




# 제10장 스프링 웹 MVC를 활용한 애플리케이션 작성 스텝

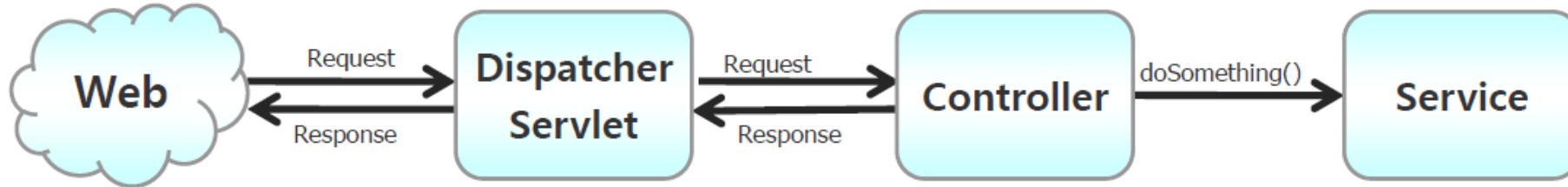
- 컨트롤러 작성
  - View 지정
  - Model 생성하기
  - 요청 URI 매칭
  - 폼 입력 값 검증
- 

# Spring MVC를 이용한 애플리케이션 작성 스텝

- 1.컨트롤러 작성
- 2.컨텍스트 설정 파일([ServletName]-servlet.xml)에 컨트롤러 설정
- 3.컨트롤러와 JSP의 연결 위해 View Resolver 설정
- 4.JSP 코드 작성
- 5.실행

# 컨트롤러 작성(1)

- ▶ 좋은 디자인은 컨트롤러가 많은 일을 하지 않고 서비스에 처리를 위임



# 컨트롤러 작성(2)

## ▶ 컨트롤러 클래스 작성

```
@Controller
public class HelloController {

    @RequestMapping("hello.do")
    public String sayHello(Model model) {
        model.addAttribute("message", "안녕하세요");
        return "hello";
    }
}
```

HelloController.java

## ▶ 컨텍스트 설정파일([servletName]-servlet.xml)에 컨트롤러 설정

```
<bean id="helloController" class="com.kosta.HelloController" />
```

dispatcher-servlet.xml

# 컨트롤러 작성(3)

## ▶ 컨트롤러와 JSP의 연결 위해 View Resolver 설정

```
<?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">

  <bean id="helloController" class="com.kosta. HelloController">

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

## ▶ jsp코드 작성

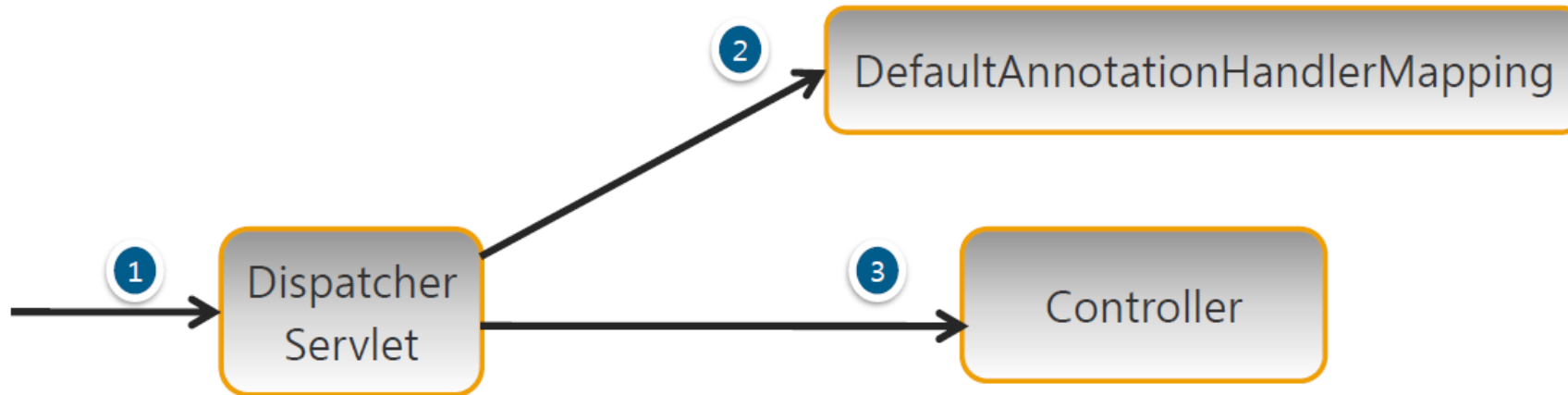
dispatcher-servlet.xml

```
<%@ page language="java" contentType="text/html; charset=EUC-KR"
pageEncoding="EUC-KR"%>
<!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=EUC-KR">
</head>
<body>음 ... ${message}
</body>
</html>
```

hello.jsp

# ○ ○ ○ @Controller 와 @RequestMapping 선언 ○ ○ ○

- ▶ 메소드 단위의 매핑이 가능.
- ▶ DefaultAnnotationHandlerMapping와 AnnotationHandlerAdapter 를 사용함.
- ▶ 기본 설정이므로 별도의 추가 없이 사용 가능



# 컨트롤러 작성(4)

- ▶ 컨트롤러는 클라이언트의 요청을 처리
- ▶ **@Controller 선언**
- ▶ 클래스 타입에 적용
- ▶ 스프링 3.0 버전부터 **@Controller** 사용을 권장

**@Controller**

```
public class BoardController {
```

```
    private BoardService boardService;
```

```
    public void setBoardService(BoardService boardService) {  
        this.boardService = boardService;  
    }
```

```
    @RequestMapping("board/write.do")  
    public String initWrite() {  
        return "board/write";  
    }
```

```
    @RequestMapping("board/list.do")  
    public ModelAndView list() {  
        List<Board> boards = boardService.findBoards();  
        ...  
    }
```

# 컨트롤러 작성(5)

- ▶ 컨트롤러 클래스를 <bean>으로 등록

```
<?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">

  <bean id="boardController" class="com.kosta.board.controller.BoardController">
    <property name="boardService" ref="boardService"></property>
  </bean>

</beans>
```

dispatcher-servlet.xml



# 컨트롤러 작성(6)

- ▶ 컨트롤러 클래스 자동스캔
- ▶ context:component-scan 선언
- ▶ @Controller 애노테이션이 적용된 클래스는 자동 스캔 대상

```
<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.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package="com.kosta.board.controller" />

</beans>
```

dispatcher-servlet.xml

# 컨트롤러 작성(7)

- ▶ @RequestMapping 선언
- ▶ 요청 URL 매핑 정보를 설정
- ▶ 클래스타입과 메소드에 설정 가능

```
@Controller
public class BoardController {

    private BoardService boardService;

    public void setBoardService(BoardService boardService) {
        this.boardService = boardService;
    }

    @RequestMapping("board/write.do")
    public String initWrite() {
        return "board/write";
    }

    @RequestMapping("board/list.do")
    public ModelAndView list() {
        List<Board> boards = boardService.findBoards();
        ...
    }
}
```

```
@Controller
@RequestMapping("board")
public class BoardController {

    private BoardService boardService;

    public void setBoardService(BoardService boardService) {
        this.boardService = boardService;
    }

    @RequestMapping("write.do")
    public String initWrite() {
        return "board/write";
    }

    @RequestMapping("list.do")
    public ModelAndView list() {
        List<Board> boards = boardService.findBoards();
        ...
    }
}
```

# 컨트롤러 작성(8)

- ▶ 컨트롤러 메소드의 HTTP 메소드 한정
  - ▶ 같은 URL의 요청에 대하여 HTTP메소드(POST, GET...)에 따라 서로 다른 메소드를 Mapping할 수 있음

```
@Controller
public class HelloController {

    @RequestMapping(value="hello.do", method=RequestMethod.GET)
    public String hello() {
        return "helloget";
    }

    @RequestMapping(value="hello.do", method=RequestMethod.POST)
    public String hello2() {
        return "hellopost";
    }
}
```

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

    @RequestMapping(method=RequestMethod.GET)
    public String hello() {
        return "helloget";
    }

    @RequestMapping(method=RequestMethod.POST)
    public String hello2() {
        return "hellopost";
    }
}
```

# 컨트롤러 작성(9)

## ▶ 컨트롤러 메소드의 파라미터 타입

| 파라미터 타입                                 | 설명  |
|---|---|
| HttpServletRequest, HttpServletResponse | 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  |
| Map, Model, ModelMap                    | 뷰에 전달할 모델 데이터를 설정할 때 사용   |
| 커맨드 객체                                  | HTTP 요청 파라미터를 저장 한 객체, 기본적으로 클래스 이름을 모델명으로 사용<br>@ModelAttribute 어노테이션 설정으로 모델명을 설정할 수 있음 |
| Errors, BindingResult                   | HTTP 요청 파라미터를 커맨드 객체에 저장한 결과, 커맨드 객체를 위한 파라미터 바로 다음에 위치                                   |
| SessionStatus                           | 폼 처리를 완료 했음을 처리하기 위해 사용. @SessionAttributes 어노테이션을 명시한 session 속성을 제거하도록 이벤트를 발생 시킨다.     |

# 컨트롤러 작성(10)

- ▶ @RequestParam 애노테이션을 이용한 파라미터 매핑

```
@Controller
public class HelloController {

    @RequestMapping("/hello.do")
    public String hello(@RequestParam("name") String name, @RequestParam("age") int age) {
        system.out.println(name);
        system.out.println(age);
        ...
    }
}
```

http://localhost:8080/hello.do?name=kim&age=22

```
@Controller
public class HelloController {

    @RequestMapping("/hello.do")
    public String hello(
        @RequestParam(value="name", required=false) String name,
        @RequestParam(value="age", defaultValue="0") int age) {
        system.out.println(name);
        system.out.println(age);
        ...
    }
}
```

필수여부

기본값

# 컨트롤러 작성(11)

- ▶ 커맨드 객체 이용
  - ▶ 뷰에서 입력한 데이터를 자바 빈 객체를 이용해서 전송

```
@Controller
public class BoardController {

    @RequestMapping("board/saveBoard.do")
    public String save(Board board) {

    }

}
```

```
public class Board {
    private String title;
    private String content;

    public void setTitle(String title) {
        this.title = title;
    }

    public void setContent(String content) {
        this.content = content;
    }

    ...
}
```

```
<form method="post" action="${pageContext.request.contextPath}/board/saveBoard.do">
    제목: <input type="text" name="title" /><br/>
    내용: <textarea name="content"></textarea><br/>
    <input type="submit"/>
</form>
```

# 컨트롤러 작성(12)

## ▶ 커맨드 객체 List로 받기

```
public class OrderCommand {  
    private List<OrderItem> orderItems;  
  
    public void setOrderItems(List<OrderItem> orderItems) {  
        this.orderItems = orderItems;  
    }  
}
```

```
@Controller  
@RequestMapping("/hello.do")  
public class HelloController {  
  
    @RequestMapping(method=RequestMethod.GET)  
    public String hello(OrderCommand command) {  
        ""  
    }  
}
```

```
<form action="hello.do" method="post">  
    <input type="text" name="orderItems[0].itemId" />  
    <input type="text" name="orderItems[0].number" />  
    <input type="text" name="orderItems[0].remark" />  
    <br/>  
    <input type="text" name="orderItems[1].itemId" />  
    <input type="text" name="orderItems[1].number" />  
    <input type="text" name="orderItems[1].remark" />  
    <br/>  
    <input type="submit" />  
</form>
```

# 컨트롤러 작성(13)

## ▶ View에서 커맨드 객체에 접근하기

- ▶ 커맨드 객체는 자동으로 반환되는 모델에 추가됨
- ▶ 컨트롤러의 `@RequestMapping` 어노테이션 메소드에서 전달받은 커맨드 객체에 접근

```
@Controller
public class BoardController {

    @RequestMapping("board/saveBoard.do")
    public String hello(Board command) {
        ...
    }
}
```

```
<body>
...
${board.title}
...
....jsp
```

`@ModelAttribute` 를 사용하여 View에서 사용할 커맨드 객체의 이름 변경

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

    @RequestMapping(method=RequestMethod.GET)
    public String hello(@ModelAttribute("faq") Board board) {
        ...
    }
}
```

```
<body>
...
${faq.title}
...
....jsp
```



# 컨트롤러 작성(14)

## ▶ 서블릿 API 직접 사용

```
javax.servlet.http.HttpServletRequest/javax.servlet.ServletRequest  
javax.servlet.http.HttpServletResponse/javax.servlet.ServletResponse  
javax.servlet.http.HttpSession
```

- ▶ HttpSession의 생성을 직접 제어해야 하는 경우
- ▶ 컨트롤러에서 쿠키를 생성해야 하는 경우
- ▶ 서블릿 API를 선호하는 경우

```
@Controller  
public class HelloController {  
  
    @RequestMapping("/hello.do")  
    public String hello(HttpServletRequest request, HttpServletResponse response) {  
        ...  
    }  
}
```

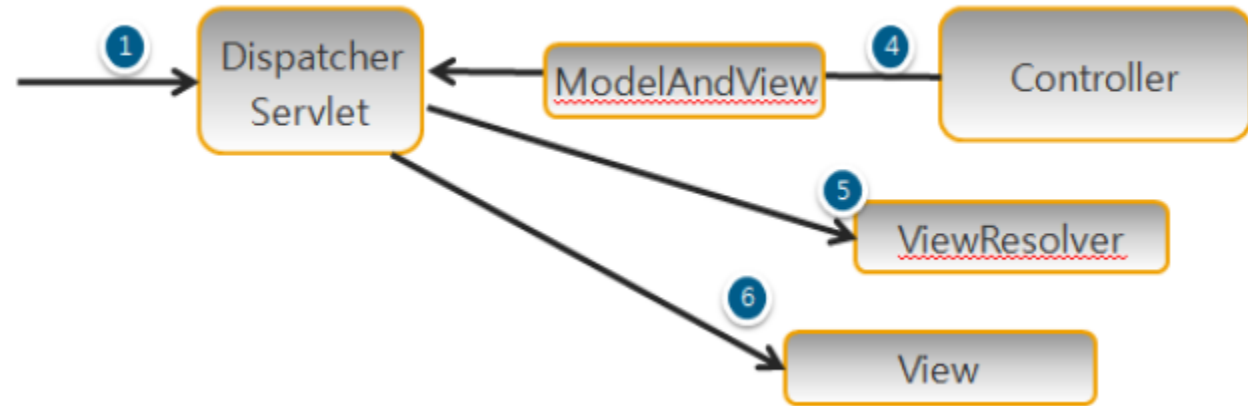
# 컨트롤러 작성(15)

## ▶ 컨트롤러 메소드의 리턴타입

| 리턴 타입                  | 설명  |
|------------------------|---|
| ModelAndView           | 뷰 정보 및 모델 정보를 담고 있는 ModelAndView 객체   |
| Model                  | 뷰에 전달할 객체 정보를 담고 있는 Model을 리턴한다. 이때 뷰 이름은 요청 URL로부터 결정된다.(RequestToViewNameTranslator)  |
| Map                    | 뷰에 전달할 객체 정보를 담고 있는 Map을 리턴한다. 이때 뷰 이름은 요청 URL로부터 결정된다.(RequestToViewNameTranslator)  |
| 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 응답을 처리한다고 가정한다. 그렇지 않을 경우 요청 URL로부터 결정된 뷰를 보여준다.<br>(RequestToViewNameTranslator) |
| @ResponseBody 어노테이션 적용 | 메소드에서 @ResponseBody 어노테이션이 적용된 경우, 리턴 객체를 HTTP 응답으로 전송한다. HttpMessageConverter를 이용해서 객체를 HTTP응답 스트림으로 변환한다.   |

# View 지정(1)

- ▶ 컨트롤러에서는 처리 결과를 보여줄 View 이름이나 객체를 리턴하고, DispatcherServlet은 View 이름이나 View객체를 이용하여 뷰를 생성
  - ▶ 명시적 지정
  - ▶ 자동 지정



- ▶ View Resolver : 논리적 뷰와 실제 JSP파일 매핑

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

*InternalResourceViewResolver는  
prefix + 논리뷰 + suffix로 설정  
예) /WEB-INF/jsp/home.jsp*

# View 지정(2)

- ▶ View 이름 명시적 지정
  - ▶ ModelAndView와 String리턴 타입

```
@Controller
public class HelloController {
    @RequestMapping("/hello.do")
    public ModelAndView hello() {
        ModelAndView mav = new ModelAndView("hello");
        return mav;
    }
}
```

```
@Controller
public class HelloController {

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

```
@Controller
public class HelloController {

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

# View 지정(3)

## ▶ View 자동 지정

- ▶ RequestToViewNameTranslator를 이용하여 URL로부터 View이름을 결정한다.
- ▶ 자동 지정 유형
  - ▶ 리턴 타입이 Model이나 Map인 경우
  - ▶ 리턴 타입이 void 이면서 ServletResponse나 HttpServletResponse 타입의 파라미터가 없는 경우

```
@Controller
public class HelloController {

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

hello 가 뷰 이름이 됨

# View 지정(4)

## ▶ 리다이렉트 뷰

- ▶ View 이름에 `redirect:` 접두어를 붙이면, 지정한 페이지로 리다이렉트 된다.
- ▶ `redirect:/product/productList.do`
- ▶ `redirect:http://localhost/product/productList.do`

```
@Controller
public class HelloController {

    @RequestMapping("/hello.do")
    public String hello() {
        ...
        return "redirect:/product/productList.do";
    }
}
```



# Model 생성하기(1)



- ▶ 뷰에 전달하는 데이터
  - ▶ @RequestMapping 애노테이션이 적용된 메서드의 Map, Model, ModelMap
  - ▶ @RequestMapping 메서드가 리턴하는 ModelAndView
  - ▶ @ModelAttribute 애노테이션이 적용된 메서드가 리턴 한 객체



# Model 생성하기(2)

- ▶ Map, Model, ModelMap을 통한 설정
  - ▶ 파라미터로 받는 방식

```
@Controller
public class HelloController {

    @RequestMapping("/hello.do")
    public String hello(Map model) {
        model.addAttribute("", "");
        ...
    }
}
```

```
@Controller
public class HelloController {

    @RequestMapping("/hello.do")
    public String hello(Model model) {
        ...
    }
}
```



# Model 생성하기(3)

- ▶ Model 인터페이스
  - ▶ Model addAttribute(String name, Object value);
  - ▶ Model addAttribute(Object value);
  - ▶ Model addAllAttributes(Collection<?> values);
  - ▶ Model addAllAttributes(Map<String, ?> attributes);
  - ▶ Model mergeAttributes(Map<String, ?> attributes);
  - ▶ boolean containsAttribute(String name);

# Model 생성하기(4)

- ▶ ModelAndView를 통한 모델 설정
  - ▶ 컨트롤러에서 처리결과를 보여줄 View와 View에 전달할 값(모델)을 저장하는 용도로 사용
  - ▶ `setViewName("viewName")`
  - ▶ `addObject(String name, Object value)`

```
@Controller
public class HelloController {

    @RequestMapping("/hello.do")
    public ModelAndView hello() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("hello");
        mav.addObject("message", "안녕하세요");
        return mav;
    }
}
```

# Model 생성하기(5)

- ▶ @ModelAttribute 애노테이션을 이용한 모델 데이터 처리
  - ▶ @RequestMapping 애노테이션이 적용되지 않은 별도 메서드로 모델에 추가될 객체를 생성

```
@Controller
public class HelloController {

    @ModelAttribute("modelAttrMessage")
    public String getModelAttrMessage() {
        return "bye bye...";
    }

    @RequestMapping("hello.do")
    public String sayHello(Model model) {
        model.addAttribute("message", "안녕하세요~");
        return "hello";
    }
}
```

```
</html>
<body>
    ${message}
    ${modelAttrMessage}
</body>
</html>
```

# ○○○@ModelAttribute 사용해 모델 속성추가하기(1)○○○

- ▶ @ModelAttribute 사용법-name 속성을 지정하지 않음

```
import org.springframework.ui.Model;
....
public class SampleController{
    @ModelAttribute
    public Sample getSample(){
        retrun new Sample();
    }
}
```

- ▶ @ModelAttribute 설정한 메소드가 HttpServletRequest를 인수로 받음

```
@ModelAttribute(name="myObject")
public SomeObject doSomething(HttpServletRequest request){ .... }
```

- ▶ @ModelAttribute 설정한 메소드에 요청 파라미터 전달하기

```
@ModelAttribute(name="myObject")
public SomeObject doSomething(@RequestParam("someArg") String myarg) { .... }
```

# ○○○@ModelAttribute 사용해 모델 속성추가하기(2)○○○

## ▶ Model객체에 직접 모델 속성 추가하기

```
import org.springframework.ui.Model;
....
public class SampleWebController{
    @ModelAttribute
    public void doSomething(Model model){
        model.addAttribute("myobject", new MyObject());
        model.addAttribute("otherobject", new otherObject());
    }
}
```

# 요청 URI 매칭(1)

- ▶ 전체 경로와 서블릿 기반 경로 매칭
  - ▶ DispatcherServlet은 DefaultAnnotationHandlerMapping 클래스를 기본으로 HandlerMapping 구현체로 사용
  - ▶ Default로 컨텍스트 내의 경로가 아닌 서블릿 경로를 제외한 나머지 경로에 대해 매핑

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

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

```
@Controller  
public class GameController {  
  
    @RequestMapping("search/game.do")  
    public String getGame() {  
        return "game/info";  
    }  
  
    @RequestMapping("info.do")  
    public String infoGame() {  
        return "game/info";  
    }  
}
```

# 요청 URI 매칭(2)

- ▶ @PathVariable 애노테이션을 이용한 URI 템플릿
  - ▶ RESTful 방식
    - ▶ http://somehost/users/kim
    - ▶ http://somehost/games/
    - ▶ http://somehost/forum/board1/10
  - ▶ @RequestMapping 애노테이션 값으로 {템플릿변수}를 사용한다.
  - ▶ @PathVariable 애노테이션을 이용해서 {템플릿변수}와 동일한 이름을 갖는 파라미터를 추가한다.

```
@Controller
Public class CharaterInfoController {
    @RequestMapping("/game/users/{userId}/characters/{characterId}")
    public String characterInfo( @PathVariable String userId
    , @PathVariable int characterId, ModelMap model){
        ...
    }
}
```

# 요청 URI 매칭(3)

- ▶ @RequestMapping 애노테이션의 추가설정 방법
  - ▶ @RequestMapping 애노테이션을 클래스와 메소드에 함께 적용하는 경우

```
@Controller
@RequestMapping("game/users/{userId}")
Public class CharaterInfoController {
    @RequestMapping("characters/{characterId}")
    public String characterInfo(@PathVariable String userId
                               , @PathVariable int characterId, ModelMap model)
    ...
}
```

- ▶ Ant 스타일의 URI패턴 지원
  - ▶ ?: 하나의 문자열과 대치
  - ▶ \*: 하나 이상의 문자열과 대치
  - ▶ \*\*: 하나 이상의 디렉토리과 대치

```
@RequestMapping("/members/*.do")
@RequestMapping("/game/*/items/{itemId}")
```



# 폼 입력값 검증(1)

- ▶ 스프링은 유효성 검사를 위한 **Validator** 인터페이스와 유효성 검사 결과를 저장할 **Errors** 인터페이스 제공
- ▶ **Validator** 인터페이스를 이용한 폼 입력 값 검증

```
package org.springframework.validation;  
  
public interface Validator {  
  
    boolean supports(Class<?> clazz); → Validator가 해당 클래스에 대한 validation 지원 여부  
  
    void validate(Object target, Errors errors);  
        → validation실행 validation결과 문제가 있는 경우 errors 객체에 문제에 대한 정보를 저장  
  
}
```

# 폼 입력값 검증(2)

## ▶ Validator 구현

```
package com.kosta.board.validator;

import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

import com.kosta.board.domain.Board;

public class BoardValidator implements Validator {

    @Override
    public boolean supports(Class<?> arg0) {
        return Board.class.isAssignableFrom(arg0);
    }

    @Override
    public void validate(Object object, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "title", "required", "제목은 반드시 입력하세요.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "content", "required", "내용을 입력하세요.");
    }
}
```

# 폼 입력값 검증(3)

- ▶ @Valid 애노테이션과 @InitBinder 애노테이션을 이용한 검증 실행
  - ▶ @Valid : 스프링 프레임워크가 validation을 호출하도록 애노테이션을 이용하여 설정함
  - ▶ @InitBinder : Validator를 바인딩

```
@RequestMapping("saveBoard.do")
public ModelAndView saveBoard(@Valid Board board, BindingResult result) {
    ModelAndView mav = new ModelAndView();
    if (result.hasErrors()) {
        mav.setViewName("board/write");
        mav.addObject("board", board);
        return mav;
    }
    boardService.saveBoard(board);
    mav.setViewName("board/list");
    mav.addObject("boards", boardService.findBoards());
    return mav;
}

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

# 폼 입력값 검증(4)

- ▶ Validation 결과 출력
  - ▶ spring 커스텀 태그를 사용

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

...
...
<spring:hasBindErrors name="board" /> ← 커스텀 태그를 이용하여 에러 정보를 설정
<form action="saveBoard.do" method="post">
<input type="hidden" name="number" value="${(board.number == '' || board.number == null) ? -1 : board.number}" />
<table>
<tr>
<td>제목</td>
<td><input type="text" name="title" value="${board.title}" /><form:errors path="board.title" /></td>
</tr>
</table>
</form>
```