



Spring MVC (ver3.4.0)

목차 (Table of Contents)

1. Spring MVC 소개
2. MVC 웹 프로그래밍 I
3. Spring MVC 이해
4. MVC 웹 프로그래밍 II
5. Spring MVC 활용

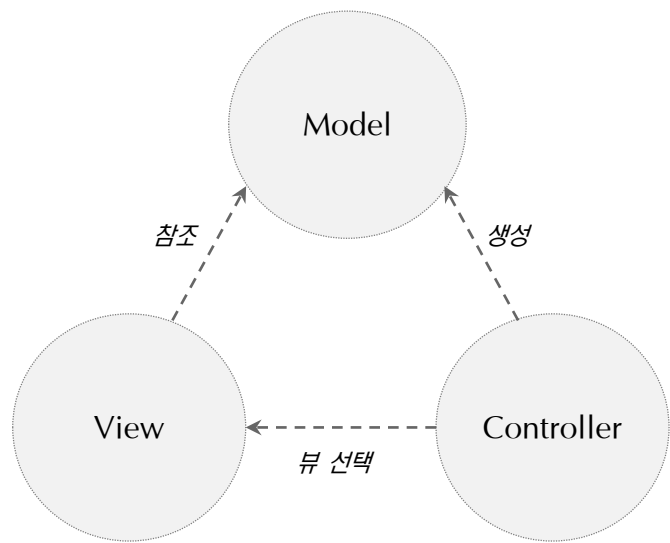


1. Spring MVC 소개

- 1.1 웹 프레임워크 아키텍처
- 1.2 Spring MVC 개요
- 1.3 Spring MVC 특징
- 1.4 Spring @MVC
- 1.5 요약

1.1 웹 프레임워크 아키텍처 (1/2) – MVC 패턴

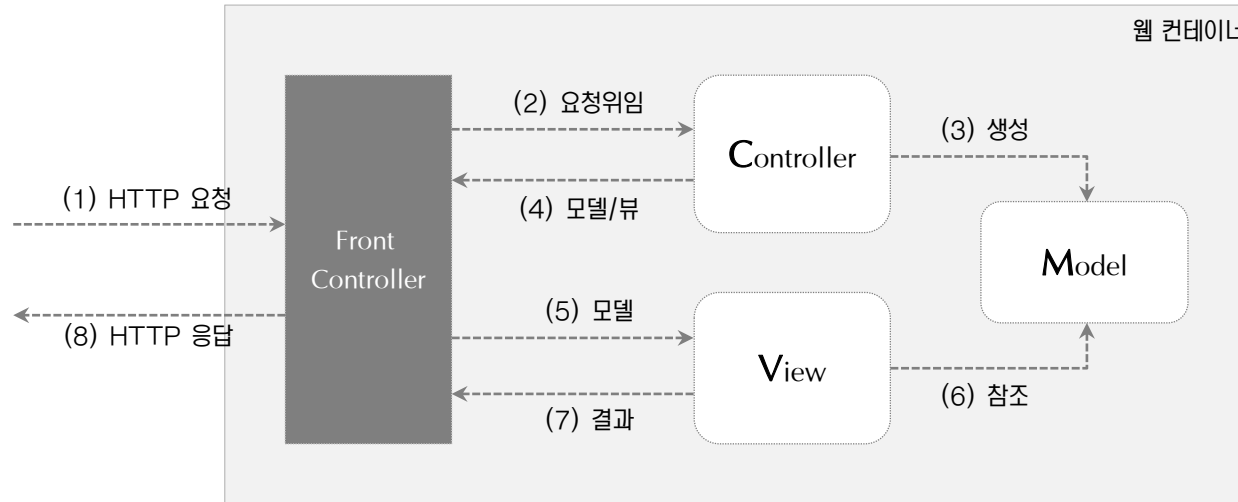
- ✓ MVC(Model-View-Controller) 패턴은 코드를 처리하는 영역을 Model, View, Controller로 분리합니다.
- ✓ 영역별로 분리하면 컴포넌트 간의 결합도가 낮아져, 컴포넌트 변경이 다른 영역 컴포넌트에 영향을 주지 않습니다.
- ✓ 사용자 화면을 비즈니스 로직과 분리하므로, 기능 확장과 변경이 편리합니다.
- ✓ 개발과정이 복잡해 초보자가 이해하고 개발하기에 다소 어려워, 처음엔 개발 진도가 늦어질수 있습니다.



Model	<ul style="list-style-type: none">- 어플리케이션 상태의 캡슐화- 상태 쿼리에 대한 응답- 어플리케이션의 기능 표현- 변경을 뷰에 통지
View	<ul style="list-style-type: none">- 모델을 화면에 시각적으로 표현- 모델에게 업데이트 요청- 사용자의 입력을 컨트롤러에 전달- 컨트롤러가 뷰를 선택하도록 허용
Controller	<ul style="list-style-type: none">- 어플리케이션의 행위 정의- 사용자 액션에 대한 모델 업데이트와 매핑- 응답에 대한 뷰 선택

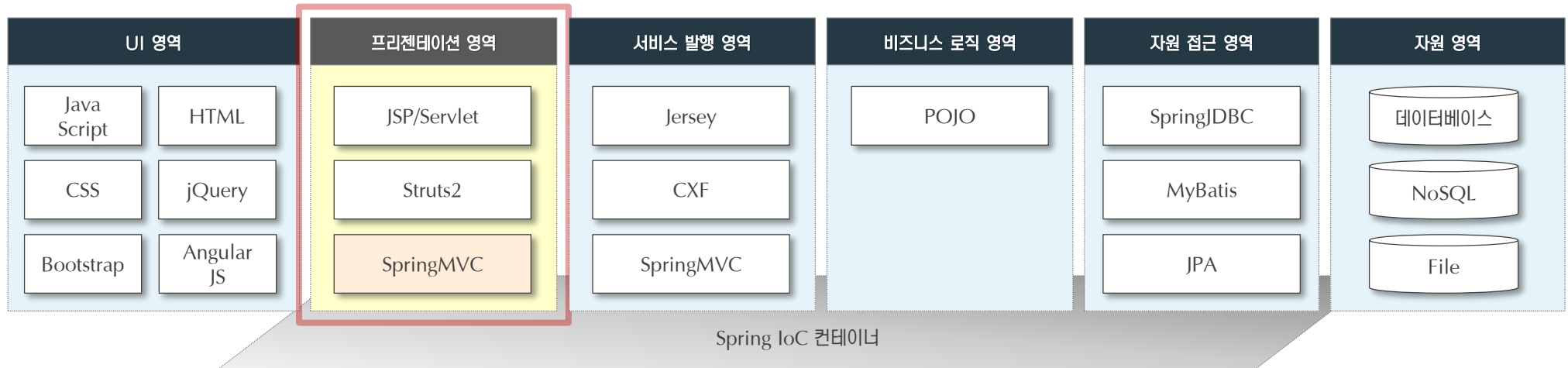
1.1 웹 프레임워크 아키텍처 (2/2) – Front Controller 패턴

- ✓ 웹 MVC 프레임워크는 J2EE 패턴 의 Front Controller 패턴에 기초합니다.
- ✓ Front Controller는 요청을 받아 어느 컴포넌트로 요청을 전송할지 결정한 후, 그 컴포넌트에 처리를 위임합니다.
- ✓ 프리젠테이션 계층의 제일 앞에서 모든 요청을 최초로 수신하여 처리하는 서블릿을 Front Controller라고 합니다.
- ✓ 웹 MVC 프레임워크는 요청을 각 컨트롤러로 분기하는 중앙 서블릿 중심으로 설계되었습니다.



1.2 Spring MVC 개요 (1/2) – 스프링기반 웹 애플리케이션

- ✓ Spring MVC는 스프링에서 프리젠테이션 계층을 담당하는 서블릿 기반 MVC 프레임워크입니다.
- ✓ Spring MVC는 다른 웹 프레임워크에 비해 특정 클래스 상속, 참조 또는 구현 제약사항이 적습니다.
- ✓ Spring은 POJO(Plain Old Java Object)를 지향하므로 복잡한 설정 없이 비즈니스 로직에 집중할 수 있습니다.
- ✓ Spring IoC 컨테이너를 사용하여 웹 프레임워크 연계를 위한 추가 설정 없이 Spring MVC를 사용할 수 있습니다.



1.2 Spring MVC 개요 (2/2) – 웹 프레임워크 트렌드 분석

- ✓ 웹 MVC 프레임워크에는 Spring MVC, Struts2, Struts1, JSF 등이 있습니다.
- ✓ 과거에는 Struts2가 대표적인 웹 프레임워크였으나, 최근에는 Spring MVC에 관심이 높아지고 있습니다.
- ✓ Struts2는 프리젠테이션 계층만 지원하는 프레임워크로써, 애플리케이션을 개발 시 다른 프레임워크와 연동해야 합니다.
- ✓ Spring MVC는 Spring 프레임워크의 기능을 활용할 수 있고, 기능 확장이 쉬워 꾸준한 상향세를 보이고 있습니다.



출처 : <https://www.google.com/trends>

1.3 Spring MVC 특징 (1/2) – 기술 확장에 관대한 웹 프레임워크

- ✓ 프레임워크에서 조합한 기술을 강요하지 않고, 프로젝트에 적합한 구성을 할 수 있습니다.
- ✓ 스프링 MVC에서 제공하는 주요 기능들은 다양한 방법으로 확장할 수 있습니다.
- ✓ 관례를 따르는 기본 설정으로 빠른 개발을 할 수 있습니다. 필요한 경우 설정을 위한 확장포인트를 제공합니다.
- ✓ 스프링 MVC를 이용하면 프로젝트에 적합하고 효율적으로 개발할 수 있는 프레임워크를 새롭게 구성할 수 있습니다.



1.3 Spring MVC 특징 (2/2) – 설정 보다는 관례(CoC)

- ✓ CoC(Convention over Configuration)는 설정 보다는 관례를 따르는 소프트웨어 설계 방식입니다.
- ✓ CoC는 개발자들이 결정할 사항들을 줄여서 단순하고 유연하고 빠른 개발이 가능합니다.
- ✓ XML과 같은 설정이 많으면 개발 복잡도가 증가합니다. 최근 프레임워크들은 어노테이션(@)으로 CoC를 실현합니다.
- ✓ Spring MVC는 공통된 관례들을 제공하고, 원하는 설정만 개별적으로 적용하도록 하여 개발의 효율을 높입니다.



1.4 Spring @MVC

- ✓ Spring 은 2.5버전 부터 어노테이션을 도입하였고, 3.0 버전에서 어노테이션 지원을 강화했습니다.
- ✓ 어노테이션을 중심으로 한 새로운 MVC를 어노테이션 기반 MVC 이라는 의미로 Spring @MVC라고도 합니다.
- ✓ 현재 Spring은 XML로 설정하던 이전 버전의 사용을 권고하지 않으며, 차기 버전에서는 없어질 수도 있습니다.
- ✓ 어노테이션을 추가함으로써 POJO에 가까운 개발이 가능하며, 메소드 단위로 요청을 처리할 수 있습니다.

[이전 방식] AbstractController 클래스를 상속하여 컨트롤러 작성

```
public class CookBookReadController extends AbstractController{  
  
    protected ModelAndView handleRequestInternal(HttpServletRequest req  
                                                , HttpServletResponse res) throws Exception {  
  
        .....  
    }  
}
```

[Spring @MVC] 어노테이션을 적용한 컨트롤러

```
@Controller  
public class CookBookController {  
  
    @RequestMapping("/cookbook")  
    public ModelAndView getAll() {  
  
        .....  
    }  
}
```

[이전 방식] Controller 인터페이스를 구현하여 컨트롤러 작성

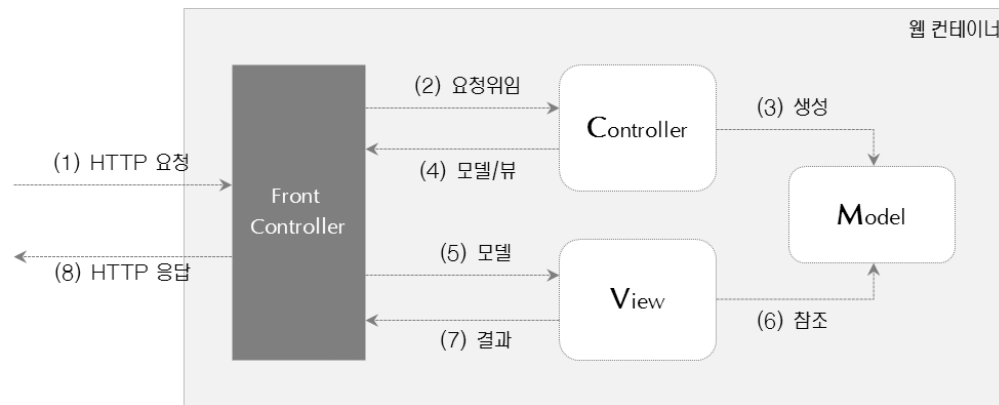
```
public class CookBookReadController implements Controller {  
  
    @Override  
    public ModelAndView handleRequest(HttpServletRequest req  
                                    , HttpServletResponse res) throws Exception {  
  
        .....  
    }  
}
```



@Controller 가 붙은 클래스는 클라이언트 요청 시점에 Front Controller에 의해 호출됩니다. Spring@MVC에서는 어노테이션을 사용하여 인터페이스나 클래스를 상속하지 않고 컨트롤러 클래스를 작성할 수 있습니다. @RequestMapping 은 XML 설정 없이 요청 URL을 매핑하는 어노테이션 입니다.

1.5 요약

- ✓ Spring MVC는 MVC 패턴 기반의 웹 애플리케이션 프레임워크입니다.
- ✓ Spring MVC는 모든 요청을 받아 각 컨트롤러로 요청을 위임하는 Front Controller를 사용합니다.
- ✓ 설정보다는 관례(CoC) 중심 프레임워크로서, 보편적인 기능은 기본으로 제공하고 그 외 설정은 확장 가능합니다.
- ✓ Spring @MVC라고 불리울 만큼 어노테이션을 이용한 편리하고 효율적인 개발을 지원합니다.



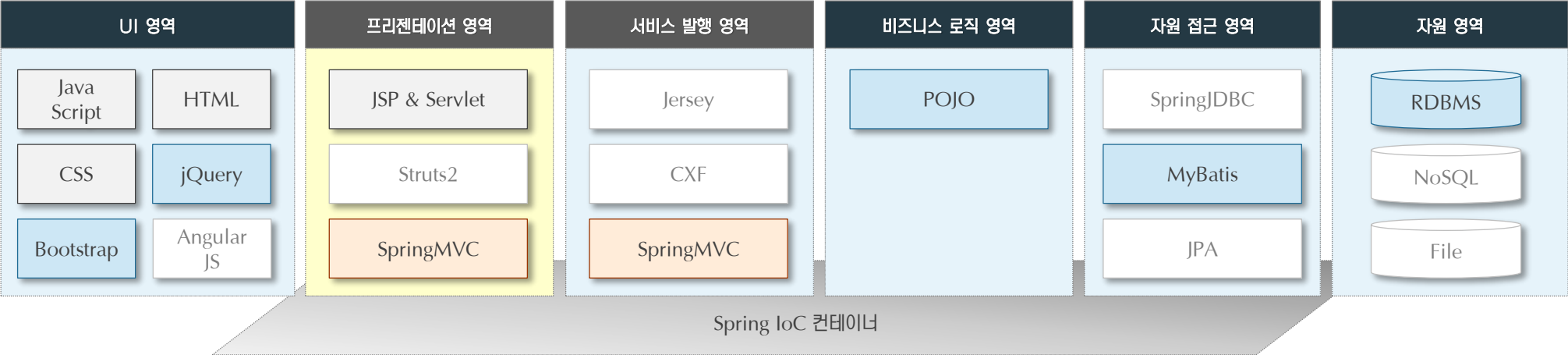


2. MVC 웹 프로그래밍 I

- 2.1 웹 애플리케이션 구조
- 2.2 실습 도메인 설명
- 2.3 실습 프로젝트 준비
- 2.4 MVC 웹 프로그래밍
- 2.5 요약

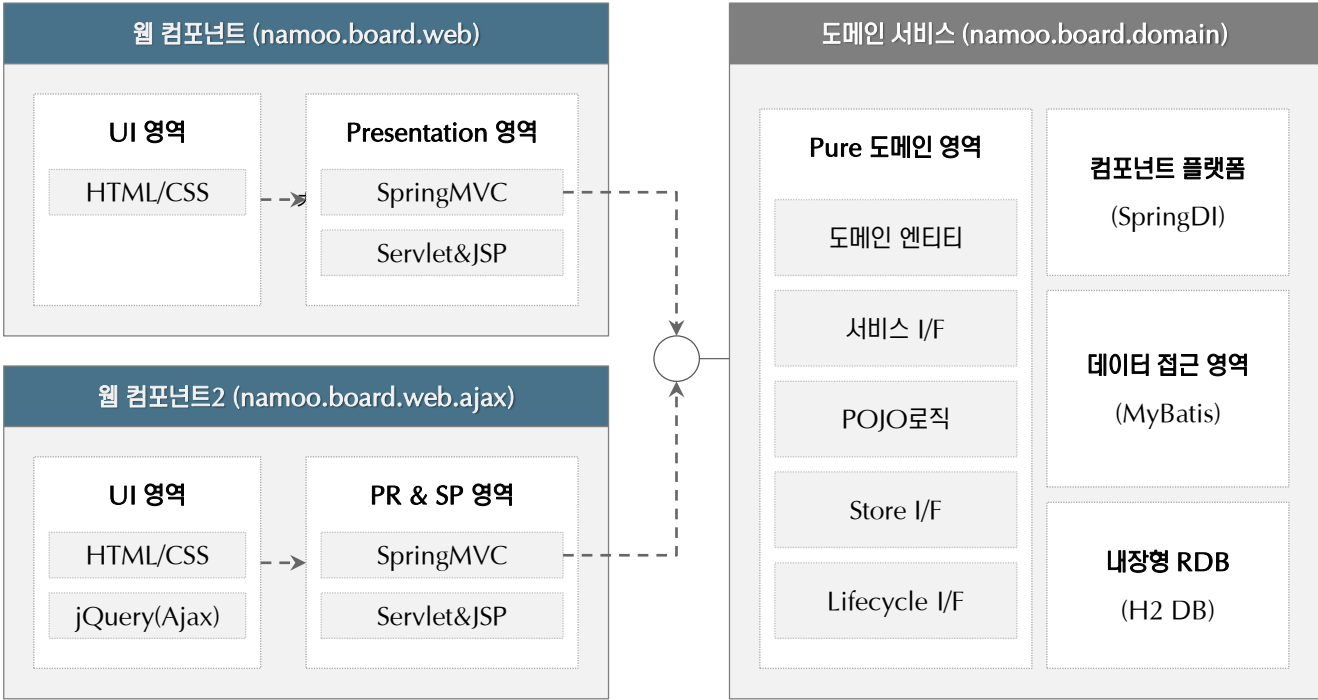
2.1 웹 애플리케이션 구조

- ✓ 실습을 위한 웹 애플리케이션은 실무에서 많이 사용하는 기술을 사용하여 아키텍처를 설계합니다.
- ✓ 프리젠테이션 영역에서는 Java 웹 개발의 기본이 되는 JSP&Servlet과 함께 SpringMVC 프레임워크를 사용합니다.
- ✓ 서비스 발행영역은 SpringMVC의 RESTful 웹 서비스 지원 기능을 사용합니다. 이때, UI는 Ajax 방식으로 구현합니다.
- ✓ 데이터는 관계형 데이터베이스에 저장하고, MyBatis를 데이터 접근 프레임워크로 사용합니다.



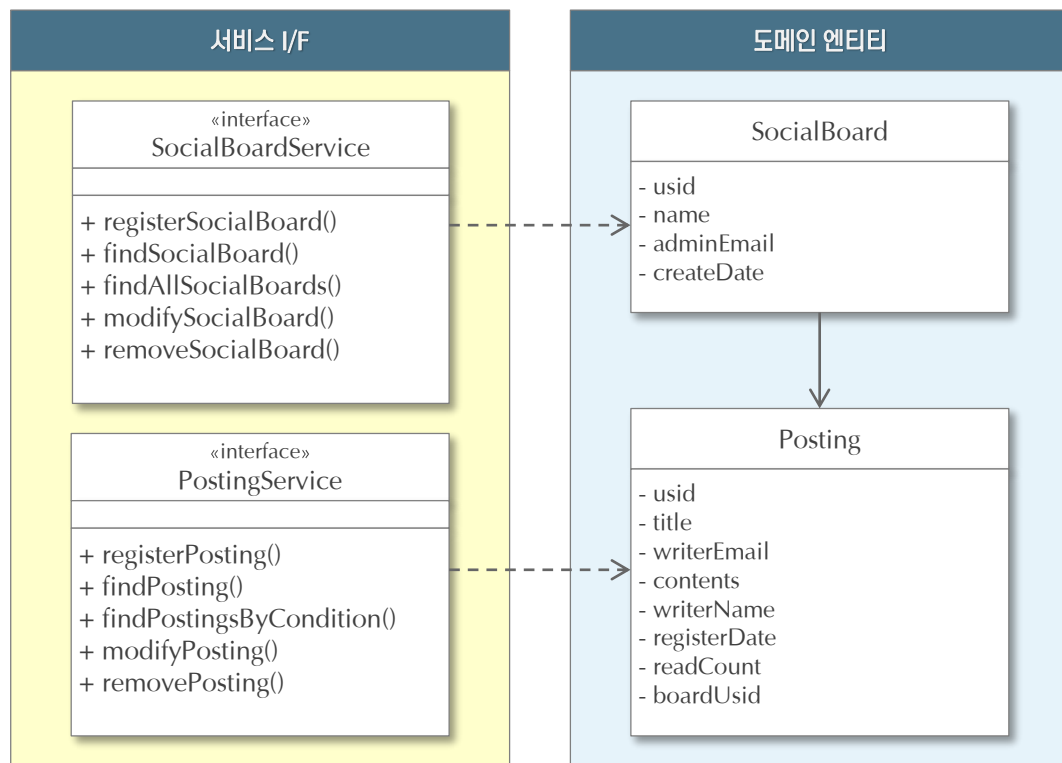
2.2 실습 도메인 설명 (1/3) – 프로젝트 구성

- ✓ namoo.board는 게시판과 게시물을 관리할 수 있는 소셜보드 서비스입니다.
- ✓ namoo.board.domain 프로젝트는 기본으로 제공하며, 게시판 도메인에 대한 서비스를 제공합니다.
- ✓ namoo.board.web 프로젝트에서는 SpringMVC 를 사용한 프리젠테이션 영역구현을 실습합니다. (2장, 4장)
- ✓ namoo.board.web.ajax 프로젝트에서는 RESTful 웹 서비스를 구현하여 Ajax 웹 프로그래밍을 실습합니다. (5장)



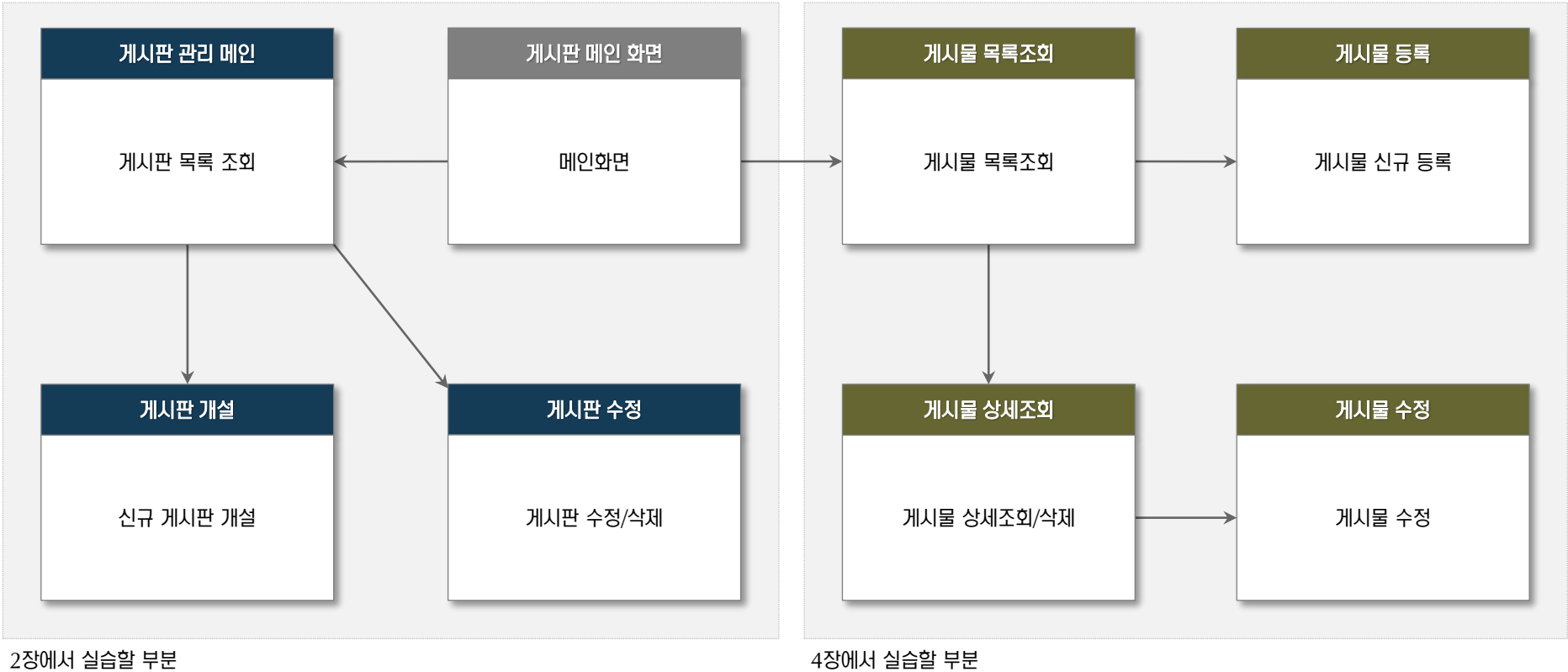
2.2 실습 도메인 설명 (2/3) – 도메인 모델

- ✓ namoo.board.domain 프로젝트에서 Pure 도메인 영역을 이해하는 것이 중요합니다.
- ✓ Pure 도메인 영역에는 도메인의 핵심개념인 엔티티 객체와 업무 로직을 나타내는 서비스 객체를 설계합니다.
- ✓ 도메인 엔티티 객체 모델링에서는 소셜보드의 핵심개념인 게시판과 게시물 객체를 식별하였습니다.
- ✓ 소셜보드 도메인의 서비스에는 게시판을 관리하는 서비스와 게시물을 관리하는 게시물 서비스가 있습니다.



2.2 실습 도메인 설명 (3/3) – 화면 Navigation

- ✓ namoo.board의 UI는 게시판을 관리하는 화면과 게시물을 등록 및 조회하는 화면으로 구성합니다.
- ✓ 게시판 관리 화면에는 게시판 관리 메인, 게시판 개설, 게시판 수정/삭제 화면이 있습니다.
- ✓ 게시물 관리 화면에는 게시물 목록조회, 게시물 상세조회, 게시물 등록, 게시물 수정 화면이 있습니다.

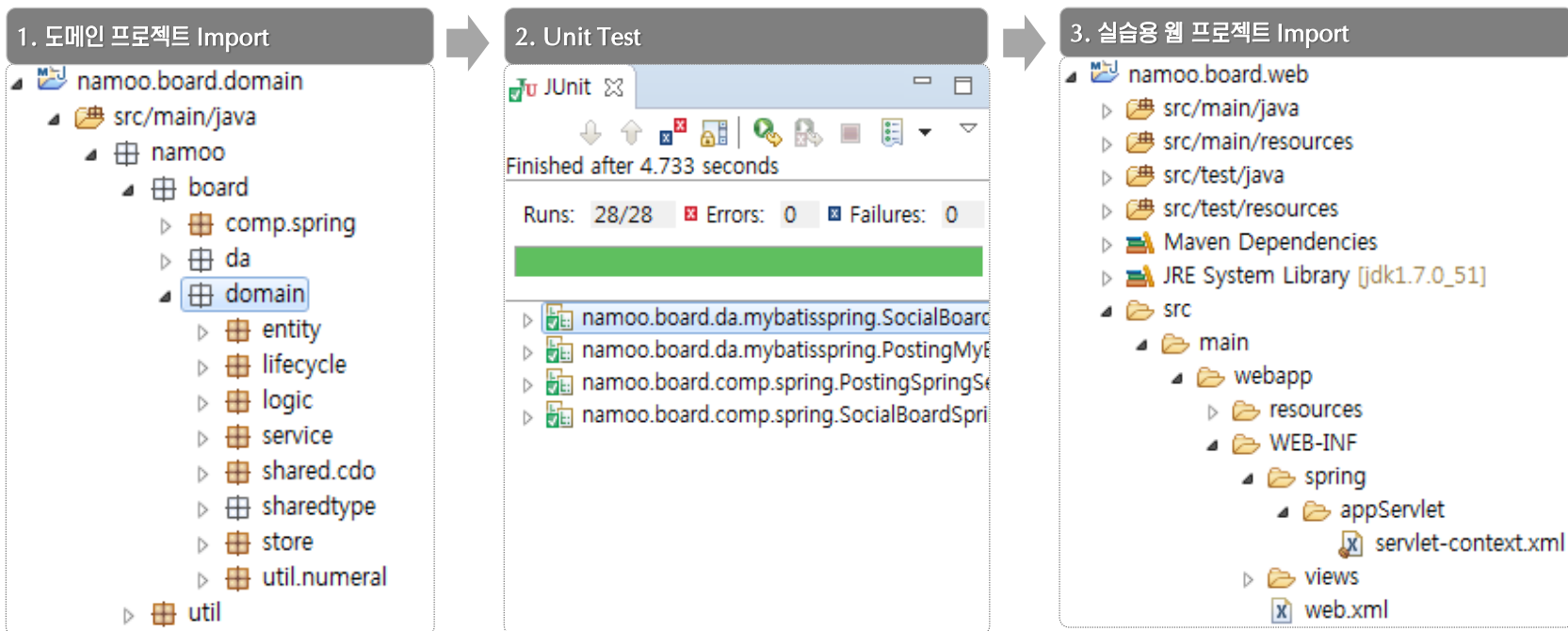


2.3 실습 프로젝트 준비 – namoo.board.web

프로
젝트

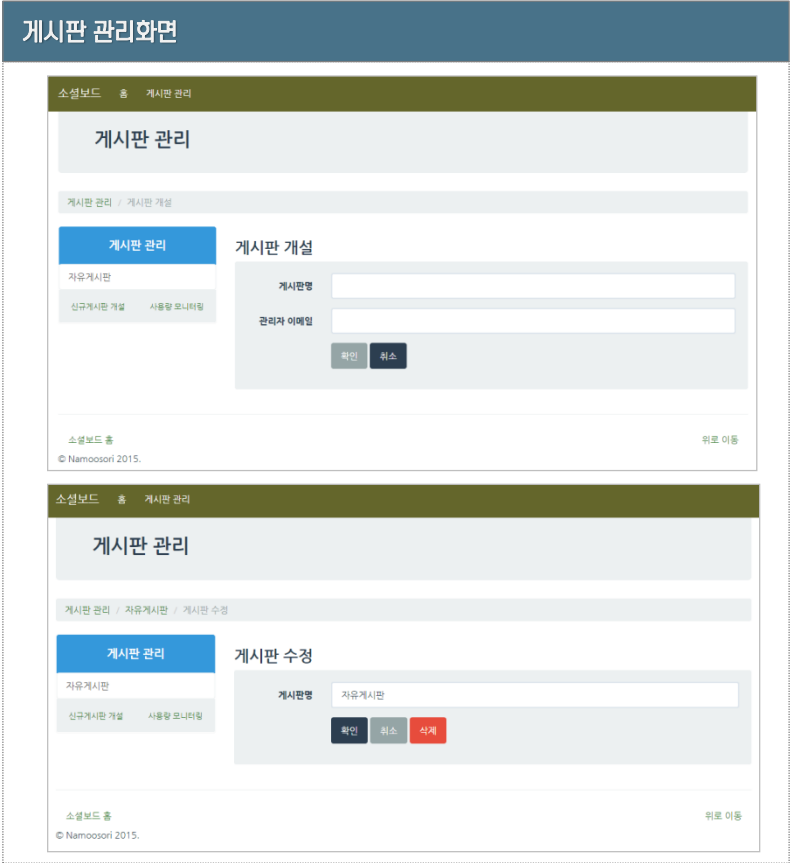
namoo.board.domain.zip
namoo.board.web-step0.zip

- ✓ namoo.board.domain.zip 파일을 이클립스에 Import 합니다.
- ✓ namoo.board.domain 프로젝트에는 서비스가 이미 구현되어 있습니다. 단위테스트로 동작여부를 확인합니다.
- ✓ namoo.board.web-step0.zip 파일을 이클립스 프로젝트로 Import 합니다.
- ✓ namoo.board.web 프로젝트는 실습대상 프로젝트 입니다.



2.4 MVC 웹 프로그래밍 (1/3) – 게시판 관리 화면

- ✓ 게시판 관리화면에는 게시판을 개설하는 화면과 게시판을 수정하고 삭제하는 화면이 있습니다.
- ✓ 화면을 조회하거나 요청을 처리하는 단위 별로 요청URL과 HTTP 메소드를 정의합니다.
- ✓ 각각의 요청을 처리하는 요청 메소드를 작성합니다. 관련된 요청 메소드를 모아서 하나의 컨트롤러 클래스를 만듭니다.
- ✓ 게시판 관리 화면에서는 6개의 요청처리 메소드를 식별하여 컨트롤러 클래스를 구성합니다.



게시판 관리화면에 필요한 기능들	
<div>게시판 관리 메인 화면 조회</div> <div>URL : {ctx}/board/list</div> <div>기능 : 게시판 목록을 보여줍니다.</div> <div>뷰 : /board/list</div>	<div>게시판 개설 화면 조회</div> <div>URL : {ctx}/board/create</div> <div>기능 : 게시판 개설화면을 보여줍니다.</div> <div>뷰 : /board/create</div>
<div>게시판 개설 처리</div> <div>URL : {ctx}/board/create</div> <div>기능 : 게시판명과 관리자 이메일을 입력받아 새로운 게시판을 개설합니다.</div> <div>뷰 : /board/list</div>	<div>게시판 수정 화면 조회</div> <div>URL : {ctx}/board/update</div> <div>기능 : 게시판 수정화면을 보여줍니다.</div> <div>뷰 : /board/update</div>
<div>게시판 수정 처리</div> <div>URL : {ctx}/board/update</div> <div>기능 : 게시판명을 입력받아 게시판명을 수정합니다.</div> <div>뷰 : /board/list</div>	<div>게시판 수정 처리</div> <div>URL : {ctx}/board/update</div> <div>기능 : 게시판명을 입력받아 게시판명을 수정합니다.</div> <div>뷰 : /board/list</div>

2.4 MVC 웹 프로그래밍 (2/3) – 게시판 목록 조회 및 개설

- ✓ BoardController 클래스를 생성하고, @Controller 어노테이션을 붙입니다.
- ✓ 게시판 관련서비스를 제공하는 SocialBoardService를 필드로 정의하고 @Autowired 어노테이션을 붙입니다.
- ✓ 요청처리 메소드를 작성합니다. @RequestMapping 어노테이션에 URL 매핑과 HTTP 방식을 설정합니다.
- ✓ 요청내용을 처리하여 모델에 담은 후, 처리결과를 보여줄 뷰 이름을 반환합니다.

```
@Controller
public class BoardController {

    @Autowired
    private SocialBoardService socialBoardService;

요청처리 메소드 구현



}
```

```
@RequestMapping(value = "/board/list",
                method = RequestMethod.GET)
public String main(HttpServletRequest req) {
    //
    List<SocialBoard> socialBoards =
        socialBoardService.findAllSocialBoards();
    req.setAttribute("boardList", socialBoards);

    return "/board/list";
}
```

```
@RequestMapping(value = "/board/create",
                method = RequestMethod.GET)
public String create(Model model) {
    List<SocialBoard> socialBoards =
        socialBoardService.findAllSocialBoards();

    model.addAttribute("boardList", socialBoards);

    return "board/create";
}
```

```
@RequestMapping(value = "/board/create",
                method = RequestMethod.POST)
public ModelAndView create(
    @RequestParam("boardName") String boardName,
    @RequestParam("adminEmail") String adminEmail ) {

    socialBoardService.registerSocialBoard(
        new SocialBoardCdo(boardName, adminEmail));
    String message = "게시판이 개설되었습니다.";
    String linkURL = "board/list";
    return MessagePage.information(message, linkURL);
}
```

2.4 MVC 웹 프로그래밍 (3/3) – 게시판 수정 및 삭제

- ✓ 이번에는 게시판 수정화면과 수정/삭제를 처리하는 기능을 구현해 봅니다.
- ✓ 이제, 게시판 관리에 필요한 모든 기능을 구현하였습니다. 서버를 띄워 결과를 확인합니다.
- ✓ 게시판 목록화면에서 [신규게시판 개설]을 클릭하여 게시판을 개설합니다.
- ✓ 개설된 게시판을 선택하여 게시판명을 수정하거나 게시판을 삭제합니다.

```
@RequestMapping(value = "/board/update", method = RequestMethod.GET)
public String update(@RequestParam("boardUsid") String boardUsid,
    HttpServletRequest req) {
    //
    SocialBoard socialBoard =
        socialBoardService.findSocialBoard(boardUsid);
    List<SocialBoard> socialBoards =
        socialBoardService.findAllSocialBoards();

    req.setAttribute("boardList", socialBoards);
    req.setAttribute("socialBoard", socialBoard);
    return "board/update";
}

@RequestMapping(value = "/board/update", method = RequestMethod.POST)
public ModelAndView update(
    @RequestParam("boardUsid") String boardUsid,
    @RequestParam("boardName") String boardName) {
    //
    NameValueCollection nameValues = NameValueCollection.getInstance();
    nameValues.add("name", boardName);
    socialBoardService.modifySocialBoard(boardUsid, boardName);

    String message = "게시판이 수정되었습니다.";
    String linkURL = "board/list";
    return MessagePage.information(message, linkURL);
}
```

```
@RequestMapping(value = "/board/delete")
public ModelAndView delete(@RequestParam("boardUsid") String boardUsid)
{
    //
    socialBoardService.removeSocialBoard(boardUsid);

    String message = "게시판이 삭제되었습니다.";
    String linkURL = "board/list";
    return MessagePage.information(message, linkURL);
}
```

실행결과

2.5 요약

- ✓ Spring MVC로 간단한 실습을 진행합니다.
- ✓ 제시된 UI와 비즈니스로직을 참고하여 컨트롤러를 작성합니다.
- ✓ SpringMVC는 클래스 단위 뿐 아니라 메소드 단위로도 요청을 매핑할 수 있습니다.
- ✓ 하나의 컨트롤러에 게시판의 생성, 조회, 수정, 삭제 처리 메소드를 구현합니다.





3. Spring MVC 이해

3.1 DispatcherServlet

3.2 컨트롤러

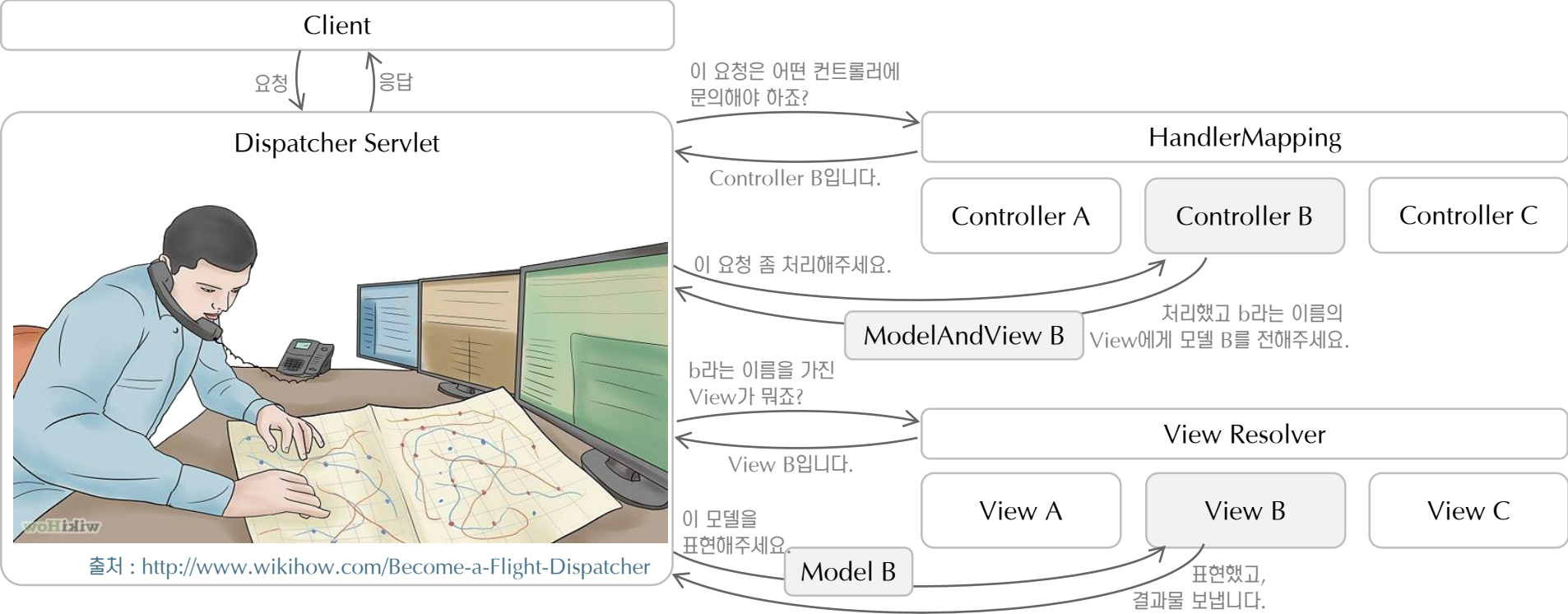
3.3 뷰

3.4 모델

3.5 요약

3.1 DispatcherServlet (1/2) – 요청처리 절차

- ✓ DispatcherServlet은 Spring MVC 의 핵심으로써, 프론트 컨트롤러 역할을 담당합니다.
- ✓ 클라이언트의 요청을 받아 응답하는 과정 속에서 담당할 대상을 선택하고 역할을 분배하는 등의 작업을 수행합니다.
- ✓ DispatcherServlet이 요청을 처리하는 과정을 이해하면 Spring MVC를 잘 활용할 수 있습니다.
- ✓ 웹 애플리케이션 설정파일(web.xml)에 서블릿 요청을 DispatcherServlet 클래스가 처리하도록 매핑 합니다.



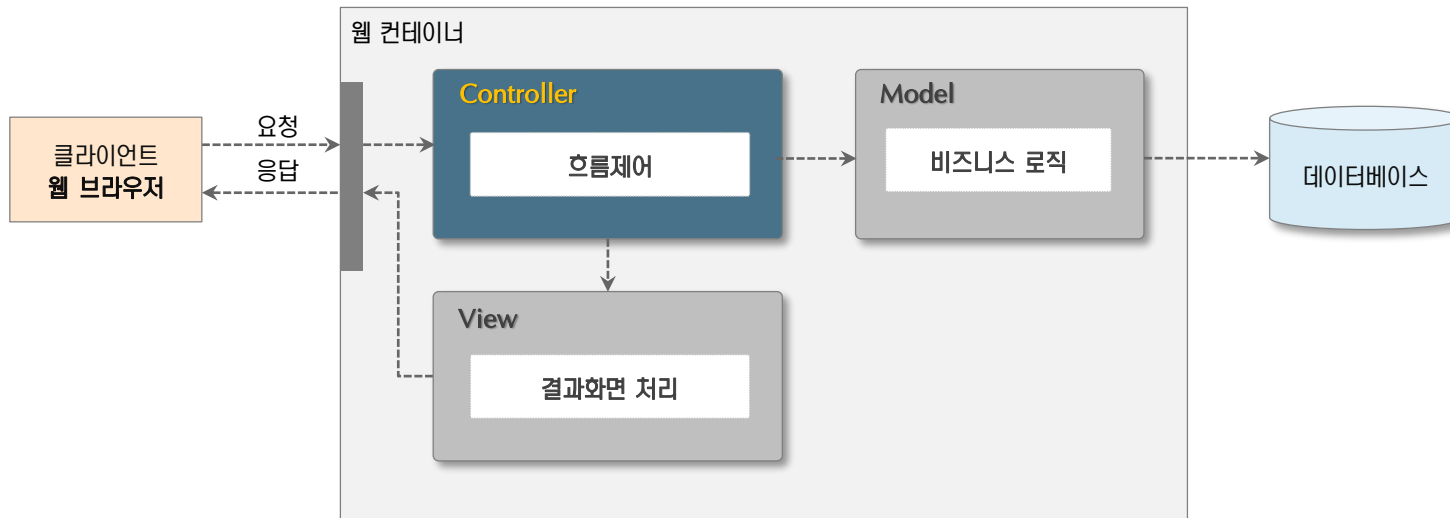
3.1 DispatcherServlet (2/2) – MVC 클래스

- ✓ 개발자는 Controller만 구현하고, 그 외의 객체는 Spring MVC가 제공하는 클래스를 사용합니다.
- ✓ DispatcherServlet은 요청에서부터 응답까지의 전체 라이프사이클을 관리합니다.
- ✓ HandlerMapping이 요청을 처리할 컨트롤러를 결정하는 기준에는 URL, 클래스명, 어노테이션 등이 있습니다.
- ✓ ModelAndView는 요청처리 결과 데이터와 화면에 표시할 View이름을 포함합니다.

클래스 명	설명
DispatcherServlet	단일 프론트 컨트롤러로 모든 HTTP 요청을 수신하여 그 밖의 오브젝트 사이의 흐름을 제어합니다.
HandlerMapping	클라이언트가 요청한 URL을 바탕으로 어느 컨트롤러를 실행할 지 결정합니다. URL, 컨트롤러 클래스명, 어노테이션등을 기준으로 결정합니다.
Controller	클라이언트 요청에 맞는 프리젠테이션 층의 애플리케이션 처리를 실행합니다. 요청처리 결과 데이터를 ModelAndView에 반영합니다.
ModelAndView	Model은 컨트롤러에서 뷰에 전달할 데이터를 저장하는 객체입니다. ModelAndView는 실제 View의 JSP정보를 갖고 있지 않으며, ViewResolver가 논리적 이름을 실제 JSP이름으로 변환합니다.
ViewResolver	View 이름을 바탕으로 View 객체를 결정합니다.
View	View는 화면에 표시하도록 요청합니다.

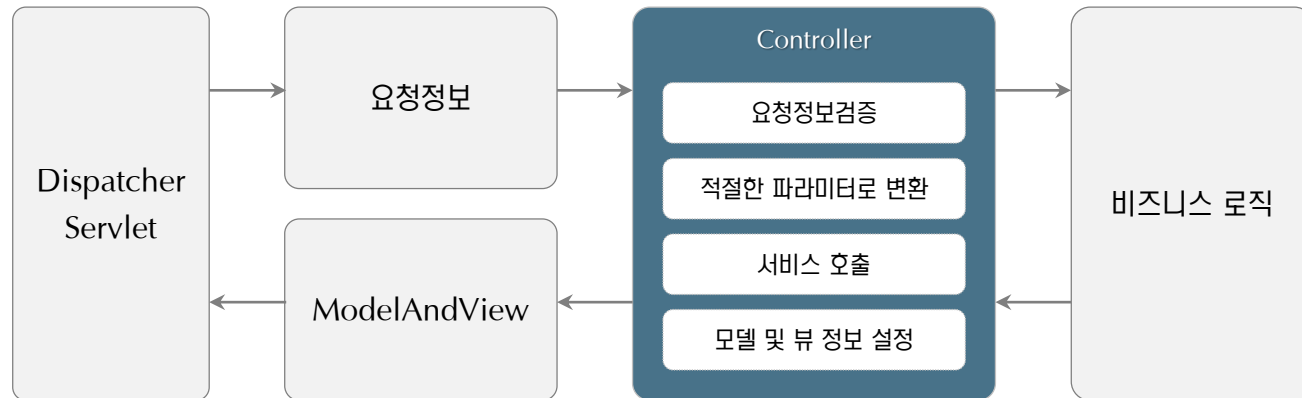
3.2 컨트롤러 (1/11) – 개요 (1/2)

- ✓ 컨트롤러는 MVC (Model, View, Controller) 중 가장 많은 작업을 처리합니다.
- ✓ 컨트롤러는 클라이언트의 모든 요청은 받아서 처리합니다.
- ✓ 컨트롤러는 요청 내용을 처리하는 서비스를 호출하여 비즈니스 로직을 처리합니다.
- ✓ 컨트롤러는 처리결과를 View에게 전달하여 결과화면을 생성하게 합니다.



3.2 컨트롤러 (2/11) – 개요 (2/2)

- ✓ Spring MVC 컨트롤러는 DispatcherServlet 에게 전달받은 요청정보의 정합성을 검증합니다.
- ✓ 서비스에게 비즈니스 로직 처리를 위임합니다. 이를 위해 적절한 파라미터로 변환하여 서비스에게 전달합니다.
- ✓ 서비스로 부터 처리결과를 받으면 어떤 뷰를 보여줘야 할지 결정하고, 처리 결과를 뷰에게 전달할 형태로 생성합니다.
- ✓ 모델과 뷰를 생성하여 DispatcherServlet에게 전달합니다.



3.2 컨트롤러 (3/11) – @Controller

- ✓ 클래스에 @Controller 어노테이션을 붙이면 빈 등록 설정 없이도 컨트롤러를 빈으로 등록할 수 있습니다.
- ✓ 단, 어노테이션을 사용하기 위해서는 설정파일에 <component-scan> 요소를 추가해야 합니다.
- ✓ <component-scan> 요소를 추가하고 어노테이션을 스캔할 범위를 패키지로 지정합니다.
- ✓ 스캔할 범위를 세밀하게 설정하려면, 하위 엘리먼트로 <include-filter> 를 추가합니다.

```
@Controller
public class HomeController {

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String home(Locale locale, Model model) {
        Date date = new Date();
        DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG, DateFormat.LONG, locale);

        String formattedDate = dateFormat.format(date);

        model.addAttribute("serverTime", formattedDate );

        return "home";
    }
}
```

컨트롤러 클래스

```
<context:component-scan base-package="com.namoo.web">
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller" />
</context:component-scan>
```

스프링 설정파일

3.2 컨트롤러 (4/11) – 핸들러 매핑 (1/6)

- ✓ 핸들러 매핑은 HTTP 요청정보를 처리하는 컨트롤러(핸들러)를 찾아주는 역할을 합니다.
- ✓ 핸들러 매핑을 여러 개 등록하여 사용하는 경우, order를 이용해 우선 순위를 정할 수 있습니다.
- ✓ 핸들러 매핑 빈의 defaultHandler 프로퍼티에 URL을 매핑 하지 못할 때 사용할 디폴트 핸들러를 지정할 수 있습니다.
- ✓ 스프링에서는 다섯 가지의 핸들러 매핑 전략 클래스를 제공합니다.

```
<bean id="homeController" class="com.namoo.web.HomeController" />

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="defaultHandler" ref="homeController"/>
</bean>
```

스프링 설정파일

3.2 컨트롤러 (5/11) – 핸들러 매핑 (2/6)

- ✓ BeanNameUrlHandlerMapping 은 핸들러 매핑을 등록하지 않으면 사용되는 디폴트 핸들러 매핑 전략입니다.
- ✓ <bean> 요소의 name과 컨트롤러 클래스를 연결합니다.
- ✓ 설정은 간편하지만, 컨트롤러의 개수만큼 빈을 등록해줘야 하는 번거로움이 있습니다.
- ✓ <bean> 요소의 name 속성은 매핑할 URL이며, class 속성은 요청을 처리할 컨트롤러 클래스입니다.

```
<bean id="handlerMapping" class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />
```

```
<bean name="/home" class="com.namoo.web.HomeController" />
```

URL로 매핑할 이름

요청을 처리할 컨트롤러 클래스

3.2 컨트롤러 (6/11) – 핸들러 매핑 (3/6)

- ✓ `ControllerBeanNameHandlerMapping` 은 빈의 id와 지정한 컨트롤러 클래스를 연결하는 핸들러 매핑 전략입니다.
- ✓ `urlPrefix` 프로퍼티에 URL에서 공통적으로 나타나는 prefix를 설정할 수 있습니다.
- ✓ `@Component` 어노테이션에 bean id 를 지정한 경우에도 핸들러로 매핑됩니다.
- ✓ 이 핸들러 매핑 전략은 빈으로 등록해야만 사용할 수 있으며, 이를 등록 시 디폴트 핸들러는 적용되지 않습니다.

```
<bean id="handlerMapping" class="org.springframework.web.servlet.mvc.support.ControllerBeanNameHandlerMapping" >
    <property name="urlPrefix" value="/mvc/" />
</bean>
```

```
<bean id="/hello" class="com.namoo.web.HomeController" />
```

URL로 매핑할 이름

요청을 처리할 컨트롤러 클래스



`urlPrefix`를 `/mvc/`로 지정하였으므로, `HomeController` 는 `/mvc/hello` 요청에 대해 매핑됩니다.

```
@Component("hello")
public class HomeController implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
                                     HttpServletResponse response) throws Exception {

        return null;
    }
}
```



스테레오타입 어노테이션에 값을 설정하여 매핑할 URL을 지정할 수 있습니다.

3.2 컨트롤러 (7/11) – 핸들러 매핑 (4/6)

- ✓ `ControllerClassNameHandlerMapping` 은 컨트롤러 클래스 이름을 URL과 연결하는 핸들러 매핑 전략입니다.
- ✓ 만약 클래스 이름이 `Controller`로 끝난다면 `Controller`를 제외하고 매핑합니다.
- ✓ 예를 들어, `HomeController`가 있는 경우, `/home` 으로 URL을 매핑합니다.
- ✓ 디폴트 핸들러 매핑이 아니므로 빈으로 등록해야만 사용할 수 있습니다.

```
<bean id="handlerMapping"  
      class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" />
```

```
public class HomeController implements Controller {  
    /home URL 매핑  
  
    @Override  
    public ModelAndView handleRequest(HttpServletRequest request,  
                                     HttpServletResponse response) throws Exception {  
        return null;  
    }  
}
```

3.2 컨트롤러 (8/11) – 핸들러 매핑 (5/6)

- ✓ SimpleUrlHandlerMapping는 URL과 컨트롤러 매핑정보를 한 곳에 모아 놓을 수 있는 핸들러 매핑 전략입니다.
- ✓ 핸들러 매핑 bean property 내의 URL과 컨트롤러 매핑정보를 연결합니다.
- ✓ 모든 URL 매핑정보가 모여있다는 장점이 있는 반면, 컨트롤러 빈 이름을 모두 나열해야 합니다.

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/board">boardController</prop>
      <prop key="/posting/*">postingController</prop>
    </props>
  </property>
</bean>

<bean id="boardController" class="com.namoo.web.BoardController"/>
<bean id="postingController" class="com.namoo.web.PostingController"/>
```

3.2 컨트롤러 (9/11) – 핸들러 매핑 (6/6)

- ✓ RequestMappingHandlerMapping은 어노테이션에 적용되는 디폴트 핸들러 매핑 전략입니다.
- ✓ @RequestMapping 어노테이션으로 URL을 매핑하는 최근 가장 보편적으로 사용하는 매핑 방법입니다.
- ✓ URL이 설정파일이 아닌 클래스 파일에 있다는 점에서 편리하며, URL 뿐 아니라 여러 요청정보를 활용할 수 있습니다.
- ✓ 메소드에 해당 어노테이션을 붙이면 메소드 단위의 매핑을 지원하므로 컨트롤러의 개수를 줄일 수 있습니다.

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

    @RequestMapping("list")
    public String list(Model model) {
        return "/product/list";
    }

    @RequestMapping("regist")
    public String regist(Model model) {
        return "/product/regist";
    }
}
```

3.2 컨트롤러 (10/11) – 요청 매핑 (1/2)

- ✓ RequestMappingHandlerMapping은 @RequestMapping 정보로 컨트롤러를 매핑합니다.
- ✓ 부모 컨트롤러 클래스에 @RequestMapping를 추가하면 하위 컨트롤러 클래스에도 적용됩니다.
- ✓ 단, 하위 클래스에서 @RequestMapping을 재정의하면 상위 클래스의 @RequestMapping은 무시됩니다.
- ✓ @RequestMapping을 인터페이스에 작성한 경우도 구현 클래스에 동일하게 적용됩니다.

```
public class ParentController {  
    @RequestMapping("find")  
    public String find() {  
        return "find";  
    }  
}
```

```
@Controller  
public class ChildController extends ParentController {  
    @Override  
    public String find() {  
        System.out.println("child find");  
        return "find";  
    }  
}
```



ChildController 는 상위 클래스인 ParentController에 정의된 @RequestMapping를 그대로 상속받습니다. 결국 /find URL을 입력했을 때, "child find " 문구가 출력됩니다.

3.2 컨트롤러 (11/11) – 요청 매핑 (2/2)

- ✓ @RequestMapping을 이용하면 URL 뿐 아니라 여러 요청정보를 활용할 수 있습니다.
- ✓ value 속성은 매핑할 URL을 설정합니다.
- ✓ method 속성은 요청을 매핑할 HTTP Method를 설정합니다. (GET, POST, PUT, DELETE 등)
- ✓ params 속성은 요청 파라미터를 설정합니다. 파라미터 값이 지정한 값과 일치할 때만 요청을 매핑합니다.

```
@RequestMapping(value="/boards/{boardId}", method=RequestMethod.GET, params="admin=true")
public String findBoard(@PathVariable("boardId") String boardId, Model model){

    .....

    return "/board/read";
}
```



/boards/{boardId} 로 요청된 GET 방식의 요청 중 admin 파라미터 값이 true인 요청만 매핑합니다.

3.3 뷰 (1/5) – 개요

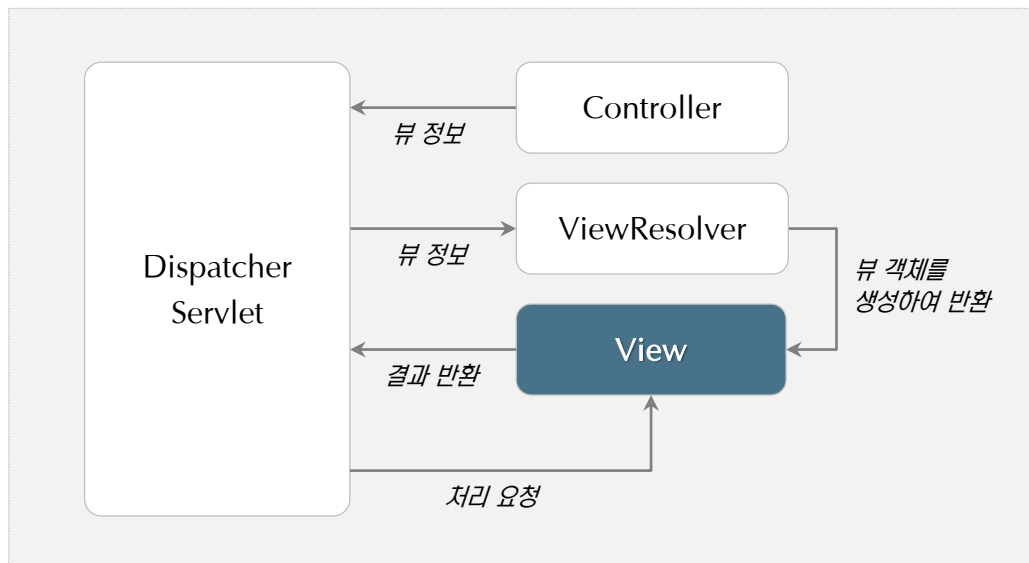
- ✓ MVC에서 뷰(View)는 모델을 전달 받아 모델의 정보를 다양한 형식으로 표현하는 기능을 담당합니다.
- ✓ 일반적으로 뷰는 HTML로 생성되어 브라우저가 결과를 표현하지만, 엑셀/PDF/XML 등의 콘텐츠로 생성되기도 합니다.
- ✓ 스프링에서 제공하는 뷰를 사용하여 여러 콘텐츠를 작성할 수도 있습니다.
- ✓ 뷰를 직접 사용하는 대신, 메시지 컨버터를 사용하면 XML, JSON 등을 생성할 수 있습니다.

```
public class Posting {  
  
    private String id;  
    private String title;  
    private String contents;  
  
    public Posting () {}  
  
    public Posting(String id, String title, String contents) {  
        this.id = id;  
        this.title = title;  
        this.contents = contents;  
    }  
}
```



3.3 뷰 (2/5) – 뷰와 컨트롤러

- ✓ MVC에서는 뷰와 컨트롤러의 연결 방식에 대한 이해가 필요합니다.
- ✓ 컨트롤러가 처리 후 돌아갈 뷰를 지정해주었다면 DispatcherServlet은 해당 뷰에 처리를 요청합니다.
- ✓ 명확히 뷰 객체를 지정하지 않고, 뷰 이름만 알려주어도 ViewResolver를 통해 뷰 객체를 생성할 수 있습니다.
- ✓ 어떠한 뷰 정보도 알려주지 않을 때, DefaultRequestToViewNameTranslator를 통해 뷰 이름을 설정할 수 있습니다.



3.3 뷰 (3/5) – 뷰 지정방법

- ✓ 뷰를 지정하는 방법에는 컨트롤러가 명시적으로 리턴하는 방법과, 설정에 의해 자동으로 지정하는 2가지 방법이 있습니다.
- ✓ 첫 번째는 컨트롤러가 ModelAndView나 View 객체에 뷰 이름을 설정하거나, 문자열로 뷰 이름을 반환하는 방법입니다.
- ✓ 두 번째는 컨트롤러가 Model, Map, void를 리턴하는 경우, 설정에 의해 자동으로 뷰를 지정하는 방법입니다.
- ✓ 뷰가 지정되지 않으면 RequestToViewNameTranslator는 요청 URI로 부터 뷰 이름을 지정합니다.

[명시적인 방법] 뷰 이름을 담은 ModelAndView 객체를 리턴함

```
@Controller
public class BoardController {

    @RequestMapping("/board")
    public ModelAndView list() {
        return new ModelAndView("board");
    }
}
```

[명시적인 방법] 뷰 이름을 문자열로 리턴함

```
@Controller
public class BoardController {

    @RequestMapping("/board")
    public String list() {
        return "board";
    }
}
```

[자동지정 방법] 요청 URI 로 부터 뷰 이름이 지정됨

```
@Controller
public class BoardController {

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



자동으로 뷰 이름이 지정되는 경우, 요청한 URI 에서 맨 앞의 / 기호와 끝에 있는 확장자를 제외한 이름이 뷰의 이름이 됩니다. 따라서 이 예제에서는 board가 뷰 이름이 됩니다.

3.3 뷰 (4/5) – ViewResolver

- ✓ 컨트롤러가 뷰 이름을 반환하면 ViewResolver를 통해 실제 사용할 뷰를 결정합니다.
- ✓ 뷰 이름은 실제 존재하는 뷰가 아니므로, ViewResolver를 통해 논리적인 뷰(뷰 이름)를 실질적인 뷰로 바꿔줍니다.
- ✓ ViewResolver는 기본 제공 ViewResolver 구현체를 등록하여 사용하거나 직접 확장하여 구현할 수 있습니다.
- ✓ 특정 ViewResolver를 빈으로 등록하지 않으면, InternalResourceViewResolver 가 자동으로 등록됩니다.

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

스프링 설정파일



InternalResourceViewResolver 에 prefix/suffix 등의 속성을 지정하고 싶다면 빈으로 등록해야 합니다.
STS 플러그인을 통해 프로젝트를 생성한 경우, 이러한 설정이 기본으로 제공되므로 편리합니다.

3.3 뷰 (5/5) – 리다이렉트 뷰

- ✓ 리다이렉트뷰는 실제 뷰를 생성하지 않고, URL만 생성하여 다른 페이지로 리다이렉트 합니다.
- ✓ 주로 컨트롤러에서 기능을 수행 후 기존에 존재하는 특정 요청으로 연결하고 싶을 때 사용합니다.
- ✓ 예를 들어 저장 후 목록조회로 연결할 때 리다이렉트를 사용할 수 있습니다.
- ✓ 리다이렉트 뷰는 RedirectView 객체나, 접두어 "redirect:"를 포함한 문자열을 리턴하여 지정합니다.

```
// 레시피를 등록하고 전체 목록 조회화면으로 리다이렉트
@RequestMapping(value = "/recipe", method = RequestMethod.POST)
public String addRecipe(Recipe recipe) {

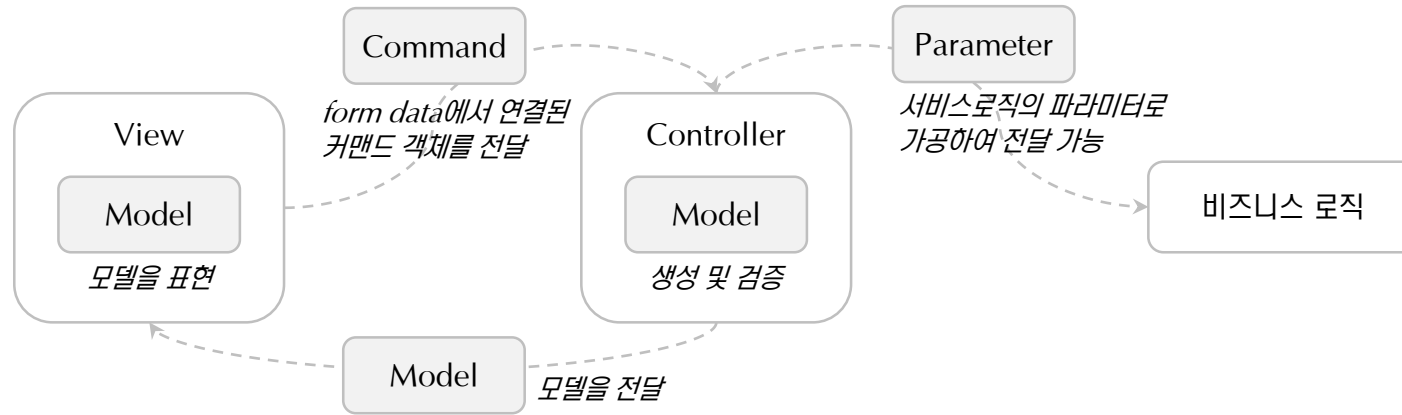
    chef.createRecipe(recipe);
    return "redirect:/";
}
```



RedirectView 오브젝트를 직접 사용하지 않아도, 문자열에 리다이렉트의 접두어를 포함시키면 ViewResolver에서 처리합니다. UriBasedViewResolver에는 "redirect:"에 대한 처리가 구현되어 있습니다. 기본으로 제공하는 InternalResourceViewResolver는 UriBasedViewResolver 클래스를 상속받기 때문에 특별한 설정을 하지 않아도 사용할 수 있습니다.

3.4 모델 (1/4) – SpringMVC 의 모델

- ✓ 컨트롤러는 뷰에서 전달된 데이터를 커맨드 객체로 전달받아 처리할 수 있습니다.
- ✓ 컨트롤러에서 받거나 생성된 모델은 검증과정을 거친 후 비즈니스 로직 메소드에 전달하는 파라미터가 될 수 있습니다.
- ✓ 컨트롤러가 모델을 리턴하면 DispatcherServlet는 모델을 뷰에게 전달합니다.
- ✓ 뷰는 전달받은 모델을 클라이언트에게 적절한 형태로 표현합니다.



3.4 모델 (2/4) – 커맨드 객체 (1/2)

- ✓ Spring MVC는 객체 속성과 HTML 폼으로 보낸 데이터의 이름이 같으면 자동으로 매핑해서 건네줍니다.
- ✓ 컨트롤러는 뷰의 Form으로 부터 전달된 데이터를 객체 파라미터로 받아 처리할 수 있습니다.
- ✓ 커맨드 객체는 모델에 추가하지 않아도 자동으로 모델에 추가되어 뷰에 전달됩니다.
- ✓ List 형태를 컨트롤러가 받기 위해서는 반드시 감싸주는 커맨드 객체가 있어야 합니다.

```
<form action="posting" method="post">
  <input type="text" name="postings[0].id" />
  <input type="text" name="postings[0].title" />
  <input type="text" name="postings[0].contents" />
  <br/>
  <input type="text" name="postings[1].id" />
  <input type="text" name="postings[1].title" />
  <input type="text" name="postings[1].contents" />
  <br/>
  <input type="submit" />
</form>
```

HTML 폼

```
public class PostingCommand {

    private List<Posting> postings;

    public void setPostings(List<Posting> postings) {
        this.postings = postings;
    }

}
```

커맨드 객체

```
@Controller
@RequestMapping("/posting")
public class PostingController {

    @RequestMapping(method=RequestMethod.POST)
    public String regist(PostingCommand command) {
        ...
    }
}
```

컨트롤러 클래스

3.4 모델 (3/4) – 커맨드 객체 (2/2)

- ✓ EL 표기법을 이용하여 컨트롤러에서 받아온 커맨드 객체를 View에서 사용할 수 있습니다.
- ✓ `@ModelAttribute`를 이용하여 이름을 따로 정해주지 않으면 커맨드 객체의 클래스명을 사용합니다.
- ✓ 이름을 변경하고 싶다면, 해당 커맨드 객체 앞에 `@ModelAttribute`를 붙여서 이름을 지정합니다.

```
@Controller
public class BoardController {

    @RequestMapping(value="/board/regist", method=RequestMethod.POST)
    public String regist(Board command) {
        ...
    }
}
```

View에서는 커맨드 객체의 클래스명(첫 글자 소문자)을 사용하여 커맨드 객체에 접근합니다.

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

View 코드 (JSP)

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

    @RequestMapping(method=RequestMethod.POST)
    public String regist(@ModelAttribute("faq") Board command) {
        ...
    }
}
```

`@ModelAttribute`를 파라미터 앞에 붙여서 설정하면, View에서 사용하는 이름을 변경할 수 있습니다.

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

View 코드 (JSP)

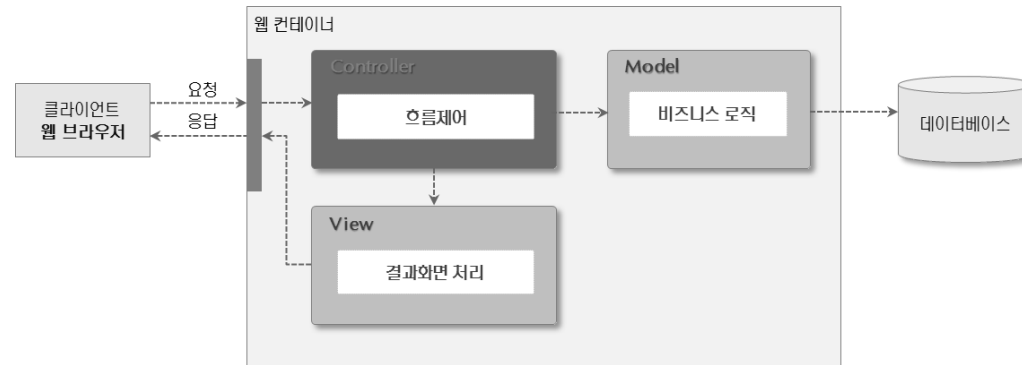
3.4 모델 (4/4) – @RequestParam, @ModelAttribute

- ✓ @RequestParam 은 메소드 파라미터를 요청 파라미터에서 1:1로 받을 경우 사용합니다.
- ✓ @ModelAttribute 는 요청 파라미터를 객체 형태로 받기 위해 사용합니다.
- ✓ 검색 조건과 같이 여러 파라미터를 객체 형태로 받거나 폼 Submit으로 전달되는 파라미터를 객체로 받는데 유용합니다.
- ✓ @ModelAttribute가 붙은 객체는 자동으로 Model에 추가되므로 뷰에서 바로 사용할 수 있습니다.

```
@RequestMapping(method=RequestMethod.POST)
public String regist(@ModelAttribute("posting") Posting posting) {
    ...
    return "posting";
}
```

3.5 요약

- ✓ DispatcherServlet은 클라이언트 요청을 받아 적절한 객체에게 위임하는 프론트 컨트롤러 입니다.
- ✓ 컨트롤러는 요청 내용을 처리하고, 처리결과를 View에게 전달하여 결과화면을 생성하게 합니다.
- ✓ 뷰(View)는 모델을 전달 받아 모델의 정보를 다양한 형식으로 표현하는 기능을 담당합니다.
- ✓ 모델(Model)은 뷰와 컨트롤러 사이에서 주고 받는 정보를 담고 있습니다.



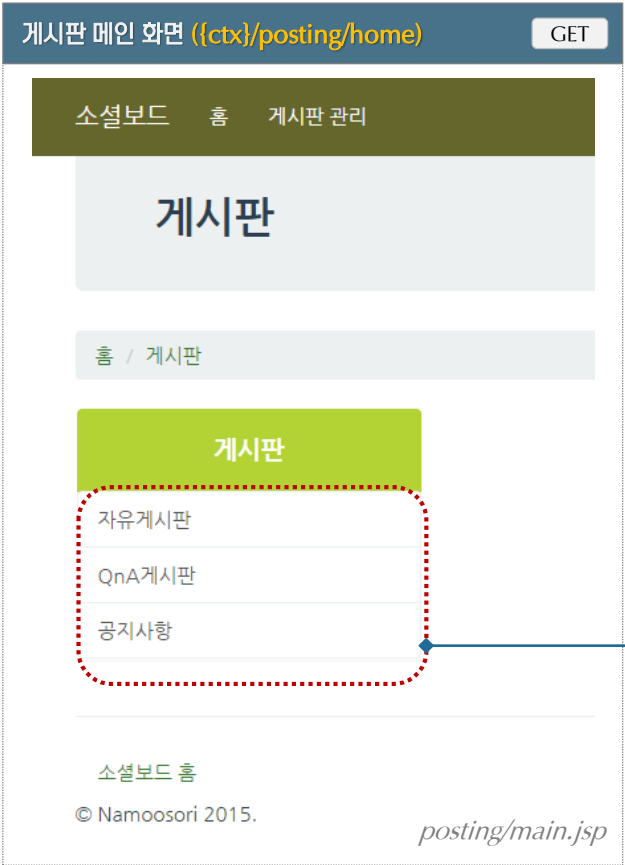


4. MVC 웹 프로그래밍 II

- 4.1 게시판 메인화면
- 4.2 게시물 목록조회
- 4.3 게시물 등록
- 4.4 게시물 상세조회
- 4.5 게시물 수정
- 4.6 게시물 삭제
- 4.7 요약

4.1 게시판 메인화면

- ✓ 게시판 메인화면은 개설된 게시판 목록을 조회하여 보여주는 화면입니다.
- ✓ PostingController 클래스를 생성하여, @Controller 어노테이션을 붙여줍니다.
- ✓ 컨트롤러 클래스와 요청 처리 메소드에서 @RequestMapping을 정의하였으므로, /posting/home 이 URL이 됩니다.
- ✓ HttpServletRequest 객체 속성으로 boardList 조회결과를 추가하여, JSP에서 게시판 목록을 출력하도록 합니다.



```
@Controller
@RequestMapping("/posting")
public class PostingController {
    //
    @Autowired
    private SocialBoardService socialBoardService;

    @Autowired
    private PostingService postingService;

    //-----
    // 게시판 메인

    @RequestMapping(value = "/home", method = RequestMethod.GET)
    public String main(HttpServletRequest req) {
        //
        List<SocialBoard> socialBoards = socialBoardService.findAllSocialBoards();
        req.setAttribute("boardList", socialBoards);

        return "posting/main";
    }
    ...
}
```

PostingController.java

4.2 게시물 목록조회

- ✓ 게시물 목록조회화면은 게시판의 게시물 목록을 보여주는 화면입니다.
- ✓ 게시물 목록조회 화면을 분석하여 컨트롤러에서 구현해야 할 기능을 식별해 봅시다.
- ✓ 먼저 좌측 게시판 목록 영역을 표현하기 위하여, 게시판 목록을 조회합니다.
- ✓ 선택한 게시판에 대한 등록된 게시물 목록을 표현하기 위하여, 게시물 목록을 조회합니다.



```
@RequestMapping(value = "/list", method = RequestMethod.GET)
public String list(@RequestParam("boardUsid") String boardUsid,
    @RequestParam("page") String page,
    HttpServletRequest req) {
    //
    SocialBoard socialBoard = socialBoardService.findSocialBoard(boardUsid);

    List<SocialBoard> socialBoards = socialBoardService.findAllSocialBoards();

    PageCriteria pageCriteria = new PageCriteria(Integer.parseInt(page), 3);

    OffsetKey offsetKey = pageCriteria.toOffsetKey();
    OffsetList<Posting> offsetList =
        postingService.findPostingsByCondition(boardUsid, offsetKey);

    // Convert OffsetList to Page
    Page<Posting> postings = new Page<Posting>(offsetList, pageCriteria);

    req.setAttribute("boardUsid", boardUsid);
    req.setAttribute("boardList", socialBoards);
    req.setAttribute("socialBoard", socialBoard);
    req.setAttribute("postings", postings);

    return "posting/list";
}
```

PostingController.java

4.3 게시물 등록

- ✓ 게시물 목록조회 화면에서 [등록] 버튼을 누르면, 게시물 등록화면으로 이동합니다.
- ✓ 게시물 등록화면을 분석하여 컨트롤러에서 구현해야 할 기능을 식별해 봅시다.
- ✓ 먼저 좌측 게시판 목록 영역을 표현하기 위하여, 게시판 목록을 조회합니다.
- ✓ 등록할 게시물 내용을 입력한 후, [확인] 버튼을 누르면 게시물을 등록합니다.

게시판 목록조회 화면 ((ctx)/posting/create) GET POST

소셜보드 홈 게시판 관리

홈 / 게시판 / 포스팅 등록

게시판
자유게시판

게시글 등록

제목

글쓴이

글쓴이 이메일

내용

posting/create.jsp

```
@RequestMapping(value = "/create", method = RequestMethod.GET)
public String create(@RequestParam("boardUsid") String boardUsid,
    HttpServletRequest req) {
    List<SocialBoard> socialBoards = socialBoardService.findAllSocialBoards();

    req.setAttribute("boardUsid", boardUsid);
    req.setAttribute("boardList", socialBoards);
    return "posting/create";
}

@RequestMapping(value = "/create", method = RequestMethod.POST)
public ModelAndView create(
    @RequestParam("boardUsid") String boardUsid,
    @RequestParam("title") String title,
    @RequestParam("contents") String contents,
    @RequestParam("writerEmail") String writerEmail,
    @RequestParam("writerName") String writerName,
    HttpServletRequest req) {
    PostingCdo postingCdo = new PostingCdo(title, writerEmail, contents);
    if (writerEmail != null) postingCdo.setWriterName(writerName);

    postingService.registerPosting(boardUsid, postingCdo);

    String message = "작성한 글이 저장되었습니다.";
    String linkURL = "posting/list?boardUsid="+boardUsid+"&page=1";

    return MessagePage.information(message, linkURL);
}
```

PostingController.java

4.4 게시물 상세조회

- ✓ 게시물 상세조회 화면은 특정 게시물에 대한 상세정보를 보여주는 화면입니다.
- ✓ 게시물 상세조회 화면을 분석하여 컨트롤러에서 구현해야 할 기능을 식별해 봅시다.
- ✓ 먼저 좌측 게시판 목록 영역을 표현하기 위하여, 게시판 목록을 조회합니다.
- ✓ 선택 게시물의 내용을 표현하기 위하여, 게시물 상세정보를 조회합니다.



```
@RequestMapping(value = "/detail", method = RequestMethod.GET)
public String detail(@RequestParam("boardUsid") String boardUsid,
    @RequestParam("postingUsid") String postingUsid,
    HttpServletRequest req) {
    //
    Posting posting = postingService.findPosting(postingUsid);
    SocialBoard socialBoard =
        socialBoardService.findSocialBoard(boardUsid);
    List<SocialBoard> socialBoards =
        socialBoardService.findAllSocialBoards();

    req.setAttribute("posting", posting);
    req.setAttribute("boardUsid", boardUsid);
    req.setAttribute("boardList", socialBoards);
    req.setAttribute("socialBoard", socialBoard);

    return "posting/detail";
}
```

PostingController.java

4.5 게시물 수정 (1/2) – 화면조회

- ✓ 게시물 상세조회 화면에서 [수정] 버튼을 클릭하면 게시물 수정화면으로 이동합니다.
- ✓ 게시물 수정 화면은 특정 게시물을 내용을 수정하는 화면입니다.
- ✓ 먼저 좌측 게시판 목록 영역을 표현하기 위하여, 게시판 목록을 조회합니다.
- ✓ 선택 게시물의 수정 전 내용을 표현하기 위하여, 게시물 상세정보를 조회합니다.

게시판 수정 화면 ((ctx)/posting/update) GET

소셜보드 홈 게시판 관리

게시판

게시판 / 자유게시판

게시판

자유게시판

자유게시판

글쓴이
나무 (namoo@namoosori.com)

제목
Namoosori 소셜보드 오픈

내용
Namoosori 소셜보드를 드디어 오픈합니다.

저장 취소

posting/update.jsp

```
@RequestMapping(value = "/update", method = RequestMethod.GET)
public String update(
    @RequestParam("boardUsid") String boardUsid,
    @RequestParam("postingUsid") String postingUsid,
    HttpServletRequest req) {
    //
    SocialBoard socialBoard =
        socialBoardService.findSocialBoard(boardUsid);
    Posting posting = postingService.findPosting(postingUsid);
    List<SocialBoard> socialBoards =
        socialBoardService.findAllSocialBoards();

    req.setAttribute("posting", posting);
    req.setAttribute("boardUsid", boardUsid);
    req.setAttribute("boardList", socialBoards);
    req.setAttribute("socialBoard", socialBoard);

    return "posting/update";
}
```

PostingController.java

4.5 게시물 수정 (2/2) – 수정처리

- ✓ 게시물 수정 화면에서 내용을 수정 후, [저장] 버튼을 누르면 게시물을 수정합니다.
- ✓ 화면에서 파라미터로 전달된 값을 메소드 매개변수로 받기 위하여 @RequestParam 어노테이션을 적용합니다.

게시판 수정 처리 ((ctx)/posting/update) POST

소셜보드 홈 게시판 관리

게시판

게시판 / 자유게시판

게시판
자유게시판

자유게시판

글쓴이
나무 (namoo@namoosori.com)

제목
Namoosori 소셜보드 오픈

내용
Namoosori 소셜보드를 드디어 오픈합니다.

저장

취소

posting/update.jsp

```
@RequestMapping(value = "/update", method = RequestMethod.POST)
public ModelAndView update(
    @RequestParam("title") String title,
    @RequestParam("contents") String contents,
    @RequestParam("postingUsid") String postingUsid,
    @RequestParam("boardUsid") String boardUsid) {
    //
    NameValueCollection nameValues = NameValueCollection.getInstance();
    nameValues.add(Posting.MODIFIABLE_TITLE, title);
    nameValues.add(Posting.MODIFIABLE_CONTENTS, contents);

    postingService.modifyPosting(postingUsid, nameValues);

    String message = "포스트가 수정되었습니다.";
    String linkURL = "posting/detail?boardUsid=" +
        boardUsid+"&postingUsid=" + postingUsid;

    return MessagePage.information(message, linkURL);
}
```

PostingController.java

4.6 게시물 삭제

✓ 게시물 상세조회 화면에서 [삭제] 버튼을 누르면 게시물을 삭제합니다.



```
@RequestMapping(value = "/delete", method = RequestMethod.GET)
public ModelAndView delete(
    @RequestParam("boardUsid") String boardUsid,
    @RequestParam("postingUsid") String postingUsid,
    HttpServletRequest req) {

    postingService.removePosting(postingUsid);

    String message = "포스트가 삭제되었습니다.";
    String linkURL = "posting/list?boardUsid=" + boardUsid + "&page=1";

    return MessagePage.information(message, linkURL);
}
```

PostingController.java

4.7 요약

- ✓ 2장에서 진행한 게시판 관리 예제에 덧붙여 게시물 관리 기능을 구현합니다.
- ✓ 2장과 같이 제시된 UI 자원과 비즈니스 로직을 활용합니다.
- ✓ 게시물 목록조회는 제시된 타입들을 이용하여 페이지로 구성합니다.
- ✓ 3장의 내용을 기초하여 더욱 보기 좋고 간편하게 모델을 받고 뷰를 지정할 수 있습니다.



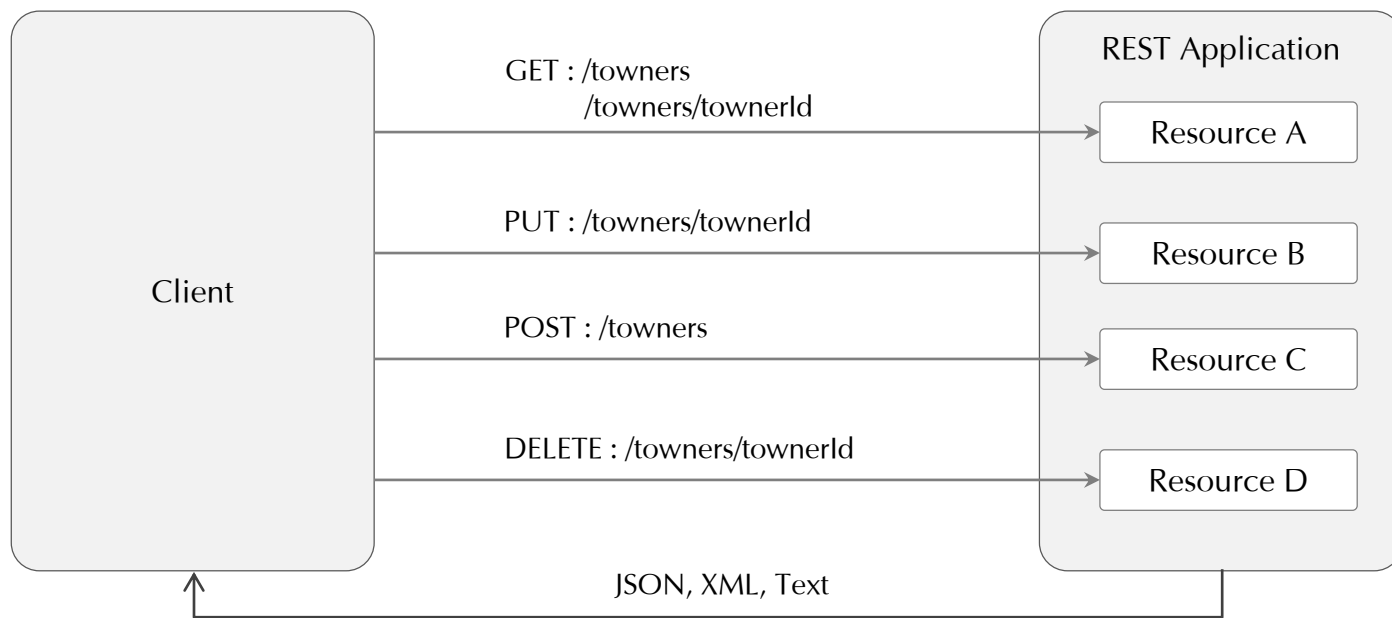


5. Spring MVC 활용

- 5.1 RESTful 웹 서비스
- 5.2 요청과 응답 가로채기
- 5.3 예외처리
- 5.4 폼 입력 값 검증
- 5.5 요약

5.1 RESTful 웹 서비스 (1/7) – 개요

- ✓ REST는 REpresentational State Transfer 의 줄임 말로,
- ✓ 리소스를 URI를 통해 고유하게 식별되게 정의하는 아키텍처 스타일을 REST라고 합니다.
- ✓ REST에서는 HTTP Method(GET, POST, PUT, DELETE 등)를 통해 리소스와 상호작용 합니다.
- ✓ REST 스타일을 준수하는 서비스를 RESTful 웹 서비스라고 하며 JSON, XML, 일반텍스트 등을 요청/응답에 사용합니다.



5.1 RESTful 웹 서비스 (2/7) – 메시지 변환

- ✓ 웹 시스템 간의 XML, JSON 형식의 데이터를 주고받는 경우는 다음과 같은 메시지 변환을 거칩니다.
- ✓ 언마샬링은 XML이나 JSON과 같은 전문을 Java 객체로 변환하는 과정입니다.
- ✓ 마샬링은 Java 객체를 XML이나 JSON 과 같은 전문으로 변환하는 과정입니다.
- ✓ `@RequestBody`, `@ResponseBody` 어노테이션을 메소드에 추가하면 SpringMVC는 메시지를 변환합니다.



5.1 RESTful 웹 서비스 (3/7) – @RestController

- ✓ 스프링은 RESTful 스타일을 지원하는 @RestController를 제공합니다. (Spring 4+)
- ✓ @RestController 어노테이션이 적용된 모든 컨트롤러는 view 대신에 특정 객체를 리턴 합니다.
- ✓ 컨트롤러가 리턴한 객체는 메시지 변환 설정에 따라 변환됩니다. (예, 객체 → JSON)
- ✓ @RestController = @Controller + @ResponseBody

```
@RestController
public class PostingWsController {

    ...

    @RequestMapping(value = "/board/{boardUsid}/posting/{postingUsid}", method = RequestMethod.GET)
    public ResponseMessage retrieve(@PathVariable("boardUsid") String boardUsid,
        @PathVariable("postingUsid") String postingUsid, HttpServletRequest req) {

        Posting posting = postingService.findPosting(postingUsid);
        SocialBoard socialBoard = socialBoardService.findSocialBoard(boardUsid);

        Map<String, Object> result = new HashMap<String, Object>();
        result.put("posting", posting);
        result.put("socialBoard", socialBoard);

        return new ResponseMessage(result);
    }

    ...
}
```

5.1 RESTful 웹 서비스 (4/7) – @PathVariable

- ✓ @RequestMapping 에 설정한 URI의 특정 부분에 접근할 때, URI 템플릿을 지정합니다.
- ✓ URI 템플릿은 중괄호 안에 변수 이름을 사용해서 표현합니다.
- ✓ @PathVariable은 URI 템플릿 변수 값을 메소드 파라미터에 할당할 때 사용합니다.

```
@Controller
public class TeamController {
    @RequestMapping("/clubs/{clubId}/teams/{teamId}")
    public String teamInfo(
        @PathVariable("clubId") String clubId,
        @PathVariable("teamId") String teamId, ModelMap model){
        ...
    }
}
```

```
@Controller
@RequestMapping("/clubs/{clubId}")
public class TeamController {

    @RequestMapping("/teams/{teamId}")
    public String teamInfo(
        @PathVariable("clubId") String clubId,
        @PathVariable("teamId") String teamId, ModelMap model){
        ...
    }
}
```

5.1 RESTful 웹 서비스 (5/7) – @RequestBody, @ResponseBody (1/2)

- ✓ HTTP 프로토콜은 header와 body로 구성되어 있습니다.
- ✓ @RequestBody는 HTTP 요청 메시지 Body를 자바 객체로 변환하는데 사용합니다.
- ✓ @ResponseBody는 자바 객체를 HTTP 응답 메시지 Body로 변환합니다.
- ✓ SpringMVC는 HttpMessageConverter를 사용하여 Java 객체와 요청/응답 Body 사이의 변환을 처리합니다.

```
<script type="text/javascript">
$(document).ready(function() {
    $("#getjson").click(function(){
        $.ajax({
            url : "/towners/json",
            method : "POST",
            contentType : "application/json", ← json 타입의 response를 Accept함
            success : function() {
                alert("전송 성공");
            },
            error : function(XHR, textStatus, errorThrown) {
                alert("Error: " + textStatus);
                alert("Error: " + errorThrown);
            }
        });
    });
});
</script>
```

```
@RequestMapping(value="/towners/json", produces = {MediaType.APPLICATION_JSON_VALUE})
public @ResponseBody List<Towner> listJson(){
    return towner.retrieveTowners();
}
```

5.1 RESTful 웹 서비스 (6/7) – @RequestBody, @ResponseBody (2/2)

- ✓ @RequestBody 어노테이션을 사용하면, 적절한 HttpMethodConverter 구현을 통해 해당 객체로 변환해 줍니다.
 - 요청 body를 @RequestBody 어노테이션이 적용된 자바 객체로 변환할 때에는 HTTP요청 헤더의 Content-Type 헤더에 명시된 미디어 타입(MIME)을 지원하는 HttpMessageConverter를 구현체로 사용합니다.
 - @ResponseBody 어노테이션을 이용해서 리턴하는 객체를 HTTP 메시지의 body로 변환할 때에는 HTTP요청 헤더의 Accept 헤더에 명시된 미디어 타입을 지원하는 HttpMessageConverter 구현체를 선택합니다.

구현 클래스	설명
ByteArrayHttpMessageConverter (*)	HTTP 메시지와 byte 배열 사이의 변환을 처리. 컨텐츠 타입은 application/octet-stream
StringHttpMessageConverter (*)	HTTP 메시지와 String 사이의 변환을 처리한다. 컨텐츠 타입은 text/plain;charset=ISO-8859-1
FormHttpMessageConverter (*)	HTML 폼 데이터를 MultiValueMap으로 전달받을 때 사용 컨텐츠 타입은 application/x-www-form-urlencoded
SourceHttpMessageConverter (*)	HTTP 메시지와 javax.xml.transform.Source 사이의 변환을 처리. 컨텐츠 타입은 application/xml 또는 text/xml
MarshallingHttpMessageConverter	스프링의 Marshaller와 Unmarshaller를 이용해서 XML HTTP 메시지와 객체 사이의 변환을 처리한다. application/xml 또는 text/xml
MappingJacksonHttpMessageConverter	Jackson 라이브러리를 이용해서 JSON HTTP 메시지와 객체 사이의 변환을 처리한다. 컨텐츠 타입은 application/json

- . HttpMessageConverter 구현 클래스 (일부)
- . AnnotationMethodHandlerAdapter는 (*) 표시된 클래스를 기본적으로 사용한다.

- ✓ namoo.board.web.ajax 프로젝트는 RESTful 웹 서비스를 구현하여 게시판 기능을 Ajax 방식으로 변경합니다.
- ✓ 먼저, 서버에서 제공하는 자원에 대한 REST API를 설계합니다. (자원을 나타내는 URI + HTTP Method)
- ✓ @RestController 어노테이션을 적용하여 RESTful 웹 서비스를 제공하는 컨트롤러 클래스를 개발합니다.
- ✓ UI에서는 jQuery 를 사용하여 Ajax를 구현하고, 화면을 JavaScript로 동적으로 생성합니다.

REST API 설계		
기능	자원을 나타내는 URI	HTTP Method
게시판 개설	/ws/board	POST
게시판 수정	/ws/board/{boardUsid}	PUT
게시판 삭제	/ws/board/{boardUsid}	DELETE
게시물 목록조회	/ws/board/{boardUsid}/postings	GET
게시물 상세조회	/ws/board/{boardUsid}/posting/{postingUsid}	GET
게시물 등록	/ws/board/{boardUsid}/posting	POST
게시물 수정	/ws/board/{boardUsid}/posting/{postingUsid}	PUT
게시물 삭제	/ws/board/{boardUsid}/posting/{postingUsid}	DELETE

실습 프로젝트 준비 및 환경설정

실습 프로젝트 Import

Jackson 라이브러리를
Maven Dependency 에 추가

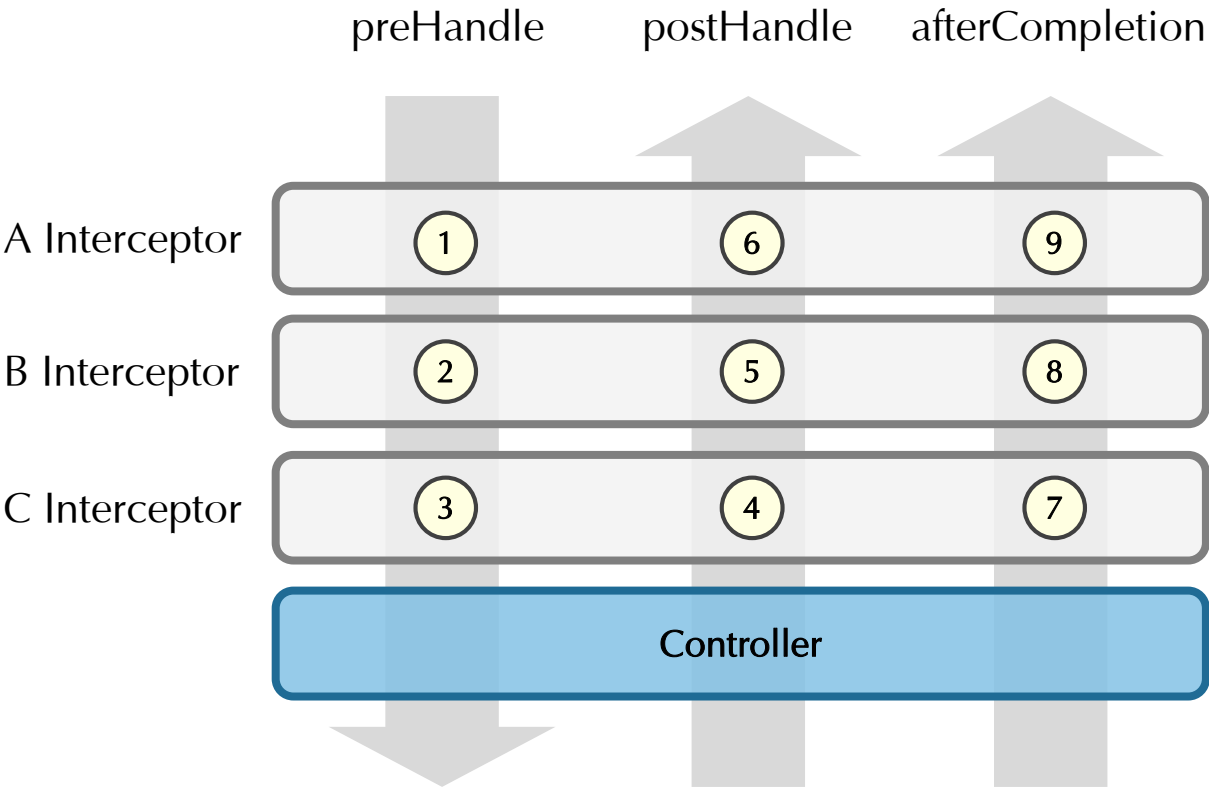
REST API 개발 및 Ajax 프로그래밍

REST API 구현

jQuery를 사용한
Ajax프로그래밍

5.2 요청과 응답 가로채기 (1/5) – 개요

- ✓ DispatcherServlet이 컨트롤러를 호출하기 전과 후에 요청과 응답을 참조하여 가공하는 기능을 인터셉터라고 합니다.
- ✓ 컨트롤러 호출 전과, 컨트롤러 호출 후, 뷰 호출 후의 세 군데에서 요청과 응답을 가로챌 수 있습니다.
- ✓ 로깅이나 모니터링 정보 수집, 접근제어 처리 등과 같이 여러 컨트롤러에 공통으로 적용할 때 유용합니다.
- ✓ 인터셉터를 체인형태로 여러 개 적용할 수 있습니다. (순서주의!!)



5.2 요청과 응답 가로채기 (2/5) – HandlerInterceptor (1/2)

- ✓ 스프링은 여러 컨트롤러에 공통으로 적용되는 기능을 쉽게 구현할 수 있도록 HandlerInterceptor를 제공합니다.
- ✓ HandlerInterceptor는 요청과 응답을 참조하여 가공할 수 있는 일종의 필터입니다.
- ✓ HandlerInterceptor 인터페이스를 구현하거나 HandlerInterceptorAdaptor 클래스를 상속합니다.
- ✓ HandlerInterceptor 인터페이스는 처리 시점에 따른 세 가지 메소드를 제공합니다.

```
public class LoggingInterceptor implements HandlerInterceptor {  
    @Override  
    public boolean preHandle(HttpServletRequest request,  
                             HttpServletResponse response, Object handler) throws Exception {  
        System.out.println("intercept!! preHandle");  
        return true;  
    }  
  
    @Override  
    public void postHandle(HttpServletRequest request,  
                           HttpServletResponse response, Object handler,  
                           ModelAndView modelAndView) throws Exception {  
        System.out.println("intercept!! postHandle");  
    }  
  
    @Override  
    public void afterCompletion(HttpServletRequest request,  
                                HttpServletResponse response, Object handler, Exception ex)  
        throws Exception {  
        System.out.println("intercept!! afterCompletion");  
    }  
}
```

컨트롤러가 호출되기 전에 호출됩니다.
false를 리턴하면 request를 바로 종료합니다.

컨트롤러가 실행되고 난 후 호출됩니다.
preHandle()에서 false를 리턴한 경우 실행되지 않습니다.

모든 작업이 완료된 후에 호출됩니다.
주로 요청 처리 중에 사용한 자원을 반납할 때 사용합니다.

5.2 요청과 응답 가로채기 (3/5) – HandlerInterceptor (2/2)

- ✓ HandlerInterceptor를 스프링 설정파일에 등록하면, 모든 컨트롤러에 적용됩니다.
- ✓ 컨트롤러 메소드 전/후/응답완료 후 호출되는 것을 확인할 수 있습니다.

servlet-context.xml

```
<interceptors>
  <beans:bean class="namoo.springmvc.cookbook.interceptor.LoggingInterceptor" />
</interceptors>
```

임의의 메소드 실행 결과

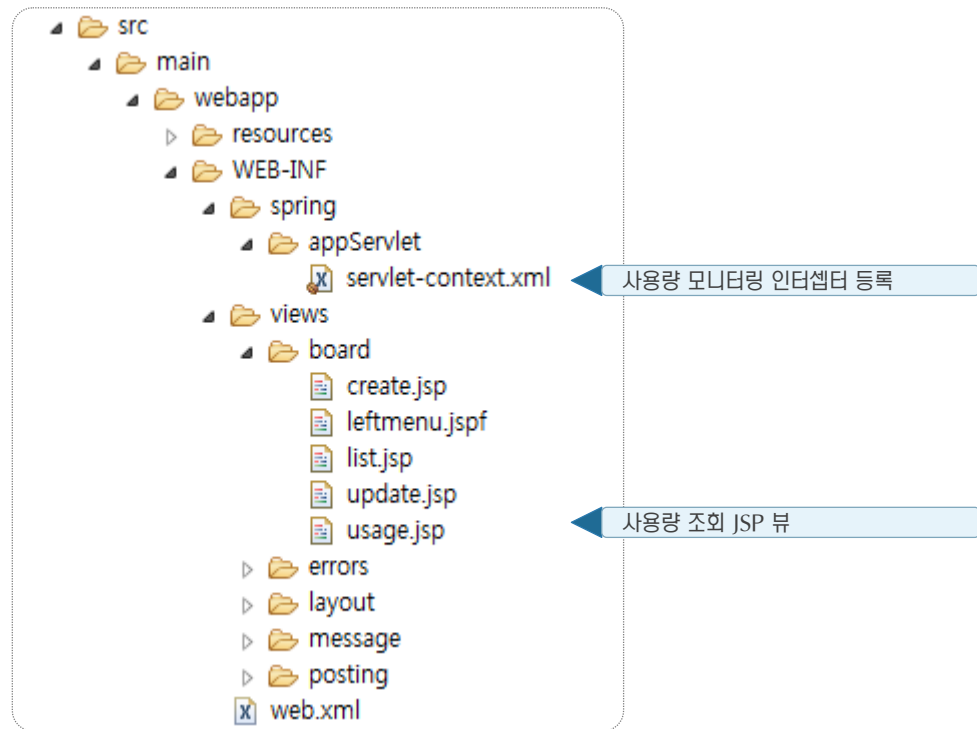
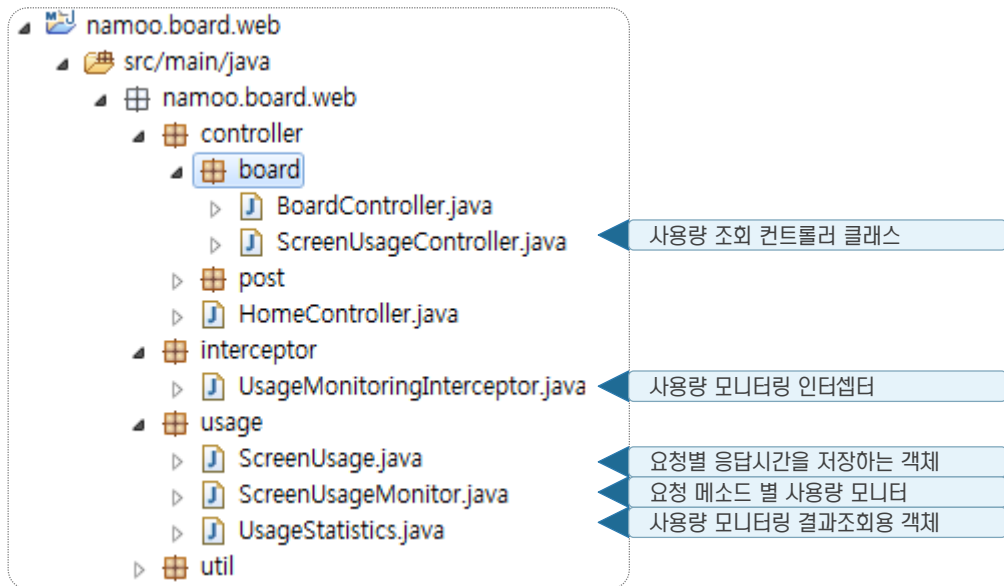
```
intercept!! preHandle
intercept!! postHandle
intercept!! afterCompletion
```

5.2 요청과 응답 가로채기 (4/5) – URL 매핑

- ✓ HandlerInterceptor는 기본적으로 모든 요청에 대해서 적용됩니다.
- ✓ 특정 요청 URL에 대해서만 인터셉터를 적용하고 싶은 경우, interceptor 요소를 사용합니다.
- ✓ HandlerInterceptor 요청 경로 패턴은 mapping 요소를 사용해서 지정합니다.

```
<interceptors>
  <interceptor>
    <mapping path="/recipe"/>
    <mapping path="/towner"/>
    <beans:bean class="namoo.springmvc.cookbook.interceptor.LoggingInterceptor" />
  </interceptor>
  <beans:bean class="namoo.springmvc.cookbook.interceptor.AnotherInterceptor" />
</interceptors>
```

- ✓ 컨트롤러의 요청 메소드 별로 평균 응답시간과 요청 횟수를 관리하는 사용량 모니터링 기능을 구현합니다.
- ✓ 응답시간을 계산하기 위해 컨트롤러 수행 전후에 추가적인 기능을 삽입할 수 있는 인터셉터를 사용합니다.
- ✓ 사용량 모니터 클래스와 인터셉터 클래스를 빈으로 등록합니다.
- ✓ 측정한 사용량을 화면을 통해 볼 수 있도록 컨트롤러를 추가하고, View를 구현합니다.



5.3 예외처리 (1/4) – 개요

- ✓ 컨트롤러 작업 중에 발생한 예외를 적절히 처리하는 것은 UX 측면에서 중요한 부분입니다.
- ✓ 별다른 처리를 하지 않으면 브라우저는 HTTP 500 에러와 서블릿 컨테이너가 출력한 에러 페이지를 표시합니다.
- ✓ SpringMVC의 HandlerExceptionHandlerResolver를 사용해서 예외 발생시 적절한 안내 페이지를 보여줄 수 있습니다.
- ✓ SpringMVC는 4개의 HandlerExceptionHandlerResolver 구현체를 제공합니다.

구현 클래스	설명
AnnotationMethodHandlerExceptionHandlerResolver	예외가 발생한 컨트롤러 내의 메소드 중에서 @ExceptionHandler 어노테이션이 적용된 메소드를 찾아서 예외 처리를 위임해주는 HandlerExceptionHandlerResolver 입니다.
DefaultHandlerExceptionHandlerResolver	다른 HandlerExceptionHandlerResolver에서 처리하지 못한 예외에 대해서 처리하는 표준 예외처리 핸들러입니다. 스프링 MVC 내부에서 발생하는 예외를 처리하므로 특별히 신경 쓰지 않아도 됩니다.
SimpleMappingExceptionHandlerResolver	web.xml의 <error-page>와 유사하게 예외 타입 별로 처리할 뷰 이름을 지정합니다.
ResponseStatusExceptionHandlerResolver	예외 발생시 해당 예외 코드를 클라이언트에게 돌려주는 것이 아니라, 특정 HTTP 응답 상태 코드로 전환하여 의미 있는 HTTP 응답 상태를 리턴하는 방법입니다.

5.3 예외처리 (2/4) – @ExceptionHandler

- ✓ 예외 발생 시, @ExceptionHandler 어노테이션이 붙은 메소드가 호출되어 예외를 처리합니다.
- ✓ @ExceptionHandler가 붙은 예외처리 메소드는 @RequestMapping 메서드와 유사하게 구현합니다.
- ✓ 예외타입 지정 시, 해당 예외를 포함한 하위 타입의 예외까지 처리합니다.
- ✓ 메소드에서 처리한 예외 객체는 뷰 코드에서 exception 기본 객체를 이용해서 예외 객체에 접근할 수 있습니다.


```
public class MyController {  
  
    @RequestMapping("/user/{userId}")  
    public ModelAndView getUser(@PathVariable("userId") String userId) {  
        ...  
    }  
  
    @ExceptionHandler({NullPointerException.class})  
    public String handleNullPointerException(NullPointerException e) {  
        return "error/nullException";  
    }  
}
```

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>  
<!DOCTYPE html>  
<html>  
<head>  
<title>Insert title here</title>  
</head>  
<body>  
    <h3>에러 메시지</h3>  
    <p>${exception}</p>  
    <p>${exception.message}</p>  
</body>  
</html>
```

- ✓ @ControllerAdvice 를 활용하여, 동일한 타입의 예외를 하나의 코드에서 공통으로 처리할 수 있습니다.
- ✓ @ControllerAdvice 를 적용한 클래스를 빈으로 등록해야 예외 처리가 가능합니다.
- ✓ 예외가 발생하면, 동일 클래스의 @ExceptionHandler 메소드 중에서 발생한 예외처리 가능한 메소드를 검색합니다.
- ✓ 처리 가능한 메소드가 없을 경우, @ControllerAdvice 클래스에 위치한 @ExceptionHandler 메소드를 검색합니다.

```
@ControllerAdvice("namoo.board.web.controller.ws")
public class WsExceptionHandler {

    @ExceptionHandler(NamooException.class)
    @ResponseBody public ResponseMessage exceptionHandler(NamooException e) {
        //
        return new ResponseMessage(e);
    }
}
```

 namoo.board.web.controller.ws 패키지 아래에 있는 컨트롤러 수행 중 예외가 발생하는 경우, 이 핸들러 클래스에서 예외를 처리합니다. 여기서는 JSON 전문으로 처리실패를 제공하기 위하여 @ResponseBody 어노테이션을 사용하여 ResponseMessage 객체를 리턴하였습니다.

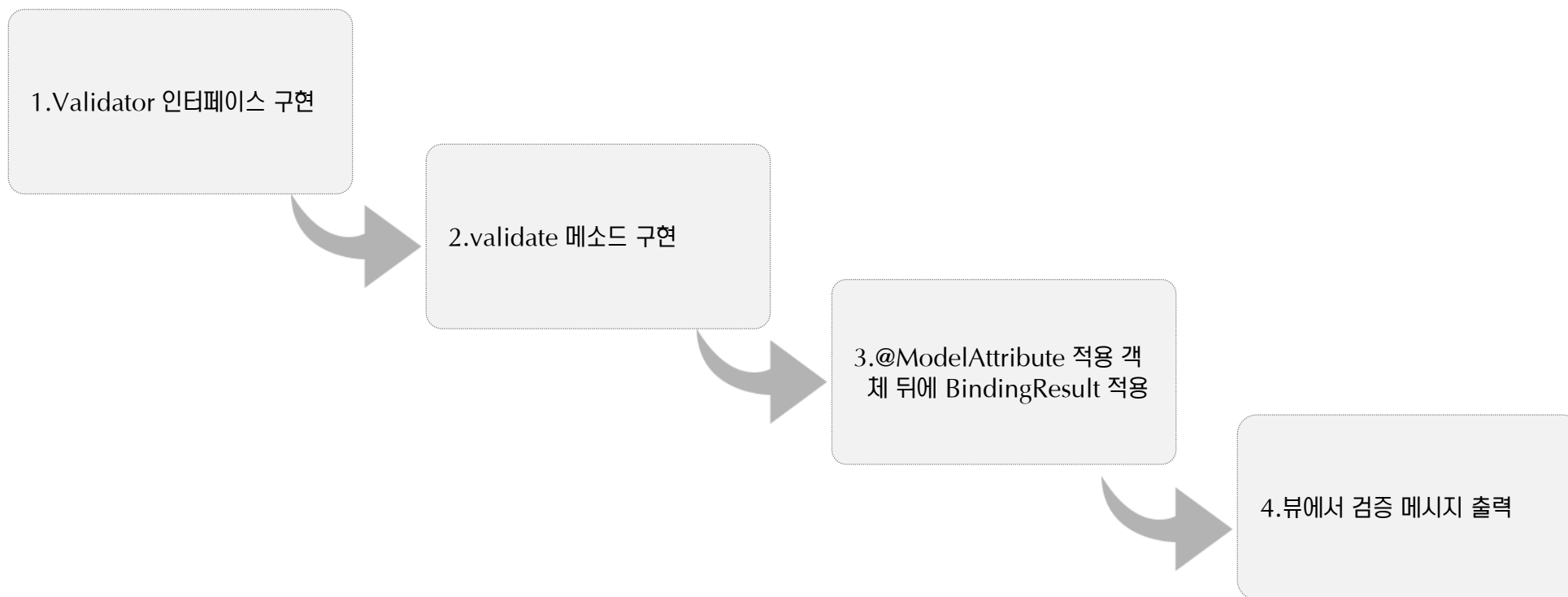
5.3 예외처리 (4/4) – SimpleMappingExceptionHandler

- ✓ SimpleMappingExceptionHandler는 예외 타입별로 에러 페이지를 지정할 수 있습니다.
- ✓ 특정 타입의 예외 발생 시 지정한 에러 페이지를 뷰 페이지로 사용합니다.
- ✓ [실습] namoo.board.web 프로젝트에 NamooException 예외가 발생할 때 에러페이지를 보여주도록 추가합니다.
 - 공통 에러페이지는 /views/errors/error.jsp 에 준비되어 있습니다.

```
<beans:bean
  class="org.springframework.web.servlet.handler.SimpleMappingExceptionHandler">
  <beans:property name="exceptionMappings">
    <beans:props>
      <beans:prop key="java.lang.NullPointerException">nullError</beans:prop>
      <beans:prop key="java.lang.Exception">error</beans:prop>
    </beans:props>
  </beans:property>
</beans:bean>
```

5.4 폼 입력 값 검증 (1/3) – 개요

- ✓ 화면에서는 서버로 입력 값을 전달하기 전에 JavaScript를 사용해서 1차적으로 오류를 검증합니다.
- ✓ 이와는 별개로, SpringMVC 에서는 폼 입력 값으로 전달받은 파라미터 값을 검증하는 기능을 제공합니다.
- ✓ 이 기능은 @ModelAttribute 로 바인딩된 객체를 검증하는데 사용합니다.
- ✓ 스프링은 유효성 검사를 위한 Validator 인터페이스와 검사 결과를 저장 할 Errors 인터페이스를 제공합니다.



5.4 폼 입력 값 검증 (2/3) – Validator 인터페이스 구현

- ✓ Validator 인터페이스와 구현 클래스입니다.
- ✓ 입력값 검사 후 통과하지 못한 경우 페이지에 검증 오류 메시지를 보여주도록 구현합니다.
- ✓ supports와 validate 메소드를 overriding하여 구현합니다.

```
public interface Validator {  
  
    // 해당 클래스 validation 지원여부  
    boolean supports(Class<?> clazz);  
  
    // 검증 결과 문제가 있는 경우 error 객체에 정보를 저장  
    void validate(Object target, Errors errors);  
}
```

```
@Component  
public class RecipeValidator implements Validator {  
  
    @Override  
    public boolean supports(Class<?> clazz) {  
        return (Recipe.class.isAssignableFrom(clazz));  
    }  
  
    @Override  
    public void validate(Object target, Errors errors) {  
        //  
        Recipe recipe = (Recipe) target;  
        if (recipe.getName() == null ||  
recipe.getName().length() == 0) {  
            errors.reject("recipeName", "레시피 이름은 반드시 입력되  
어야 합니다.");  
        }  
    }  
}
```

5.4 폼 입력 값 검증 (3/3) – Controller & JSP 적용

- ✓ 메소드에 적용할 때, 반드시 검증하려는 객체 바로 뒤에 BindingResult 를 추가합니다.
- ✓ 메소드에서 validate 메소드를 호출해서 입력 값에 대한 검증을 수행합니다.
- ✓ 검증결과 오류가 발생한 경우 bindingResult에 해당 오류가 담겨서 리턴 됩니다.
- ✓ 뷰에서는 bindingResult를 받아서 발생한 오류 메시지를 화면에 표시합니다.

```
@Controller
public class RecipeController {

    @Autowired
    private RecipeValidator validator;
    ...

    @RequestMapping(value = "/recipe", method = RequestMethod.POST)
    public String addRecipe(Recipe recipe, BindingResult
bindingResult, Model model) {
        // Validation 체크
        this.validator.validate(recipe, bindingResult);

        if (bindingResult.hasErrors()) {
            // validation 오류가 있는 경우 bindingResult에서 에러 정보를
가지고 처리함
            model.addAttribute("errors", bindingResult);
            return "recipe";
        } else {
            chef.createRecipe(recipe);
            return "redirect:/";
        }
    }
}
```

```
<html>
<head>
<title>Recipe Home</title>
</head>
<body>
<h1>레시피 등록</h1>
<p>레시피를 만드세요.</p>
<form action="{ctx}/recipe" method="post"
<table border="1">
<tr>
<td>레시피명</td>
<td><input type="text" name="
</tr>
<tr>
<td>재료</td>
<td><input type="text" name="
</tr>
<tr>
<td>조리법</td>
<td><textarea rows="10" cols=
name="description"></textarea></td>
</tr>
<tr>
<td colspan="2">
<input type="submit" value="레시피등록" />
<input type="button" value="취소" onclick="javascript:history.back(-1);" />
</td>
</tr>
</table>
<c:forEach var="error" items="{errors.allErrors}">
    ${error.defaultMessage} <br/>
</c:forEach>
</body>
</html>
```

레시피 등록

레시피를 만드세요.

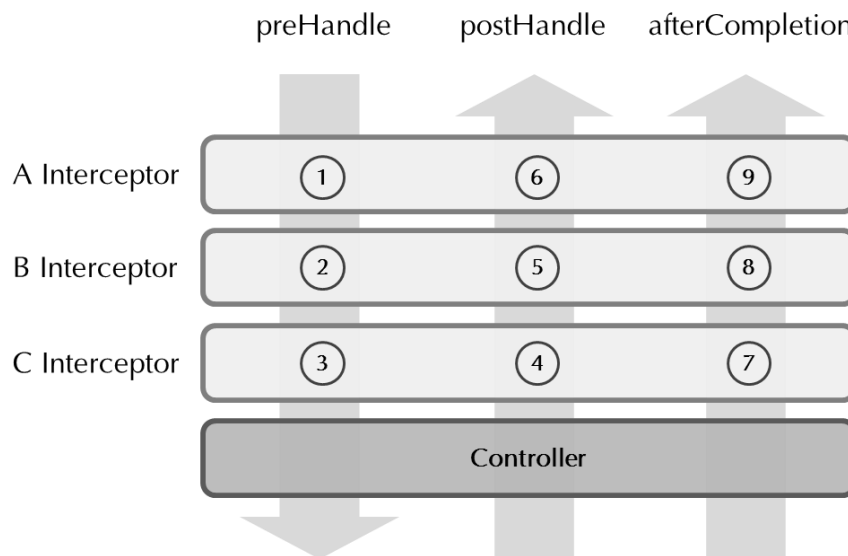
레시피명	<input type="text"/>
재료	<input type="text" value="김치, 돼지고기"/>
조리법	<div>김치를 볶은다음...</div>

레시피등록 취소

레시피 이름은 반드시 입력되어야 합니다.

5.5 요약

- ✓ @RestController, @PathVariable 등을 사용하여 RESTful 웹 서비스를 구현할 수 있습니다.
- ✓ 인터셉터를 사용하여 여러 컨트롤러에 공통으로 적용되는 기능을 쉽게 구현할 수 있습니다.
- ✓ HandlerExceptionResolver 클래스는 컨트롤러에 발생한 예외를 적절한 형태로 처리할 수 있습니다.
- ✓ 입력 값 유효성을 검사하기 위해, Validator 인터페이스와 검사 결과를 저장할 Errors 인터페이스 구현합니다.



♣ [웹프레임워크 – SpringMVC] 요약

1. Spring MVC 소개

- Spring MVC는 MVC 패턴 기반이며, 모든 요청을 받아 각 컨트롤러로 요청을 위임해주는 Front Controller를 사용합니다.
- 설정보다는 관례(CoC) 중심 프레임워크로서, 보편적인 기능은 기본 제공하고 그 외 설정은 확장 가능합니다.
- Spring @MVC라고 불리울 만큼 어노테이션을 이용한 편리하고 효율적인 개발을 지원합니다.

2. MVC 웹 프로그래밍 I

- 제시된 UI와 비즈니스로직을 참고하여 Spring MVC를 이용한 간단한 실습을 진행합니다.
- SpringMVC는 메소드 단위로도 요청을 매핑할 수 있으므로, 하나의 컨트롤러에 구현 가능합니다.

3. Spring MVC 이해

- DispatcherServlet은 클라이언트 요청을 받아 적절한 객체에게 위임하는 프론트 컨트롤러 입니다.
- 컨트롤러는 요청 내용을 처리하고, 처리결과를 View에게 전달하여 결과화면을 생성하게 합니다.
- 뷰(View)는 모델을 전달 받아 모델의 정보를 다양한 형식으로 표현하는 기능을 담당합니다.
- 모델(Model)은 뷰와 컨트롤러 사이에서 주고 받는 정보를 담고 있습니다.

4. MVC 웹 프로그래밍 II

- 2장에서 진행한 게시판 관리 예제에 덧붙여 게시물 관리 기능을 구현합니다.
- 3장의 내용을 기초하여 더욱 보기좋고 간편하게 모델을 받고 뷰를 지정할 수 있습니다.

5. Spring MVC 활용

- @RestController, @PathVariable 등을 사용하여 RESTful 웹 서비스를 구현할 수 있습니다.
- 인터셉터, 예외리졸버 등을 이용하여 복잡한 기능도 간결하게 구현 가능합니다.

토론

- ✓ 질문과 대답
- ✓ 토론

