlet>
  <servlet-name>rest</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
```

각각 별도의 WebApplicationContext를 생성하게 된다.  
front.xml에서는 rest.xml에 설정된 빈 객체를 사용할 수 없다.

## 서로 다른 DispatcherServlet이 공통 빈을 필요로 하는 경우

- ContextLoaderListener를 사용하여 공통으로 사용될 빈을 설정할 수 있다.





# ContextLoaderListener의 설정

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/service.xml, /WEB-INF/persistence.xml</param-value>
</context-param>
```

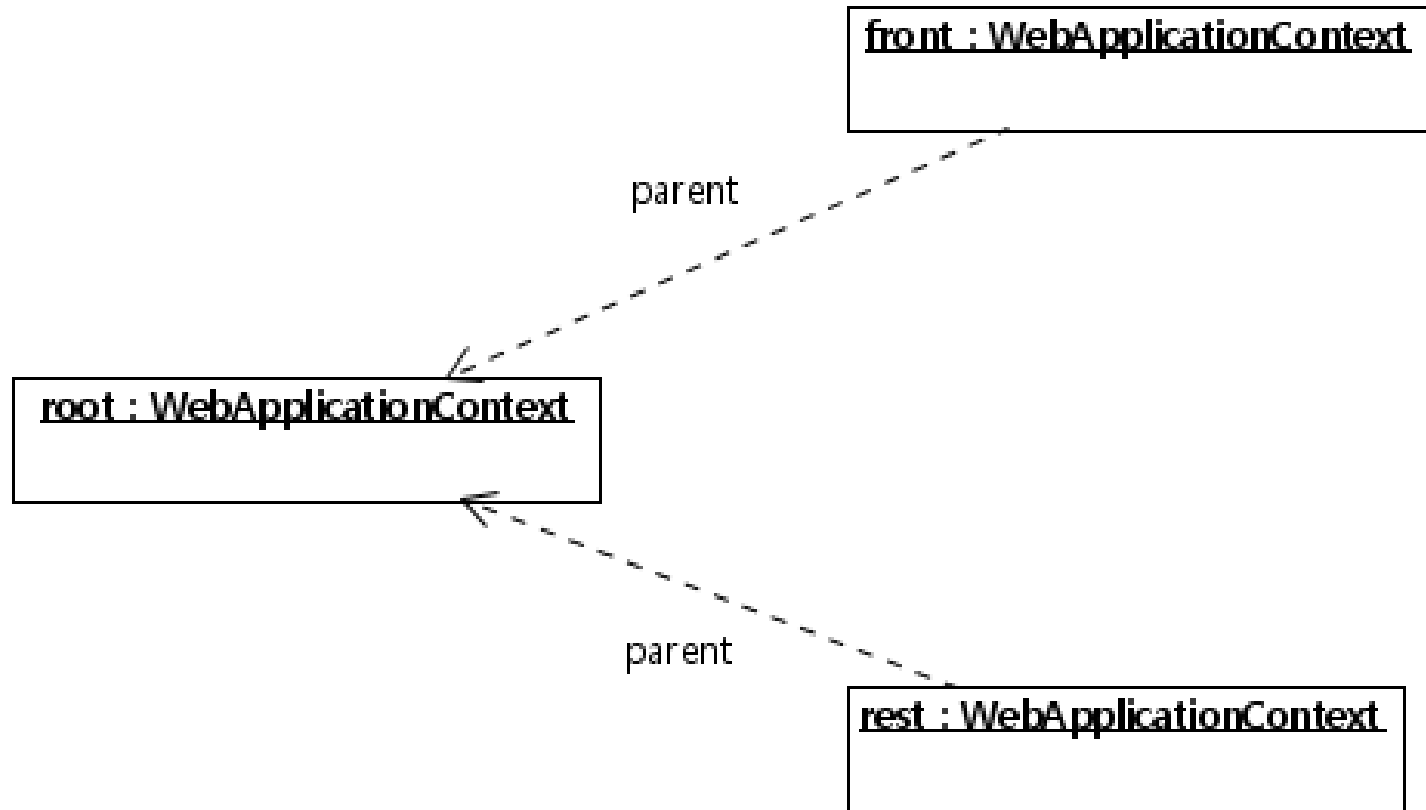
```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

```
<servlet>
  <servlet-name>front</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
```

```
<servlet>
  <servlet-name>rest</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
```

front.xml과 rest.xml에서는 service.xml과 persistence.xml에서 설정된 빈 객체를 사용할 수 있다.

# 생성되는 WebApplicationContext 간의 관계



ContextLoaderListener가 생성하는 `WebApplicationContext`는 웹 어플리케이션에서 루트 컨텍스트가 되며 자식 컨테이너들은 `root`가 제공하는 빈을 사용할 수 있다.

# root WebApplicationContext의 설정 파일

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/service.xml, /WEB-INF/persistence.xml</param-value>
</context-param>
```

\*\* 컨텍스트 파라미터를 명시하지 않으면  
/WEB-INF/applicationContext.xml을 설정 파일로 사용한다.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:config/service.xml
    classpath:common.xml
    /WEB-INF/config/message_conf.xml</param-value>
</context-param>
```

# 캐릭터 인코딩 처리를 위한 필터 설정

요청 파라미터의 캐릭터 인코딩이 ISO-8559-1이 아닌 경우 request.setCharacterEncoding()으로 알맞게 설정해야 한다.

## 1. 코드에서 설정

- Response.setCharacterEncoding("UTF-8");

## 2. web.xml에서 설정

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

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

# 컨트롤러 구현

- 스프링 3.0 버전부터 @Controller 어노테이션을 이용하여 컨트롤러 클래스 구현 하도록 권장
- @Controller 어노테이션과 @RequestMapping 어노테이션을 이용함으로써 고전 방식의 스프링 MVC의 복잡함을 줄일 수 있다.

## @Controller 어노테이션과 @RequestMapping 어노테이션

- 컨트롤러 클래스를 구현하려면 @Controller 어노테이션과 @RequestMapping 어노테이션을 이용한다.
- 1) 컨트롤러 클래스에 @Controller 어노테이션을 적용한다
  - 2) 클라이언트의 요청을 처리할 메서드에 @RequestMapping 어노테이션을 적용한다.
  - 3) 설정 파일에 컨트롤러 클래스를 빈으로 등록한다.

# 컨트롤러 구현 클래스의 예

```
@Controller
```

```
public class HelloController {  
    @RequestMapping("/hello.do")  
    public String hello()  
    {  
        return "hello";  
    }  
}
```

# 컨트롤러 객체의 생성

```
<bean id="helloController"  
      class="exam.test.HelloController"/>
```

스프링 설정파일에 컨트롤러를 객체를 생성하게 되면  
@RequestMapping 어노테이션 값으로 설정한 URI와 매칭되는 클라이언트의  
요청을 해당 메서드에서 처리하게 된다.



# 컨트롤러 메서드의 HTTP 전송방식

```
@Controller
public class NewArticleController {

    @RequestMapping(value="/article/newArticle.do", method = RequestMethod.GET)
    public String form() {
        return "article/newArticleForm";
    }

    @RequestMapping(value="/article/newArticle.do", method = RequestMethod.POST)
    public String submit() {
        return "article/newArticleSubmitted";
    }
}
```

## 해당 클래스가 처리할 기본 URI를 지정

```
@Controller
```

```
@RequestMapping("/article/newArticle.do")
```

```
public class NewArticleController {
```

```
    @RequestMapping(method = RequestMethod.GET)
```

```
    public String form() {
```

```
        return "article/newArticleForm";
```

```
    }
```

```
    @RequestMapping(method = RequestMethod.POST)
```

```
    public String submit() {
```

```
        return "article/newArticleSubmitted";
```

```
    }
```

# HTML 폼과 커맨드 객체

```
<form method="post">
  <input type="hidden" name="parentId" value="0" />
  제목: <input type="text" name="title" /><br/>
  내용: <textarea name="content"></textarea><br/>
  <input type="submit" />
</form>
```



입력 항목의 이름과 일치하는  
프로퍼티에 값이 저장

```
public class NewArticleCommand {
    private String title;
    private String content;
    private int parentId;

    public void setTitle(String title) {
        this.title = title;
    }
    public void setContent(String content) {
        this.content = content;
    }
    public void setParentId(int parentId) {
        this.parentId = parentId;
    }
    ... // get 메서드
}
```

# Html 폼에 입력한 데이터를 자바 빈 객체로 전달

- @RequestMapping 어노테이션이 적용된 메서드의 파라미터로 자바빈 타입을 추가한다.

```
@Controller
@RequestMapping("/article/newArticle.do")
public class NewArticleController {

    @RequestMapping(method = RequestMethod.POST)
    public String submit(NewArticleCommand command) {
        //
        return "article/newArticleSubmitted";
    }
}
```

# 타입 변환 처리

- 폼의 입력값은 모두 문자열
- 스프링은 자바빈의 타입 변환 처리를 통해 알맞게 변환해준다.
- 기본데이터 타입이 아닌 경우
  - @InitBinder 어노테이션과 커스텀 데이터 타입 매핑 처리로 변환함.(뒤에서 다룸)

# 뷰에서 커멘트 객체 접근

```
@Controller
@RequestMapping("/article/newArticle.do")
public class NewArticleController {

    @RequestMapping(method = RequestMethod.POST)
    public String submit(NewArticleCommand command) {
        //
        return "article/newArticleSubmitted";
    }
}
```

<body>

제목 : \${newArticleCommand.title}

</body>

커멘드객체는 자동으로 모델에 추가되므로  
클래스이름을 이용해서 접근할 수 있다.  
(단, 첫글자는 소문자이다)

# 뷰에서 사용할 모델의 이름 변경

- @ModelAttribute 어노테이션을 이용해서 커맨드 객체의 모델 이름을 지정 할 수 있다.

```
@Controller
@RequestMapping("/article/newArticle.do")
public class NewArticleController {

    @RequestMapping(method = RequestMethod.POST)
    public String submit(
        @ModelAttribute("command") NewArticleCommand command) {
        //
        return "article/newArticleSubmitted";
    }
}
```

```
> <body>
제목 : ${command.title }
... ..
```

# 커맨드 객체로 List 받기

- 스프링 MVC는 List타입의 프로퍼티에 대한 바인딩도 처리해준다.

```
public class OrderCommand {  
  
    private List<OrderItem> orderItems;  
    private Address address;  
  
    public List<OrderItem> getOrderItems() {  
        return orderItems;  
    }  
  
    public void setOrderItems(List<OrderItem> orderItems) {  
        this.orderItems = orderItems;  
    }  
  
    public Address getAddress() {  
        return address;  
    }  
  
    public void setAddress(Address address) {  
        this.address = address;  
    }  
}
```



# List 타입의 프로퍼티에 값 전달

```
<form method="post">
  상품1: ID - <input type="text" name="orderItems[0].itemId" />
  개수 - <input type="text" name="orderItems[0].number" />
  주의 - <input type="text" name="orderItems[0].remark" />
  <br/>
  상품2: ID - <input type="text" name="orderItems[1].itemId" />
  개수 - <input type="text" name="orderItems[1].number" />
  주의 - <input type="text" name="orderItems[1].remark" />
  .
  <input type="submit" />
```

---

```
@Controller
@RequestMapping("/order/order.do")
public class OrderController {
    @RequestMapping(method = RequestMethod.POST)
    public String submit(OrderCommand orderCommand) {
        return "order/orderCompletion";
    }
}
```

```
public class OrderCommand {  
  
    private List<OrderItem> orderItems;  
    private Address address;  
}
```

스프링 2.5 버전까지는 List 타입의 프로퍼티를 미리 초기화 해 주어야 List 타입의 프로퍼티에 올바르게 값을 전달 할 수 있었지만, 스프링 3 버전 부터 List타입을 초기화 해 주지 않아도 스프링이 알아서 알맞게 처리 해 준다.

# 컨트롤러 메서드의 파라미터 타입

- 컨트롤러의 @RequestMapping 어노테이션이 적용된 메서드는 커맨드 클래스뿐만 아니라 HttpServletRequest, HttpSession, Locale 등 웹 어플리케이션과 관련된 다양한 타입의 파라미터를 가질 수 있다.

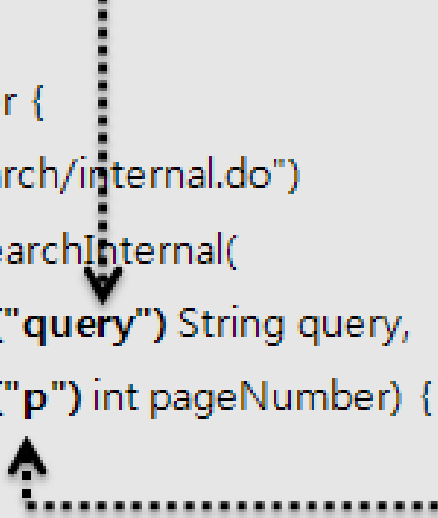
# 컨트롤러 메서드의 파라미터 타입

파라미터 타입	설명
HttpServletRequest, HttpServletResponse, HttpSession	서블릿 API
Java.util.Locale	현재 요청에 대한 Locale
InputStream, Reader	요청 콘텐츠에 직접 접근할 때 사용
OutputStream, Writer	응답 콘텐츠를 생성할 때 사용
@PathVariable 어노테이션 적용 파라미터	URI 템플릿 변수에 접근할 때 사용
@RequestParam 어노테이션 적용 파라미터	HTTP 요청 파라미터를 매핑
@RequestHeader 어노테이션 적용 파라미터	HTTP 요청 헤더를 매핑
@CookieValue 어노테이션 적용 파라미터	HTTP 쿠키 매핑
@RequestBody 어노테이션 적용 파라미터	HTTP 요청의 몸체 내용에 접근할 때 사용, HttpMessageConvert를 이용해서 HTTP 요청 데이터를 해당 타입으로 변환한다.
Map, Model, ModelMap	뷰에 전달할 모델 데이터를 설정할 때 사용
커맨드 객체	HTTP 요청 파라미터를 저장한 객체, 기본적으로 클래스 이름을 모델명으로 사용, @ModelAttribute 어노테이션을 사용하여 모델명을 설정할 수 있다.
Errors, BindingResult	HTTP 요청 파라미터를 커맨드 객체에 저장한 결과, 커맨드 객체를 위한 파라미터 바로 다음에 위치.
SessionStatus	폼 처리를 완료 했음을 처리하기 위해 사용. @sessionAttribute 어노테이션을 명시한 session 속성을 제거하도록 이벤트를 발생시킨다.

## @RequestParam 어노테이션을 이용한 파라미터 매핑

- 컨트롤러를 구현하면서 가장 많이 사용되는 어노테이션
- http 요청 파라미터를 메서드의 파라미터로 전달받을 때 사용된다.

http://host/chap06/search/internal.do?query=spring&p=3



```
@Controller
public class SearchController {
    @RequestMapping("/search/internal.do")
    public ModelAndView searchInternal(
        @RequestParam("query") String query,
        @RequestParam("p") int pageNumber) {
        ...
    }
}
```

```
@Controller
public class SearchController {
    |
    @RequestMapping("/search/internal.do")
    public ModelAndView searchInternal(
        @RequestParam("query") String query,
        @RequestParam("p") int pageNumber) {
        System.out.println("query=" + query + ",pageNumber=" + pageNumber);
        return new ModelAndView("search/internal");
    }
}
```

http://localhost:8080/chap06/search/internal.do?query=spring?&p=a

타입이 맞지 않아 오류 발생

http://localhost:8080/chap06/search/internal.do?query=spring

매개변수의 개수가 맞지 않아 오류 발생

# 생략가능한 파라미터의 설정

```
@Controller
public class SearchController {

    @RequestMapping("/search/internal.do")
    public ModelAndView searchInternal(
        @RequestParam(value="query", required=false) String query,
        @RequestParam(value="p", required=false) int pageNumber) {
        System.out.println("query=" + query + ", pageNumber=" + pageNumber);
        return new ModelAndView("search/internal");
    }
}
```

→ null을 할당할수 없는 기본 자료형의 경우 에러발생

## 기본자료형에 defaultValue를 설정한다.

```
@RequestMapping("/search/internal.do")
public ModelAndView searchInternal(
    @RequestParam(value="query", required=false) String query,
    @RequestParam(value="p", defaultValue="1") int pageNumber) {
    System.out.println("query=" + query + ",pageNumber=" + pageNumber);
    return new ModelAndView("search/internal");
}
```

<http://localhost:8080/chap06/search/internal.do?query=spring>

요청시에 p의 값을 생략하면 default값이 적용된다



# @CookieValue 어노테이션을 이용한 쿠키 매핑

- @CookieValue 어노테이션을 이용하여 쿠키값을 파라미터를 전달 받을 수 있다.

```
@Controller
public class CookieController {

    @RequestMapping("/cookie/view.do")
    public String view(
        @CookieValue("auth") String auth) {
        System.out.println("auth 쿠키: " + auth);
        return "cookie/view";
    }
}
```

required 속성의 기본값은 true이므로  
해당 쿠키가 존재하지 않으면 500에러 발생

```
@RequestMapping("/cookie/view.do")
public String view(
    @CookieValue(value="auth", required=false) String auth) {
    System.out.println("auth 쿠키: " + auth);
    return "cookie/view";
}
```

```
@RequestMapping("/cookie/view.do")
public String view(
    @CookieValue(value="auth", defaultValue="0") String auth) {
    System.out.println("auth 쿠키: " + auth);
    return "cookie/view";
}
```

# @RequestHeader 어노테이션을 이용한 헤더 매핑

- @RequestHeader 를 이용하여 HTTP 요청 헤더의 값을 메서드의 파라미터로 전달 받을 수 있다.

```
@Controller
public class HeaderController {

    @RequestMapping("/header/check.do")
    public String check(@RequestHeader("Accept-Language") String languageHeader) {
        System.out.println(languageHeader);
        return "header/pass";
    }
}
```

# 서블릿 API 직접 사용

- @RequestMapping 어노테이션이 적용되는 메서드는 다음의 파라미터를 전달받을 수 있다.
  - javax.servlet.http.HttpServletRequest
  - javax.servlet.http.HttpServletResponse
  - javax.servlet.http.HttpSession
  - javax.servlet.ServletRequest
  - javax.servlet.ServletResponse

# 서블릿 API를 사용 하는 경우

스프링 MVC가 제공하는 어노테이션을 이용해서 요청 파라미터,헤더,쿠키,세션정보에 접근할 수 있기 때문에 다음의 경우를 제외하고 직접 서블릿API를 사용해야 하는 경우는 드물다.

- HttpSession의 생성을 직접 제어해야 하는 경우
- 컨트롤러에서 쿠키를 생성해야 하는 경우
- 서블릿 API 사용을 선호하는 경우

```
@RequestMapping("/someUrl")
public ModelAndView process(HttpServletRequest request, ...)
{
    if(someCondition)
    {
        HttpSession session = request.getSession();
    }
    ..
}
```

# 컨트롤러 메서드의 리턴 타입

리턴타입	설명
<b>ModelAndView</b>	뷰 정보 및 모델 정보를 담고 있는 ModelAndView 객체
<b>Model</b>	뷰에 전달할 객체 정보를 담고 있는 Model을 리턴한다. 이때 뷰 이름은 요청 URL로부터 결정된다. (RequestToViewNameTranslator를 통해 뷰 결정)
<b>Map</b>	뷰에 전달할 객체 정보를 담고 있는 Map를 리턴한다. 이때 뷰 이름은 요청 URL로부터 결정된다.(RequestToViewNameTranslator를 통해 뷰 결정)
String	뷰 이름을 리턴한다.
View 객체	View 객체를 직접 리턴. 해당 View 객체를 이용해서 뷰를 생성한다.
void	메서드가 ServletResponse나 HttpServletResponse 타입의 파라미터를 갖는 경우 메서드가 직접 응답 처리한다고 가정한다. 그렇지 않을 경우 URL로부터 결정된 뷰를 보여준다. (RequestToViewNameTranslate를 통해 뷰 결정)
@ResponseBody 어노테이션 적용	메서드에서 @ResponseBody 어노테이션이 적용된 경우, 리턴 객체를 HTTP 응답으로 전송한다. HttpMessageConvert를 이용해서 객체를 HTTP 응답 스트림으로 변환한다.

# 컨트롤러 클래스 자동 스캔

- @Controller 어노테이션은 @Component 어노테이션과 마찬가지로 컴포넌트 스캔 대상이다.
- `<context:component-scan>` 태그를 이용해서 @Controller 어노테이션이 적용된 컨트롤러 클래스를 **자동으로 로딩**할 수 있다.

```
<context:component-scan |  
    base-package="madvirus.spring.chap06.controller"/>
```

# 뷰 지정

- 뷰 이름 명시적 지정(ModelAndView와 String 리턴 타입)
- 뷰 이름 자동 지정
- 리다이렉트 뷰



# 뷰 이름 명시적 지정

- 뷰 이름을 명시적으로 지정하려면 ModelAndView나 String을 리턴한다.

```
@RequestMapping("/index.do")
public ModelAndView index() {
    ModelAndView mav = new ModelAndView("index");
    //
    return mav;
}
```

---

```
ModelAndView mav = new ModelAndView();
mav.setViewName("search/game");
```

---

```
@RequestMapping("/help/main.do")
public String helpMain(ModelMap model) {
    //
    return "help/main";
}
```

---

# 뷰 이름 자동 지정

- 리턴 타입이 Model이나 Map인 경우
- 리턴타입이 void 이면서 ServletResponse나 HttpServletResponse 타입의 파라미터가 없는 경우

```
@RequestMapping("/search/game2.do")
public Map<String, Object> search()
{
    HashMap<String, Object> model = new HashMap<String, Object>();
    //
    return model;
}
```

/search/game2.do -> search/game2

뷰에 전달할 모델을 갖고 있는 Map을 리턴하는 경우  
RequestToViewNameTranslate이 URL로 부터 뷰의 이름을 결정한다.

\*\* 전체 경로 사용여부에 따라 뷰 이름 결정 방식이 달라짐  
(요청 URI 매칭에서 설명)

# 리다이렉트 뷰

- 뷰 이름에 "redirect:" 접두어를 붙여 지정한 페이지로 리다이렉트 시킨다.
- Redirect:/bbs/list
  - 현재 서블릿 컨텍스트에 대한 상대적인 경로로 리다이렉트
- Redirect:http://host/list
  - 지정한 절대 URL로 리다이렉트

```
ModelAndView mav = new ModelAndView();  
mav.setViewName("redirect:/error.do");  
return mav;
```

# 모델 생성하기

- 뷰에 전달되는 모델 데이터
- Map, Model, ModelMap을 통한 모델
- ModelAndView를 통한 모델 설정
- @ModelAttribute 어노테이션을 이용한 모델 데이터 처리

# 뷰에 전달 되는 모델 데이터의 예제

```
@Controller
public class GameSearchController {
    private SearchService searchService;

    @ModelAttribute("searchTypeList")
    public List<SearchType> referenceSearchTypeList() {
        List<SearchType> options = new ArrayList<SearchType>();
        options.add(new SearchType(1, "전체"));
        options.add(new SearchType(2, "아이템"));
        options.add(new SearchType(3, "캐릭터"));
        return options;
    }

    @RequestMapping("/search/game.do")
    public ModelAndView search(@ModelAttribute("command") SearchCommand command,
        ModelMap model) {
        String[] geuryList = getPopularQueryList();
        model.addAttribute("popularQueryList", geuryList);

        ModelAndView mav = new ModelAndView("search/game");
        SearchResult result = searchService.search(command);
        mav.addObject("searchResult", result);
        return mav;
    }
}
```

```
public class SearchService {  
    public SearchResult search(SearchCommand command) {  
        return new SearchResult();  
    }  
}
```

```
public class SearchCommand {  
  
    private String type;  
    private String query;  
    private int page;  
  
    //setter,getter
```

```
public class SearchType {  
  
    private int code;  
    private String text;  
  
    public SearchType(int code, String text) {  
        this.code = code;  
        this.text = text;  
    }  
    //setter, getter..
```

```
<body>
인기 키워드:
<c:forEach var="popularQuery" items="${popularQueryList}">
    ${popularQuery}
</c:forEach>
<form action="game.do">
<select name="type">
    <c:forEach var="searchType" items="${searchTypeList}">
        <option value="${searchType.code}"
            <c:if test="${command.type == searchType.code}">selected</c:if>>
            ${searchType.text}
        </option>
    </c:forEach>
</select>
<input type="text" name="query" value="${command.query}" />
<input type="submit" value="검색" />
</form>
검색 결과: ${searchResult}
</body>
```

## **@ModelAttribute 어노테이션을 이용한 데이터 처리**

- @RequestMapping 어노테이션이 적용되지 않은 별도 메서드로 모델에 추가할 객체를 생성
- 커맨드 객체의 초기화 작업을 수행



# @ModelAttribute을 이용한 참조 데이터 생성

- 동일한 모델 데이터를 두 개 이상의 요청처리 화면에 보여줘야 할 경우
- 공통 모델 데이터를 설정 해 주는 메서드를 구현한 뒤
- 요청 처리 메서드에서 호출하도록 구현한다.

```

private void referenceData(Model model)
{
    //..
    model.addAttribute("searchTypeList", searchList);
    model.addAttribute("popularQueryList", queryList);
}

@RequestMapping("/search/main.do")
public String main(Model model) {
    referenceData(model);
    return "search/main";
}

@RequestMapping("/search/game.do")
public ModelAndView search(
    @ModelAttribute("command") SearchCommand command,
    Model model) {
    referenceData(model);

    ModelAndView mav = new ModelAndView("search/game");
    SearchResult result = searchService.search(command);
    mav.addObject("searchResult", result);
    return mav;
}

```

- Model 이나 ModelMap과 같이 모델 정보를 설정할 때 사용할 타입이 동일해야 한다.
- 각 요청 처리 메서드에 공통으로 사용되는 모드를 설정하기 위한 코드가 중복된다.

```
@ModelAttribute("searchTypeList")
```

```
public List<SearchType> referenceSearchTypeList() {  
    List<SearchType> options = new ArrayList<SearchType>();  
    options.add(new SearchType(1, "전체"));  
    options.add(new SearchType(2, "아이템"));  
    options.add(new SearchType(3, "캐릭터"));  
    return options;  
}
```

main메서드와 search메서드에서  
referenceSearchTypeList와  
popularQueryList를 호출하지 않아도  
@ModelAttribute이 설정되어 있으므로  
@ModelAttribute이 설정된 메소드가 반환하  
는 모델에 접근할 수 있다.

```
@ModelAttribute("popularQueryList")
```

```
public String[] getPopularQueryList() {  
    return new String[] { "게임", "창천2", "위" };  
}
```

```
@RequestMapping("/search/main.do")
```

```
public String main() {  
    return "search/main";  
}
```

인기 키워드:

```
<c:forEach var="popularQuery" items="{popularQueryList}">  
    ${popularQuery}  
</c:forEach>
```

```
<form action="game.do">
```

```
<select name="type">
```

```
<c:forEach var="searchType" items="{searchTypeList}">  
    <option value="${searchType.code}">${searchType.text}</option>  
</c:forEach>
```

```
</select>
```

```
@RequestMapping("/search/game.do")
```

```
public ModelAndView search(@ModelAttribute("command") SearchCommand command,  
    ModelMap model) {  
    String[] queryList = getPopularQueryList();  
    model.addAttribute("popularQueryList", queryList);  
  
    ModelAndView mav = new ModelAndView("search/game");  
    SearchResult result = searchService.search(command);  
    mav.addObject("searchResult", result);  
    return mav;  
}
```

# 커맨드 객체 초기화

- @ModelAttribute 어노테이션을 사용하면 GET 요청과 POST 요청에 대해 알맞게 커맨트 객체를 초기화 할 수 있다.

```
@Controller
@RequestMapping("/account/create.do")
public class CreateAccountController {
    @ModelAttribute("command")
    public MemberInfo formBacking(HttpServletRequest request) {
        if (request.getMethod().equalsIgnoreCase("GET")) {
            MemberInfo mi = new MemberInfo();
            Address address = new Address();
            address.setZipcode(autoDetectZipcode(request.getRemoteAddr()));
            mi.setAddress(address);
            return mi;
        } else {
            return new MemberInfo();
        }
    }

    @RequestMapping(method = RequestMethod.GET)
    public String form() {
        return "account/creationForm";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String submit(@ModelAttribute("command") MemberInfo memberInfo,
        BindingResult result) {
        new MemberInfoValidator().validate(memberInfo, result);
        if (result.hasErrors()) {
            return "account/creationForm";
        }
        return "account/created";
    }

    private String autoDetectZipcode(String remoteAddr) {
        return "000000";
    }
}
```

@ModelAttribute 어노테이션이 적용된 메서드가  
@RequestMapping 어노테이션이 적용된 메서드 보다 먼저  
호출되기 되면 formBacking 메서드가 생성한 모델의 정보  
가 submit메서드의 첫번째 파라미터로 전달된다.

# @ModelAttribute 어노테이션 적용 메서드에 전달 가능한 파라미터 타입

- @ModelAttribute 어노테이션이 적용된 메서드는 @RequestMapping 어노테이션이 적용된 메서드와 동일한 타입의 파라미터를 가질 수 있다.

```
@ModelAttribute("command")
public MemberInfo formBacking(HttpServletRequest request) {
    if (request.getMethod().equalsIgnoreCase("GET")) {
        MemberInfo mi = new MemberInfo();
        Address address = new Address();
        address.setZipcode(autoDetectZipcode(request.getRemoteAddr()));
        mi.setAddress(address);
        return mi;
    } else {
        return new MemberInfo();
    }
}
```

@ModelAttribute 어노테이션 적용 메서드는 HttpServletRequest 뿐만 아니라 필요에 따라 Locale, @RequestParam 어노테이션 적용 파라미터 @PathVariable 어노테이션 적용 파라미터를 이용하여 모델 객체를 생성하는데 필요한 정보를 구할 수 있다.

# 요청 URI 매칭

- 전체 경로와 서블릿 기반 경로 매칭 설정
- @PathVariable 어노테이션을 이용한 URI 템플릿
- @RequestMapping 어노테이션의 추가 설정 방법

# 전체 경로와 서블릿 기반 경로 매칭 설정

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>*.do</url-pattern>
  <url-pattern>/game/*</url-pattern>
</servlet-mapping>
```

```
@RequestMapping("/search/game.do")
public String search()
{

}

@RequestMapping("/game/info")
public String info()
{

}
```

<url-pattern>의 값을 /game/\*으로 설정 했기 때문에 /game/info의 요청은 처리 되지 않는다.  
<url-pattern>의 값을 디렉토리를 포함한 패턴으로 지정하게 되면 서블릿 경로는 /game 되며 서블릿 경로를 제외한 나머지 경로를 이용해서 판단하게 된다.  
@RequestMapping 어노테이션 값은 /game/info 이므로 요청 URL info와는 매칭 되지 않는다.



# 서블릿 경로를 포함한 전체 경로 이용

```
<bean  
  class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"  
  p:alwaysUseFullPath="true"/>
```

```
<bean  
  class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"  
  p:alwaysUseFullPath="true"/>
```

```
<servlet-mapping>  
  <servlet-name>dispatcher</servlet-name>  
  <url-pattern>*.do</url-pattern>  
  <url-pattern>/game/*</url-pattern>  
</servlet-mapping>
```

```
@RequestMapping("/game/info")  
public String info()  
{  
}
```

@PathVariable 어노테이션을 이용한 URI 템플릿

- <http://somehost/users/userinfo?id=simon>
- <http://somehost/users/simon>
- URI에 아이디나 이름등이 포함되도록 URI를 구성
- @RequestMapping 어노테이션 값으로 {템플릿 변수}를 사용
- @PathVariable 어노테이션을 이용 {템플릿변수}와 동일한 이름을 갖는 파라미터를 추가한다.

```
@Controller
public class CharacterInfoController1 {

    @RequestMapping("/game/users/{userId}/characters/{characterId}")
    public String characterInfo(@PathVariable String userId,
                               @PathVariable int characterId, ModelMap model) {
        model.addAttribute("userId", userId);
        model.addAttribute("characterId", characterId);
        return "game/character/info";
    }
}
```



http://localhost/chap06/game/users/simon/charater/1

userId와 characterId의 값은 각각 simon과 1이 된다.

# @RequestMapping 어노테이션의 추가 설정 방법

```
@Controller
@RequestMapping("/game/users/{userId}")
public class CharacterInfoController {

    @RequestMapping("/characters/{characterId}")
    public String characterInfo(@PathVariable String userId,
                               @PathVariable int characterId, ModelMap model) {
        model.addAttribute("userId", userId);
        model.addAttribute("characterId", characterId);
        return "game/character/info";
    }
}
```

characterInfo 메서드에 적용된 어노테이션의 값은 클래스에 적용된 어노테이션 값을 포함한 /game/user/{userId}/characters/{characterId}가 된다.

# 폼 입력 값 검증

스프링 MVC에서 폼 값을 검증하고 에러를 처리하도록 Validator인터페이스와 Errors클래스를 제공한다.

- Validator 인터페이스를 이용한 폼 값 검증
- Errors 인터페이스와 BindingResult 인터페이스
- DefaultMessageCodesResolver와 에러 메시지
- ValidationUtils 클래스를 이용한 값 검증
- @Valid 어노테이션과 @InitBinder 어노테이션을 이용한 검증실행

# Validator 인터페이스를 이용한 폼 값 검증

- `boolean support(Class)`
  - 해당 클래스에 대한 값 검증을 지원하는지의 여부를 리턴한다.
- `void validate(Object target, Errors errors)`
  - Target 객체에 대한 검증을 실행한다.

## Validation의 구현 예

```
public class MemberInfoValidator implements Validator {  
    @Override  
    public boolean supports(Class<?> clazz) {  
        return MemberInfo.class.isAssignableFrom(clazz);  
    }  
}
```

대상 클래스가 MemberInfo 또는  
하위 클래스 인지 검사

@Override

```
public void validate(Object target, Errors errors) {
```

```
    MemberInfo memberInfo = (MemberInfo) target;
```

```
    if (memberInfo.getId() == null || memberInfo.getId().trim().isEmpty()) {
```

```
        errors.rejectValue("id", "required");  
    }
```

```
    if (memberInfo.getName() == null || memberInfo.getName().trim().isEmpty()) {
```

```
        errors.rejectValue("name", "required");  
    }
```

```
    Address address = memberInfo.getAddress();
```

```
    if (address == null) {
```

```
        errors.rejectValue("address", "required");  
    }
```

```
    if (address != null) {
```

```
        errors.pushNestedPath("address");
```

```
        try {
```

```
            if (address.getZipcode() == null
```

```
                || address.getZipcode().trim().isEmpty()) {
```

```
                errors.rejectValue("zipcode", "required");  
            }
```

```
            if (address.getAddress1() == null
```

```
                || address.getAddress1().trim().isEmpty()) {
```

```
                errors.rejectValue("address1", "required");  
            }
```

```
        }
```

```
    } finally {
```

```
        errors.popNestedPath();  
    }
```

각 프로퍼티의 값을 검증하여 올바  
르지 않을 경우 뷰에서 출력할 에러  
메시지를 추가한다.

## Validator을 적용하는 Controller의 예

```
@Controller
@RequestMapping("/account/create.do")
public class CreateAccountController {
    @ModelAttribute("command")
    public MemberInfo formBacking(HttpServletRequest request) {
        if (request.getMethod().equalsIgnoreCase("GET")) {
            MemberInfo mi = new MemberInfo();
            Address address = new Address();
            address.setZipcode(autoDetectZipcode(request.getRemoteAddr()));
            mi.setAddress(address);
            return mi;
        } else {
            return new MemberInfo();
        }
    }

    @RequestMapping(method = RequestMethod.GET)
    public String form() {
        return "account/creationForm";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String submit(@ModelAttribute("command") MemberInfo memberInfo,
        BindingResult result) {
        new MemberInfoValidator().validate(memberInfo, result);
        if (result.hasErrors()) {
            return "account/creationForm";
        }
        return "account/created";
    }

    private String autoDetectZipcode(String remoteAddr) {
        return "000000";
    }
}
```



## Errors 인터페이스와 BindingResult 인터페이스

- Erros : 유효성 검증 결과를 저장할 때 사용
- BindingResult : 폼 값을 커맨드 객체에 바인딩 한 결과를 저장하고 에러 코드로 부터 에러 메시지를 가져옴.

# Errors 인터페이스의 메서드

`void reject(String errorCode)`

Register a global error for the entire target object, using the given error description.

`void reject(String errorCode, Object[] errorArgs, String defaultMessage)`

Register a global error for the entire target object, using the given error description.

`void reject(String errorCode, String defaultMessage)`

Register a global error for the entire target object, using the given error description.

`void rejectValue(String field, String errorCode)`

Register a field error for the specified field of the current object (respecting the current locale), using the given error description.

`void rejectValue(String field, String errorCode, Object[] errorArgs, String defaultMessage)`

Register a field error for the specified field of the current object (respecting the current locale), using the given error description.

`void rejectValue(String field, String errorCode, String defaultMessage)`

Register a field error for the specified field of the current object (respecting the current locale), using the given error description.

```
@RequestMapping(method = RequestMethod.POST)
public String submit(@Valid LoginCommand loginCommand,
    BindingResult result) {
    if (result.hasErrors()) {
        return formViewName;
    }
    try {
        authenticator.authenticate(loginCommand);
        return "redirect:/index.jsp";
    } catch (AuthenticationException e) {
        result.reject("invalidIdOrPassword", new Object[] { loginCommand
            .getUserId() }, null);
        return formViewName;
    }
}
```

아이디나 암호필드 자체에 에러가 있기 보다는 객체 자체에 문제가 있을 경우, reject 메서드를 이용하여 글로벌 에러정보를 추가함.

# 객체의 개별 프로퍼티에 대한 에러정보 추가

```
@Override
public void validate(Object target, Errors errors) {
    MemberInfo memberInfo = (MemberInfo) target;
    if (memberInfo.getId() == null || memberInfo.getId().trim().isEmpty()) {
        errors.rejectValue("id", "required");
    }
    if (memberInfo.getName() == null || memberInfo.getName().trim().isEmpty()) {
        errors.rejectValue("name", "required");
    }
}
```

rejectValue() 메서드는 객체의 개별 필드에 대한 에러 정보를 추가할 때 사용된다.

```
errors.rejectValue("id", "id.invalidLength");
errors.rejectValue("id", "id.invalidCharacter");
```

특정 필드에 대한 에러정보를 두 번 이상 등록할 수 있다.

# Errors 인터페이스의 에러 발생 여부 확인 메서드

boolean hasErrors()

Int getErrorCount()

Boolean hasGlobalErrors()

Int getGlobalErrorCount()

Boolean hasFieldErrors()

Int getFieldErrorCount()

boolean hasFieldErrros(String field)

Int getFieldErrorCount(String field)

# 에러 발생 여부 확인 예

```
@Controller
@RequestMapping("/account/create.do")
public class CreateAccountController {
    @RequestMapping(method = RequestMethod.POST)
    public String submit(@ModelAttribute("command") MemberInfo memberInfo,
        BindingResult result) {
        new MemberInfoValidator().validate(memberInfo, result);
        if (result.hasErrors()) {
            return "account/creationForm";
        }
        return "account/created";    }
}
```

폼 검증 결과 에러가 존재할 경우 다시 폼을 보여 주도록 처리 하기 위해 hasErrors() 메소드를 사용.

# @Valid 어노테이션과 @InitBinder 어노테이션을 이용한 검증 실행

@Valid 어노테이션은 연관된 객체의 유효성을 검증한다는 것을 표시  
스프링 3 MVC는 JSR 303의 @Valid 어노테이션과 스프링 프레임워크의  
@InitBinder 어노테이션을 이용해서 Validator에 대한 직접 호출 없이 유효  
성 검사코드를 실행하도록 한다.

# @InitBinder 어노테이션 사용 코드 작성 예

```
@Controller
@RequestMapping("/login/login.do")
public class LoginController {
    private String formViewName = "login/form";

    @Autowired
    private Authenticator authenticator;

    @RequestMapping(method = RequestMethod.GET)
    public String form() {
        return formViewName;
    }

    @ModelAttribute
    public LoginCommand formBacking() {
        return new LoginCommand();
    }

    @RequestMapping(method = RequestMethod.POST)
    public String submit(@Valid LoginCommand loginCommand,
        BindingResult result) {
        if (result.hasErrors()) {
            return formViewName;
        }
        try {
            authenticator.authenticate(loginCommand);
            return "redirect:/index.jsp";
        } catch (AuthenticationException e) {
            result.reject("invalidIdOrPassword", new Object[] { loginCommand
                .getUserId() }, null);
            return formViewName;
        }
    }

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(new LoginCommandValidator());
    }

    public void setAuthenticator(Authenticator loginService) {
        this.authenticator = loginService;
    }
}
```



# Spring에서 세션의 사용

```
@Controller
@SessionAttributes("id")
public class JoinController {

    @ModelAttribute("name")
    public String pro()
    {
        return "홍길동";
    }

    @RequestMapping(value="/join.do", method=RequestMethod.GET)
    public ModelAndView form()
    {
        ModelAndView view = new ModelAndView();
        view.setViewName("joinForm");
        view.addObject("title", "회원 가입 양식");
        view.addObject("id", "tiger");
        return view;
    }
}
```

\*\* 현재 컨트롤러 내에서 "id"라는 이름의 모델의 값을 세션으로 저장

이후의 모든 뷰 페이지에서 이 세션값에 접근이 가능함.

```
<html>
<head>
<meta http-equiv="Content-Type"
<title>Insert title here</title>
</head>
<body>
아이디 : ${id }
</body>
</html>
```

\*\* 세션의 파기 ->

```
@Controller
public class LogoutController {

    @RequestMapping("/logOut.do")
    public ModelAndView logOut(HttpSession session)
    {
        session.invalidate();
        return new ModelAndView("logOut");
    }
}
```