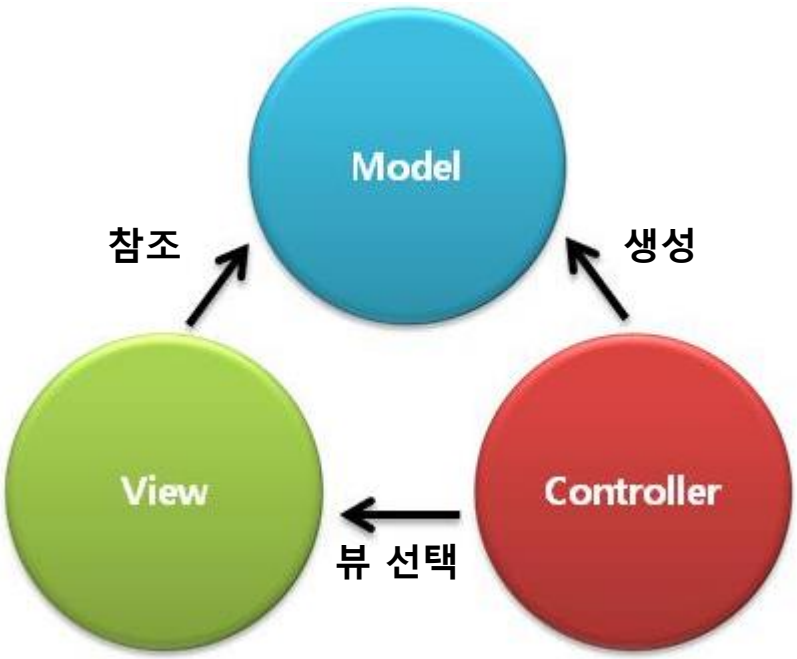
Spring MVC

1-1. 웹 프레임워크 아키텍처 – MVC 패턴

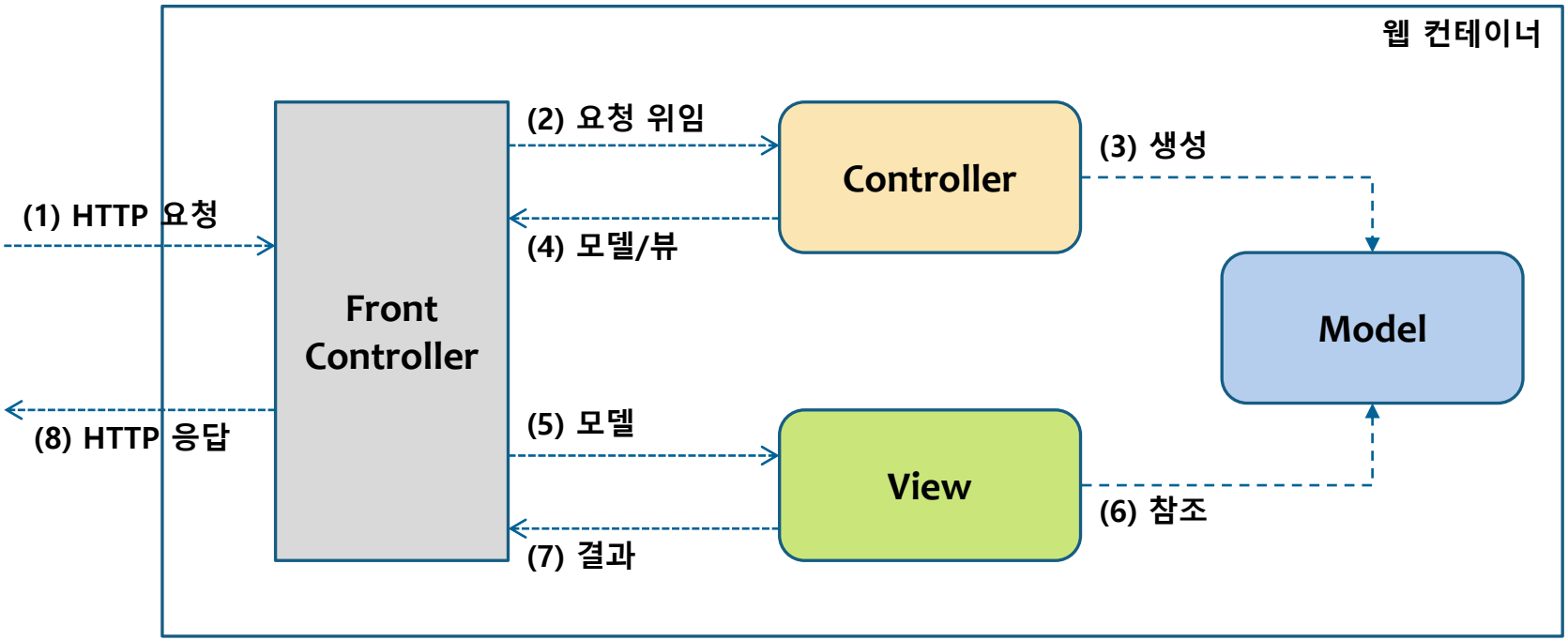
- **MVC (Model – View – Controller) 패턴은 코드를 처리하는 영역을 Model, View, Controller로 분리**
- **영역별로 분리하면 컴포넌트 간의 결합도가 낮아져, 컴포넌트 변경이 다른 영역 컴포넌트에 영향을 주지 않게 됨**
- **화면과 비즈니스 로직을 분리하므로, 기능 확장과 유지보수가 편리해 짐**
- **개발과정이 복잡하고, 초기 개발 속도가 느림**



MVC	설명
Model	- 애플리케이션 상태의 캡슐화 - 상태 쿼리에 대한 응답 - 어플리케이션의 기능 표현 - 변경을 뷰에 통지
View	- 모델을 화면에 시각적으로 표현 - 모델에게 업데이트 요청 - 사용자의 입력을 컨트롤러에 전달 - 컨트롤로가 뷰를 선택하도록 허용
Controller	- 어플리케이션의 행위 정의 - 사용자 액션에 대한 모델 업데이트와 매핑 - 응답에 대한 뷰 선택

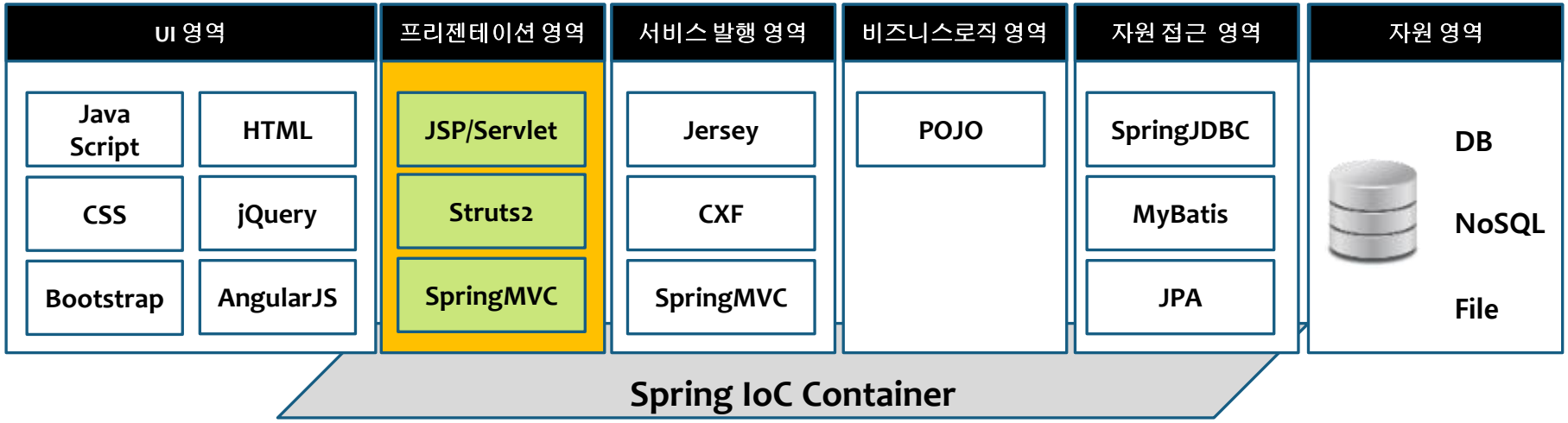
1-1. 웹 프레임워크 아키텍처 – Front Controller 패턴

- **웹 MVC 프레임워크는 JEE 패턴의 Front Controller 패턴에 기초**
- **Front Controller는 요청을 받아 어느 컨트롤러에 요청을 전송할 지 결정하여 다른 컴포넌트에 처리를 위임**
- **프리젠테이션 계층의 제일 앞에서 모든 요청을 최초로 수신하여 처리하는 서블릿을 말함**
- **웹 MVC 프레임워크는 요청을 각 컨트롤러로 분기하는 중앙 서블릿 중심으로 설계**



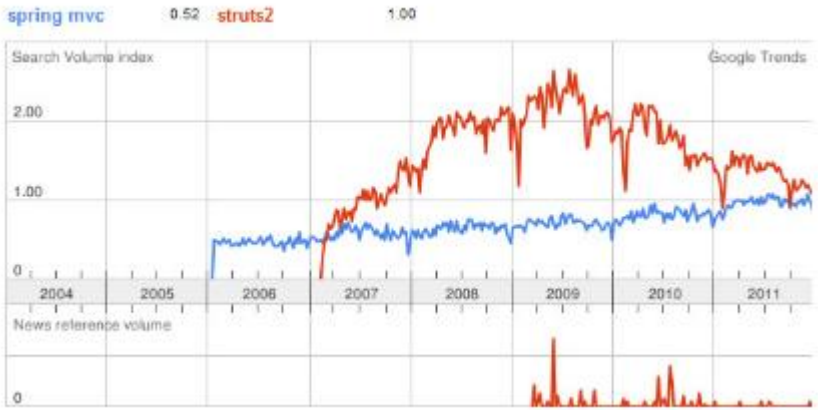
1-2. Spring MVC 개요 – Spring 기반 웹 어플리케이션

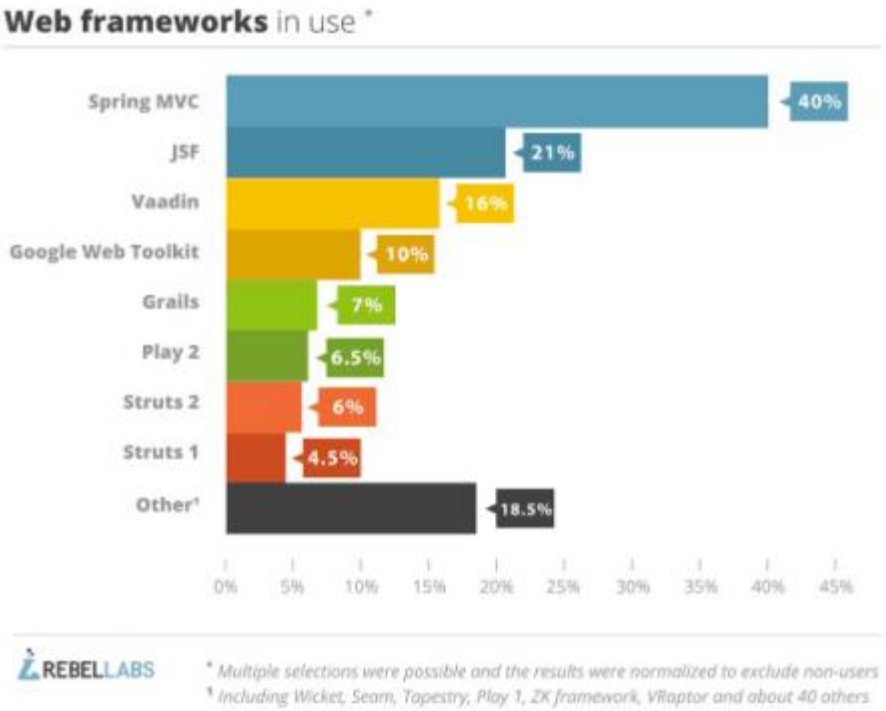
- **Spring MVC는 스프링에서 프리젠테이션 계층을 담당하는 서블릿 기반 MVC 프레임워크**
- **Spring MVC는 다른 웹 프레임워크에 비해 특정 클래스 상속, 참조, 구현에 제약사항이 적음**
- **Spring은 POJO를 지향하므로 복잡한 설정 없이 비즈니스 로직에 집중할 수 있음**
- **Spring IoC 컨테이너를 사용하여 웹 프레임워크 연계를 위한 추가 설정 없이 Spring MVC를 사용할 수 있음**



1-2. Spring MVC 개요 – 웹 프레임워크 트랜드 분석

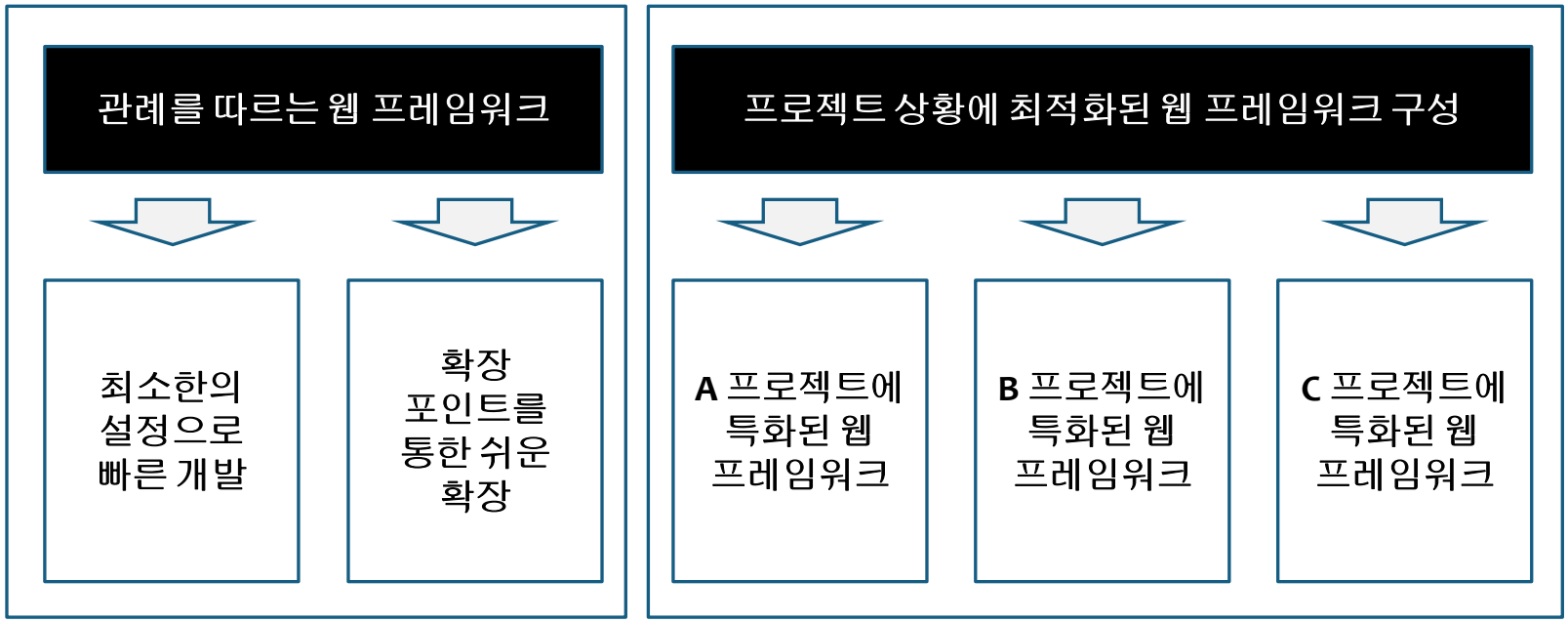
- **웹 MVC 프레임워크에는 Spring MVC, Struts2, Struts1, JSF 등이 있음**
- **이전에는 Struts2가 대표적인 웹 프레임워크였으나, 최근에는 Spring MVC에 관심이 높아지고 있음**
- **Struts2는 프리젠테이션 계층만 지원하는 프레임워크로써, 애플리케이션 개발 시 다른 프레임워크와 연동해야 함**
- **Spring MVC는 Spring 프레임워크의 기능을 활용할 수 있고, 기능 확장이 쉬워 꾸준한 상승세를 보임**





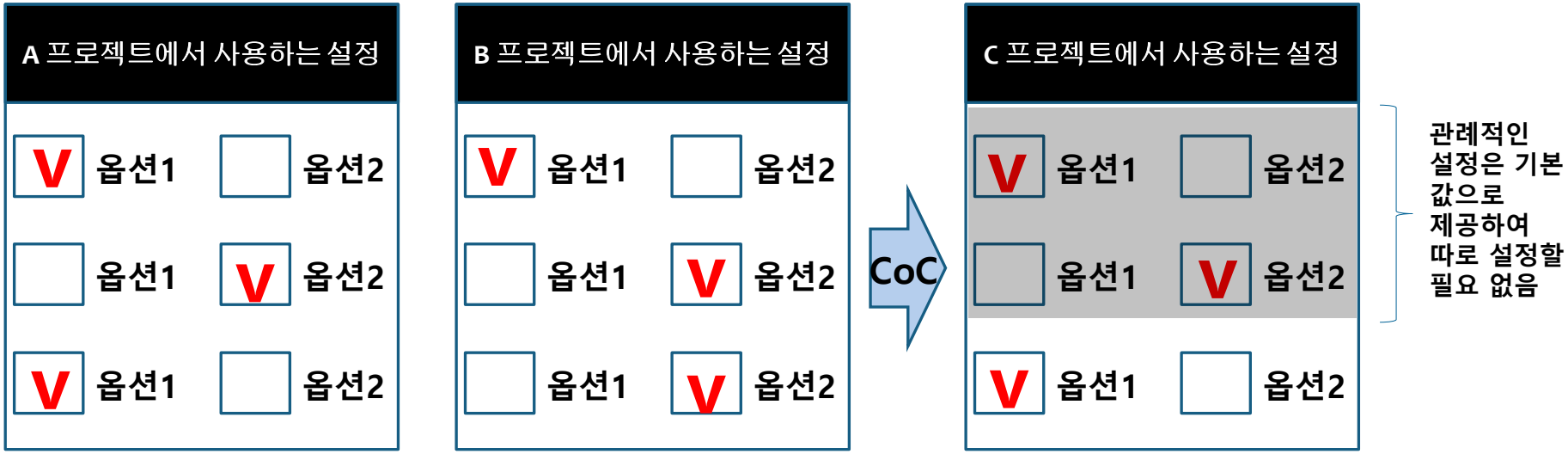
1-3. Spring MVC 특징 – 기술 확장에 관대한 웹 프레임워크

- **프레임워크에서 조합한 기술을 강요하지 않고, 프로젝트에 적합한 구성 가능**
- **Spring MVC에서 제공하는 주요 기능들은 다양한 방법으로 확장 가능**
- **관례를 따르는 기본 설정으로 빠른 개발 가능**
  → 필요한 경우 설정을 위한 확장포인트를 제공
- **Spring MVC를 이용하면 프로젝트에 적합하고 효율적으로 개발할 수 있는 프레임워크를 새롭게 구성 가능**



1-3. Spring MVC 특징 – 설정 보다는 관례(CoC)

- **CoC(Convention over Configuration)는 설정 보다는 관례를 따르는 SW 설계 방식**
- **CoC는 개발자들이 결정할 사항들을 줄여서 단순하고 유연하게 개발할 수 있도록 지원**
- **XML과 같은 설정이 많으면 개발 복잡도가 증가 → 최근 프레임워크들은 어노테이션으로 CoC를 실현**
- **Spring MVC는 공통된 관례들을 제공 → 원하는 설정만 개별적으로 적용하도록 하여 개발의 효율 높임**



1-4. Spring @MVC

- **Spring은 2.5버전부터 어노테이션을 도입하였고, 3.0 버전에서 어노테이션 지원을 강화**
- **어노테이션을 중심으로 한 새로운 MVC를 어노테이션 기반 MVC이라는 의미로 Spring @MVC라고도 함**
- **현재 Spring은 XML로 설정하던 이전 버전의 사용을 권고하지 않으며, 차기 버전에서는 없어질 수도 있음**
- **어노테이션을 추가함으로써 POJO에 가까운 개발이 가능하며, 메소드 단위로 요청을 처리할 수 있음**

[이전 방식] AbstractController 클래스를 상속하여 컨트롤러 작성

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

[이전 방식] Controller 인터페이스를 구현하여 컨트롤러 작성

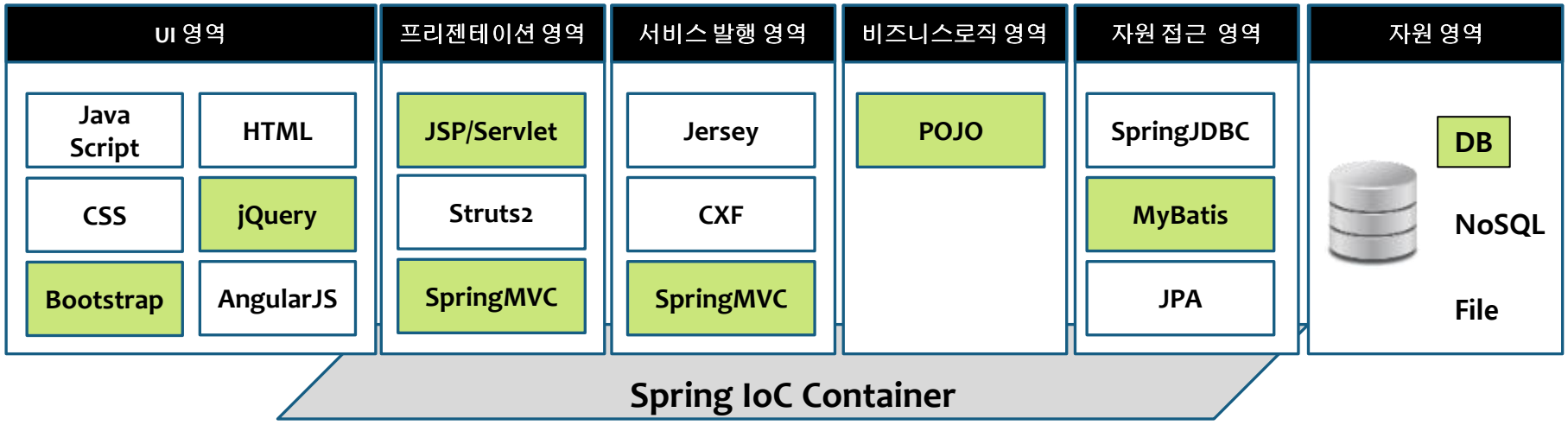
```
public class CookbookReadController implements Controller {
    @Override
    public ModelAndView handleRequest(
            HttpServletRequest req, HttpServletResponse res)
                throws Exception {
        ...
    }
}
```

[Spring @MVC] 어노테이션을 적용한 컨트롤러

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

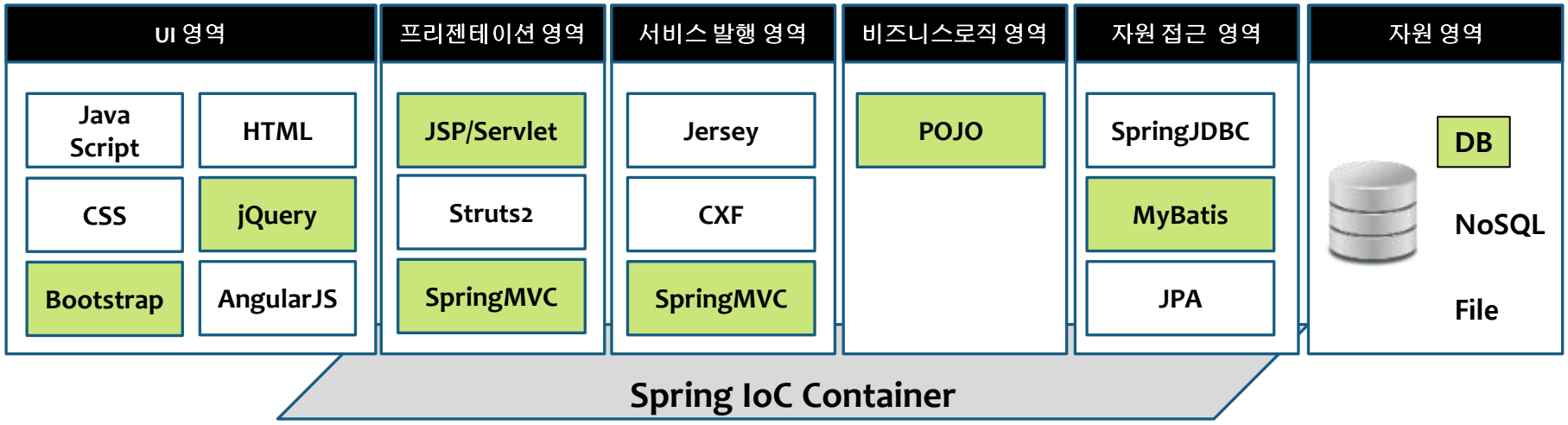
- @Controller가 붙은 클래스는 클라이언트가 요청 시점에 Front Contoller에 의해 호출 됨
- Spring @MVC에서는 어노테이션을 사용하여 인터페이스나 클래스를 상속하지 않고 컨트롤러 클래스를 작성할 수 있음
- @RequestMapping은 XML 설정 없이 요청 URL을 매핑하는 어노테이션

- **실습**
  - **프리젠테이션 영역에서는 Java 웹 개발의 기본이 되는 JSP/Servlet과 함께 SpringMVC 프레임워크 사용**
  - **서비스 발행영역은 SpringMVC의 RESTful 웹 서비스 지원 기능을 사용**
  - **데이터는 관계형 데이터베이스에 저장**
  - **MyBatis를 데이터 접근 프레임워크로 사용**



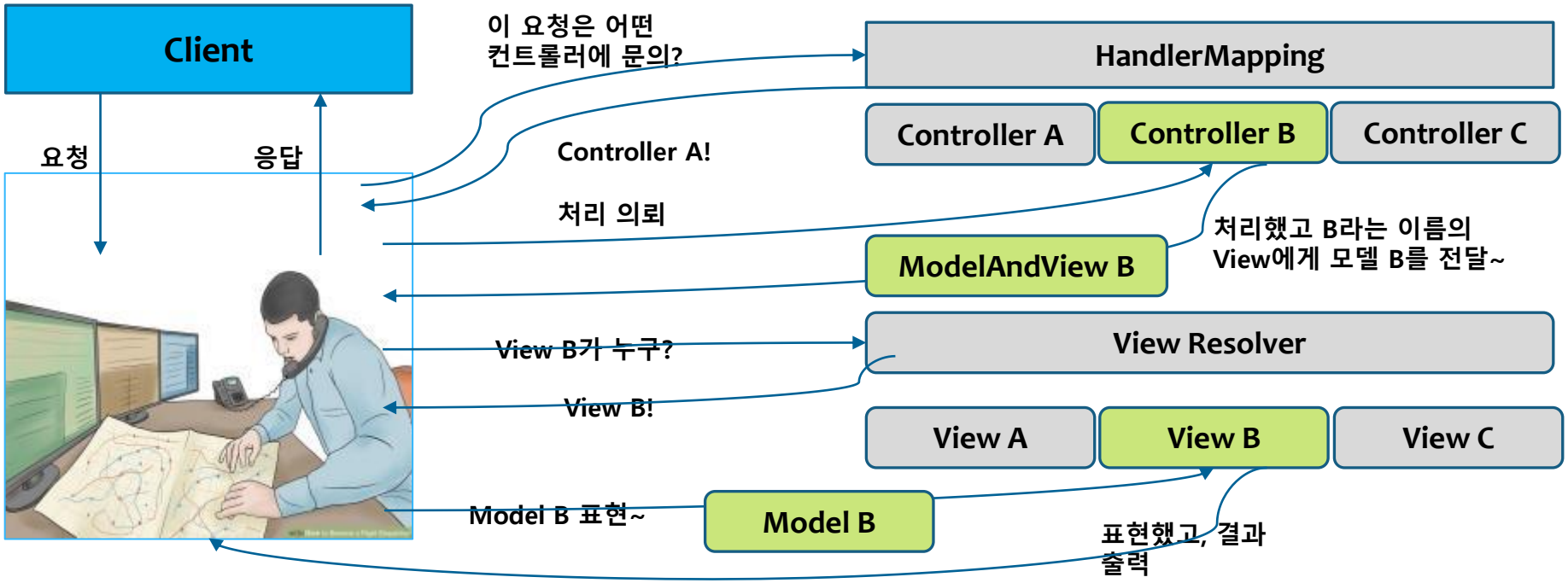
2-2. 실습

- 프리젠테이션 영역에서는 Java 웹 개발의 기본이 되는 JSP/Servlet과 함께 SpringMVC 프레임워크 사용
- 서비스 발행영역은 SpringMVC의 RESTful 웹 서비스 지원 기능을 사용
- 데이터는 관계형 데이터베이스에 저장
- MyBatis를 데이터 접근 프레임워크로 사용



3-1. DispatcherServlet – 요청처리 절차

- **DispatcherServlet은 Spring MVC의 핵심으로써, Front Controller 역할**
- **클라이언트 요청을 받아 응답하는 과정에서 담당할 대상 선택하고 역할을 분배하는 등의 작업 수행**
- **DispatcherServler이 요청을 처리하는 과정을 이해하면 Sping MVC를 잘 활용할 수 있음**
- **웹 애플리케이션 설정파일 (web.xml)에 서블릿 요청을 DispatcherServlet 클래스가 처리하도록 매핑**



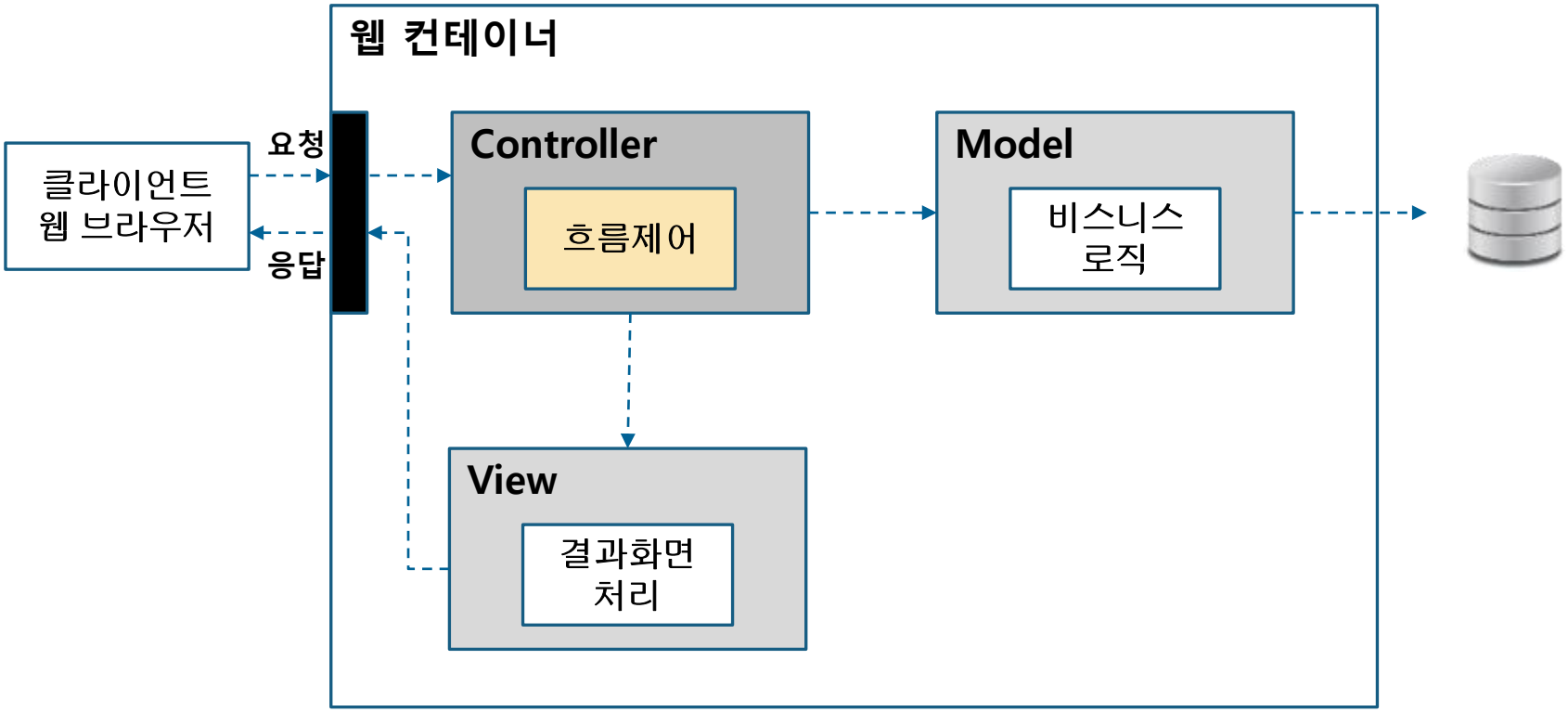
3-1. DispatcherServlet – MVC 클래스

- **개발자는 Controller만 구현하고, 그 외의 객체는 Spring MVC가 제공하는 클래스를 사용**
- **DispatcherServlet은 요청에서부터 응답까지의 전체 라이프사이클을 관리**
- **HandlerMapping이 요청을 처리할 컨트롤러를 결정하는 기준에는 URL, 클래스명, 어노테이션 등이 있음**
- **ModelAndView는 요청처리 결과 데이터와 화면에 표시할 View 이름을 포함**

클래스 명	설명
DispatcherServlet	단일 프론트 컨트롤러로 모든 HTTP 요청을 수신하여 그 밖의 오브젝트 사이의 흐름 제어
HandlerMapping	클라이언트가 요청한 URL을 바탕으로 어느 컨트롤러를 실행할 지 결정 URL, 컨트롤러 클래스명, 어노테이션을 기준으로 결정
Controller	클라이언트 요청에 맞는 프리젠테이션 층의 애플리케이션 처리 실행 요청처리 결과 데이터를 ModelAndView에 반영
ModelAndView	Model은 컨트롤러에서 뷰에 전달할 데이터를 저장하는 객체 ModelAndView는 실제 View의 JSP 정보를 갖고 있지 않으며, ViewResolver가 논리적 이름을 실제 JSP이름으로 변환
ViewResolver	View 이름을 바탕으로 View 객체를 결정
View	화면에 표시되는 객체

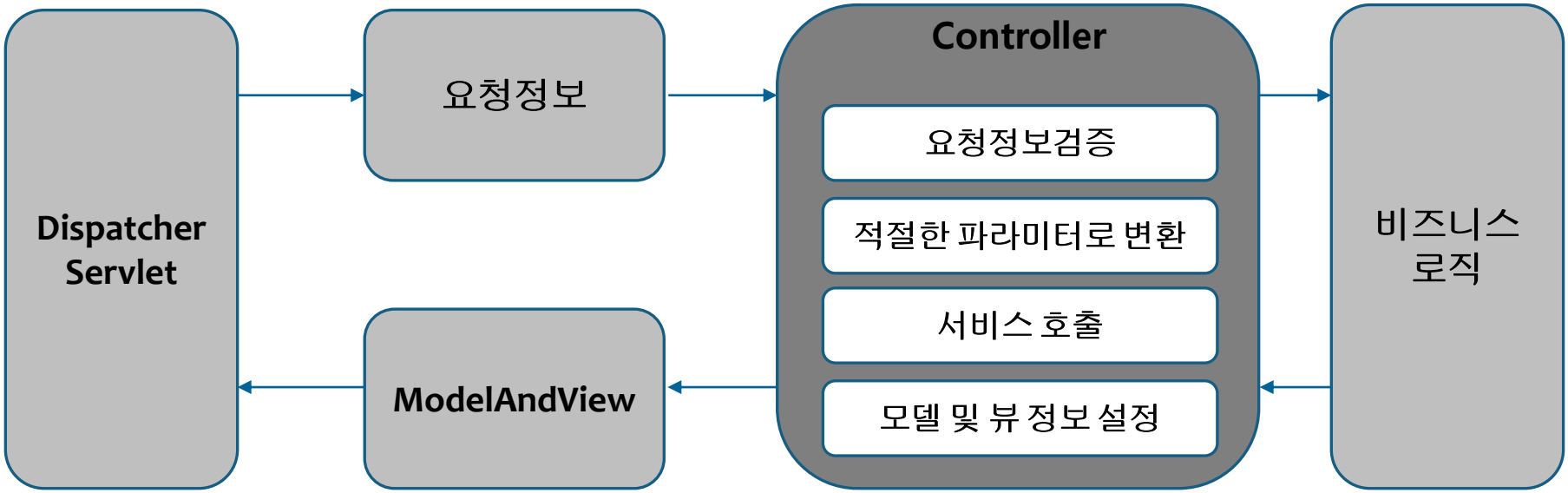
3-2. Controller – 개요

- **Controller는 MVC(Model, View, Controller) 중 가장 많은 작업을 처리**
- **Controller는 클라이언트의 모든 요청을 처리**
- **Controller는 요청 내용을 처리하는 서비스를 호출하여 비즈니스 로직을 처리**
- **Controller는 처리결과를 View에게 전달하여 결과화면을 생성**



3-2. Controller – 개요

- **Spring MVC 컨트롤러는 DispatcherServlet에게 전달받은 요청 정보의 정합성을 검증**
- **서비스에게 비즈니스 로직 처리를 위임 → 이를 위해 적절한 파라미터로 변환하여 서비스에게 전달**
- **서비스로 부터 처리결과를 받으면 어떤 뷰를 보여줘야 할지 결정 → 처리 결과를 뷰에게 전달할 형태로 생성**
- **모델과 뷰를 생성하여 DispatcherServlet에게 전달**



3-2. Controller – @Controller

- **클래스에 @Controller 어노테이션을 붙이면 빈 등록 설정 없이도 컨트로롤러를 빈으로 등록 가능**
  - **→ 어노테이션을 사용하기 위해서는 설정파일에 <component-scan> 요소 추가**
  - **→ <component-scan> 요소를 추가하고 어노테이션을 스캔할 범위를 패키지로 지정**
  - **→ 스캔을 범위를 세밀하게 설정하려면, 하위 요소로 <include-filter> 추가**

```
@Controller
public class HomeController {
    @RequestMapping(value="/", method=RequestMethod.GET)
    public String home(Local local, Model model) {
        Date date = new Date();
        DateFormat dateFormat = DateFormat.getDateTimeInstance(
                                DateFormat.LONG, DateFormat.LONG, locale);

        String formattedDate = dateFormat.format(date);

        model.addAttribute("serverTime", formattedDate);

        return "home";
    }
}
```

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

3-2. Controller – 핸들러 매핑

- **핸들러 매핑은 HTTP 요청정보를 처리하는 컨트롤러(핸들러)를 찾아주는 역할**
- **핸들러 매핑을 여러 개 등록하여 사용하는 경우, order를 이용해 우선 순위 지정 가능**
- **핸들러 매핑 빈의 defaultHandler 프로퍼티에 URL을 매핑하지 못할 때 사용할 디폴트 핸들러를 지정할 수 있음**
- **Spring에서는 5가지의 핸들러 매핑 전략 클래스를 제공**

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

3-2. Controller – 핸들러 매핑

- **BeanNameUrlHandlerMapping은 핸들러 매핑을 등록하지 않으면 사용되는 디폴트 핸들러 매핑 전략**
- **<bean> 요소의 name과 컨트롤러 클래스를 연결**
- **설정은 간편하나, 컨트롤러의 개수만큼 빈을 등록해야 함**
- **<bean> 요소의 name 속성은 매핑할 URL이며, class 속성은 요청을 처리할 컨트롤러 클래스**

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

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

URL로 매핑할 이름 **요청을 처리할 컨트롤러 클래스**

3-2. Controller – 핸들러 매핑

- **ControllerBeanNameHandlerMapping은 빈의 id와 지정한 컨트롤러 클래스를 연결하는 핸들러 매핑 전략**
- **urlPrefix 프로퍼티에 URL에서 공통적으로 나타나는 prefix 설정 가능**
- **@Component 어노테이션에 bean id를 지정한 경우에도 핸들러로 매핑**
- **이 핸들러 매핑 전략은 빈으로 등록해야만 사용할 수 있음**
  → 등록 시 Default Handler 적용되지 않음

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

urlPrefix를 /mvc/로 지정하였으므로, HomeController는 /mvc/hello 요청에 대해 매핑

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

URL로 매핑할 이름 — **요청을 처리할 컨트롤러 클래스**

```
@Component("hello")
public class HomeController implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
                                      HttpServletResponse response)
                    throws Exception {
        return null;
    }
}
```

스테레오타입 어노테이션에 값을 설정하여 매핑 할 URL 지정 가능

3-2. Controller – 핸들러 매핑

- **ControllerClassNameHandlerMapping은 컨트롤러 클래스 이름을 URL과 연결하는 핸들러 매핑 전략**
- **만약 클래스 이름이 Controller로 끝난다면 Controller를 제외하고 매핑**

  ex. HomeController가 있는 경우, /home으로 URL을 매핑

- **Default Handler 매핑이 아니므로 빈으로 등록해야만 사용 가능**

```
<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. Controller – 핸들러 매핑

- **SimpleUrlHandlerMapping는 URL과 컨트롤러 매핑정보를 한 곳에 모아 놓을 수 있는 핸들러 매핑 전략**
- **핸들러 매핑 bean property 내의 URL과 컨트롤러 매핑정보를 연결**
- **모든 URL 매핑정보가 모여있다는 장점이 있는 반면, 컨트롤러 빈 이름을 모두 나열 해야 함**

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

3-2. Controller – 핸들러 매핑

- **DefaultAnnotationHandlerMapping는 어노테이션에 적용되는 Default Handler 매핑 전략**
- **@RequestMapping 어노테이션으로 URL을 매핑하며 최근 가장 보편적으로 사용되는 매핑 방법**
- **URL이 설정파일이 아닌 클래스 파일에 있다는 점에서 편리**
  → URL 이외의 다른 요청정보도 활용 가능
- **메소드에 해당 어노테이션을 붙이면 메소드 단위의 매핑을 지원하므로 컨트롤러의 개수를 줄일 수 있음**

```
@Controller
@RequesetMapping("board")
public class BoardController{
    @RequestMapping("list")
    public String list(Moel model) {
        return "/product/list";
    }

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

3-2. Controller – 요청 매핑

- **DefaultAnnotationHandlerMapping는 @RequestMapping 정보 컨트롤러를 매핑**
- **부모 컨트롤러 클래스에 @RequestMapping를 추가하면 하위 컨트롤러 클래스에도 적용**
  → 하위 클래스에서 @RequestMapping을 재정의하면 상위 클래스의 @RequestMapping은 무시
- **@RequestMapping을 인터페이스에 작성한 경우도 구현 클래스에 동일하게 적용 됨**

```
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";
    }
}
```

ChildController는 상위 클래스인 ParentController에 정의된 @RequestMapping을 그대로 상속
→ /find URL을 입력했을 때, "child find" 문구가 출력

3-2. Controller – 요청 매핑

- **@RequestMapping을 이용하면 URL 이외에 여러 요청정보 활용 가능**
- **value 속성은 매핑할 URL 설정**
- **method 속성은 요청을 매핑할 HTTP METHOD를 설정**
  - **GET, POST, PUT, DELETE**
- **params 속성은 요청 파라미터를 설정**
  - **파라미터 값이 지정한 값과 일치할 때만 요청을 매핑**

```
@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 매핑

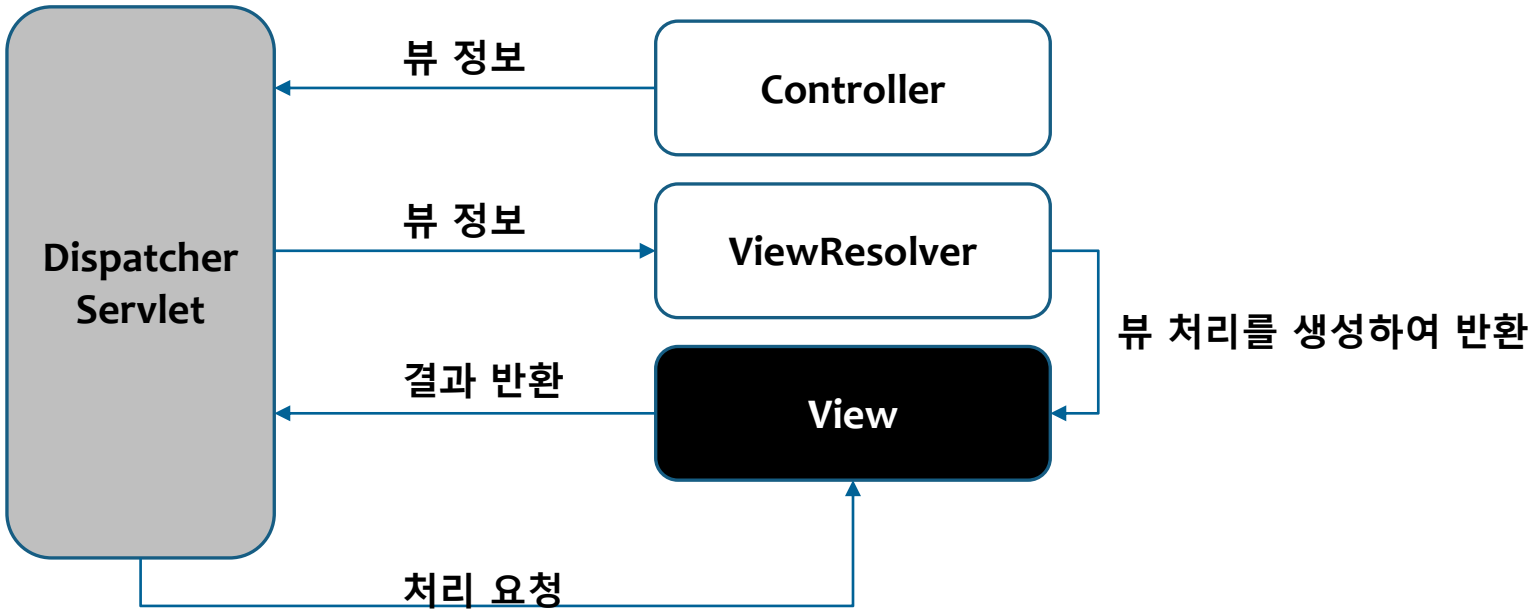
3-3. View – 개요

- MVC에서 뷰(View)는 모델을 전달 받아 모델의 정보를 다양한 형식으로 표현하는 기능을 담당
- 일반적으로 뷰는 HTML로 생성되어 브라우저에 결과를 표현
  - 엑셀/PDF/XML 등의 콘텐츠로 생성 가능
- Spring에서 제공하는 뷰를 사용하여 여러 콘텐츠 작성 가능
- 뷰를 직접 사용하는 대신, 메시지 컨버터를 사용하면 XML, JSON 타입으로 생성 가능

```
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. View – View and Controller

- MVC에서 뷰와 컨트롤러가 어떻게 연결되는지에 대한 이해가 필수
- 컨트롤러가 처리 후 돌아갈 뷰를 지정해 주었다면 DispatcherServlet은 해당 뷰에 처리를 요청
- 명확히 뷰 객체를 지정하지 않고, 뷰 이름만 알려주어도 ViewResolver를 통해 뷰 객체를 생성할 수 있음
- 어떠한 뷰 정보도 알려주지 않을 때, DefaultRequestToViewNameTranslator를 통해 뷰 이름을 설정할 수 있음



3-3. View – View 지정방법

- **뷰를 지정하는 방법**
  - **컨트롤러가 명시적으로 리턴하는 방법**
    : 컨트롤러가 ModelAndVIew나 View객체에 뷰 이름을 설정하거나, 문자열로 뷰 이름을 반환하는 방법
  - **설정에 의해 자동으로 지정하는 방법**
    : 컨트롤러가 Model, Map, void를 반환하는 경우, 설정에 의해 자동으로 뷰를 지정하는 방법
- **뷰가 지정되지 않으면 RequestToViewNameTranslator는 요청 URL로 부터 뷰 이름을 지정**

```java
@Controller
public class BoardController {

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

```java
@Controller
public class BoardController {

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

```java
@Controller
public class BoardController {

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

요청 URL로 부터 뷰 이름이 지정

자동으로 뷰 이름이 지정되는 경우, 지정한 URL에서 맨 앞의 / 기호와 끝에 이는 확장자를 제외한 이름이 뷰의 이름이 됨

3-3. View – ViewResolver

- 컨트롤러가 뷰 이름을 반환하면 ViewResolver를 통해 실제 사용할 뷰를 결정
- 뷰 이름은 실제 존재하는 뷰가 아니므로, ViewResolver를 통해 논리적인 뷰(뷰 이름)를 실질적인 뷰로 바꿔줌
- ViewResolver는 Spring MVC에서 제공하는 ViewResolver 구현체를 등록하여 사용하거나 직접 확장하여 구현 가능
- 특정 ViewResolver를 Bean으로 등록하지 않으면, InternalResourceViewResolver가 자동으로 등록

```
<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>
```

InternalResourceViewResolver에 prefix/suffix 등의 속성을 지정하고 싶을 경우에는 빈으로 등록 해야 함

3-3. View – RedirectView

- 리다이렉트뷰는 실제 뷰를 생성하지 않고, URL만 생성하여 다른 페이지로 리다이렉트
- 주로 컨트롤러에서 기능을 수행 한 후 기존에 존재하는 특정 요청으로 연결하고 싶을 때 사용

ex. 저장 후 등록조회로 연결할 때, 리다이렉트를 사용

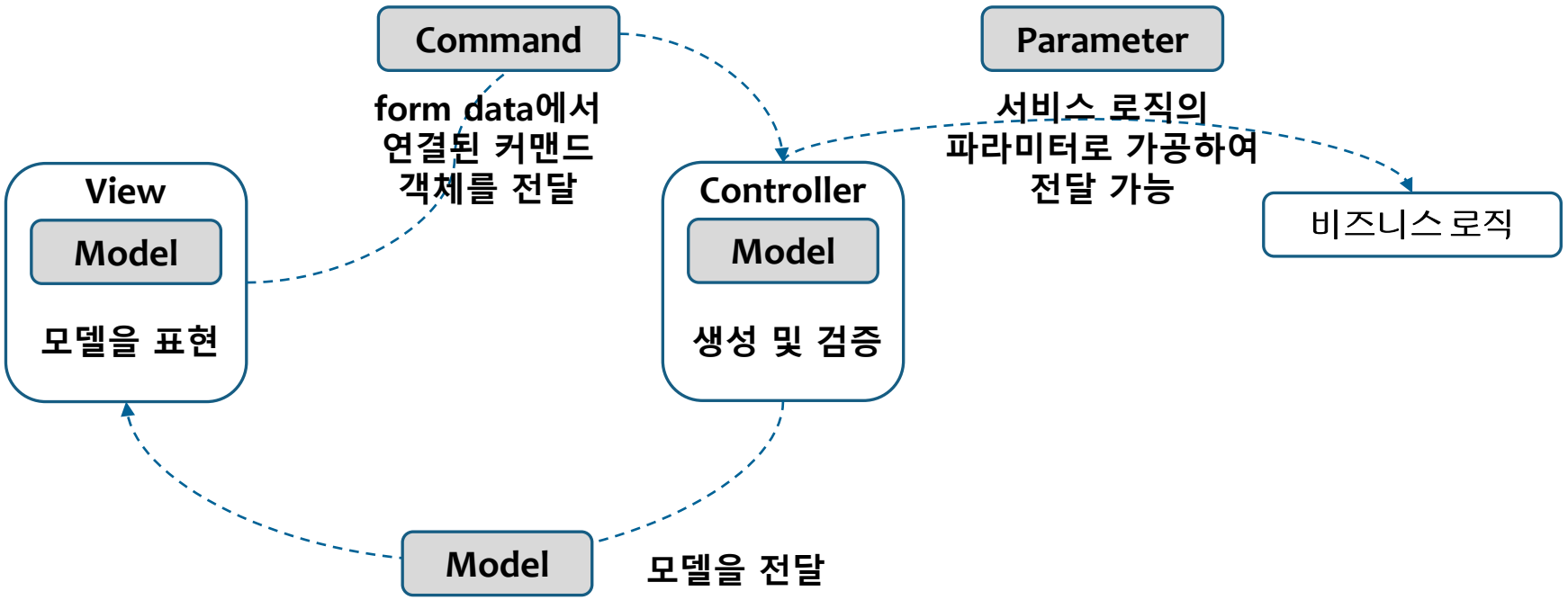
- 리다이렉트뷰는 RedirectView 객체나, 접두어 “redirect:”를 포함한 문자열을 리턴하여 지정

```
// 레시피 등록 후 전체 목록화면으로 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 포시키면 뷰리졸버에서 처리 됨
- UrlBasedViewResolver에는 “redirect:”에 대한 처리가 구현되어 있음
- 기본으로 제공되는 InternalResourceViewResolver는 UrlBasedViewResolver 클래스를 상속받기 때문에 특별한 설정을 하지 않아도 사용할 수 있음

3-4. Model – Spring MVC의 모델

- 컨트롤러는 뷰에서 전달 된 데이터를 커맨드 객체로 전달받아 처리 가능
- 컨트롤러에서 받거나 생성된 모델은 검증과정을 거친 후 비즈니스 로직 메소드에 전달하는 파라미터로 가공 될 수 있음
- 컨트롤러가 모델을 리턴하면 DispatcherServlet는 모델을 뷰에게 전달
- 뷰는 전달받은 모델을 클라이언트에게 적절한 형태로 표현



3-4. Model – Command 객체

- **Spring MVC는 객체 속성과 HTML 폼으로 보낸 데이터의 이름이 같으면 자동으로 매핑에서 전달**
- **컨트롤러는 뷰의 Form으로부터 전달된 데이터를 객체 파라미터로 받아 처리 가능**
- **Command 객체는 모델에 추가하지 않아도 자동으로 모델에 추가되어 뷰에 전달**
- **List 형태로 컨트롤러가 받기 위해서는 반드시 감싸주는 Command 객체가 필요**

```
<form action="posting" method="post">
    <input type="text" name="posting[0].id" />
    <input type="text" name="posting[0].title" />
    <input type="text" name="posting[0].contents" />
    <br/>
    <input type="text" name="posting[1].id" />
    <input type="text" name="posting[1].title" />
    <input type="text" name="posting[2].contents" />
    <br/>
    <input type="submit" />
</form>
```

```
public class PostingCommand {
    private List<Posting> postings;

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

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

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

3-4. Model – Command 객체

- **EL 표기법을 이용하여 컨트롤러에서 받아온 Command 객체를 View에서 사용 가능**
- **@MoveAttribute를 이용하여 이름을 따로 정해주지 않으면 Command 객체의 클래스명을 사용**
- **이름을 변경하고 싶다면, 해당 Command 객체 앞에 @ModelAttribute를 붙여서 이름을 지정**

```
@Controller
public class BoardController {

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

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

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

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

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

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

@ModelAttribute를 파라미터 앞에 붙여서 설정하면, View에서 사용하는 이름을 변경할 수 있음

3-4. Model – @RequestParam, @ModelAttribute

- **@RequestParam은 메소드 파라미터를 요청 파라미터에서 1:1로 받을 경우 사용**
- **@ModelAttribute는 요청 파라미터를 객체 형태로 받기 위해 사용**
- **검색 조건과 같이 여러 파라미터를 객체 형태로 받거나 폼 제출로 넘어오는 파라미터를 바로 객체로 받을 때 유용**
- **@ModelAttribute가 붙은 객체는 자동으로 Model에 추가되므로 뷰에서 바로 사용 가능**

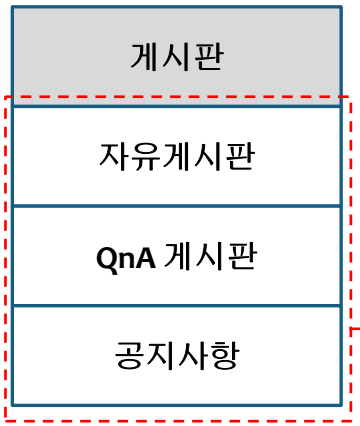
```
@RequestMapping(method=RequestMethod.POST)
public String regist(@ModelAttribute("posting") Posting posting) {
    ...
    return "posting";
}
```

3-5. 정리

- **DispatcheServlet 이란?**
- **Controller의 역할은?**
- **View의 역할은?**
- **Model의 역할은?**

4-1. 게시판 메인화면

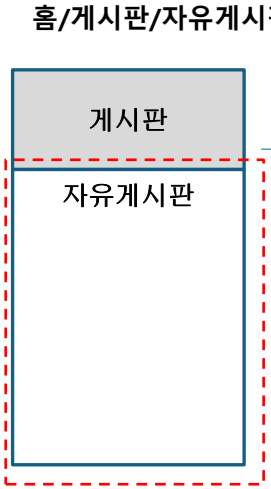
- **게시판 메인화면은 개설된 게시판 목록을 조회하여 보여주는 화면**
- **PostingController 클래스 생성 @Controller 어노테이션 추가**
- **컨트롤러 클래스와 여청 처리 메소드에서 @RequestMapping를 정의**
  → /posting/home이 URL
- **HttpServletRequest 객체 속성으로 boardList 조회결과를 추가**
  → JSP에서 게시판 목록 출력




```java
@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";
    }
    ...
}
```

4-2. 게시판 목록조회

- **게시물 목록조회 화면은 게시판의 게시물 목록을 보여주는 화면**
- **게시물 목록조회 화면을 분석하고 컨트롤러에서 구현해야 할 기능 식별**
  - **게시판 목록 영역을 표현 → 게시판 목록 조회**
  - **선택한 게시판에 대한 게시물 목록 표현 → 게시물 목록 조회**



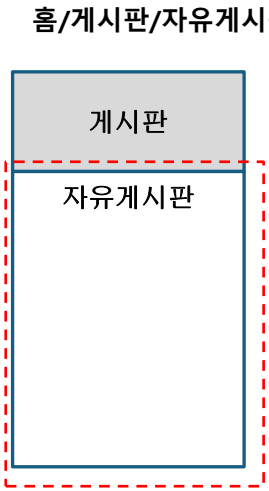
자유게시판

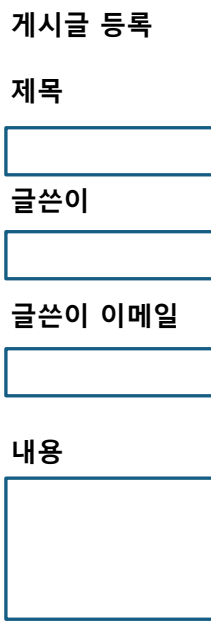
번호	제목	작성일	작성자	조회
001-0001	자유 게시물 오픈	2016-07-12	관리자	0

```
@RequestMapping(value="/list", method=RequestMethod.GET)
public String list(@RequestParam("boardUsid") String boardUsid,
                   @RequesetParam("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 =
        postingServlet.findPostingByCondition(boardUsid, offsetKey);

    // Convert OffsestList ro Page
    Page<Posting> postings = new Page<Posting>(offsetList, pageCriteria);

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

    return "posting/list";
}
```

4-3. 게시판 등록

- **게시물 목록조회 화면에서 [등록] 버튼을 누르면, 게시물 등록화면으로 이동**
- **게시물 등록화면을 분석하여 컨트롤러에서 구현해야 할 기능 식별**
  - **좌측 게시판 목록 영역을 표현하기 위해, 게시판 목록을 조회**
  - **등록할 게시물 내용 입력 후, [확인] 버튼을 누르면 게시물 등록**






```
@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 postinCdo = new PostingCdo(title, writerEmail, contents);
    if (writerEmail != null)
        postingCdo.setWriterName(writerName);

    postingService.registerPosting(boardUsid, postingCdo);

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

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

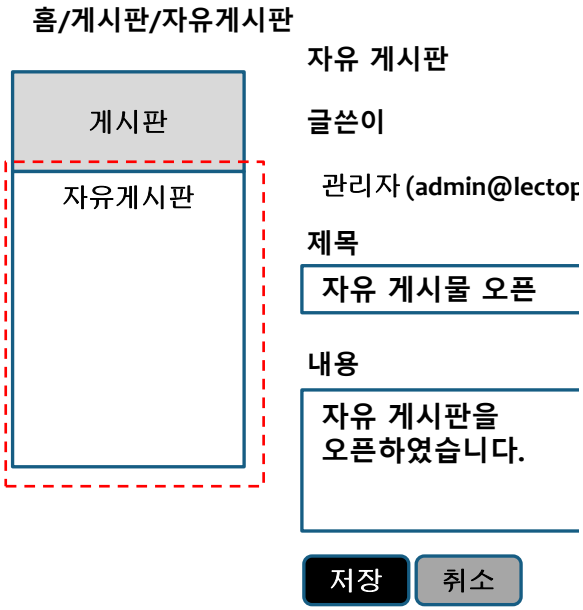
4-4. 게시물 상세조회

- **게시물 상세조회 화면은 특정 게시물에 대한 상세정보를 보여주는 화면**
- **게시물 상세조회 화면을 분석하여 컨트롤러에서 구현해야 할 기능 식별**
  - **좌측 게시판 목록 영역을 표현하기 위해, 게시판 목록을 조회**
  - **선택 게시물의 내용을 표현하기 위해, 게시물 상세정보 조회**


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

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

    return "posting/detail";
}
```

4-5. 게시물 수정 – 화면조회

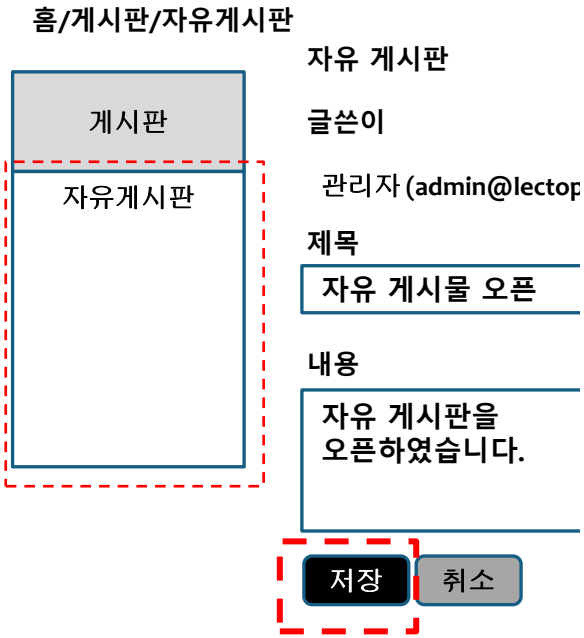
- **게시물 상세조회 화면에서 [수정] 버튼을 클릭하면, 게시물 수정화면으로 이동**
- **게시물 수정 화면은 특정 게시물의 내용을 수정하는 화면**
  - **좌측 게시판 목록 영역을 표현하기 위해, 게시판 목록을 조회**
  - **선택 게시물의 수정 전 내용을 표현하기 위해, 게시물 상세정보 조회**




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

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

    return "posting/update";
}
```

4-5. 게시물 수정 – 수정처리

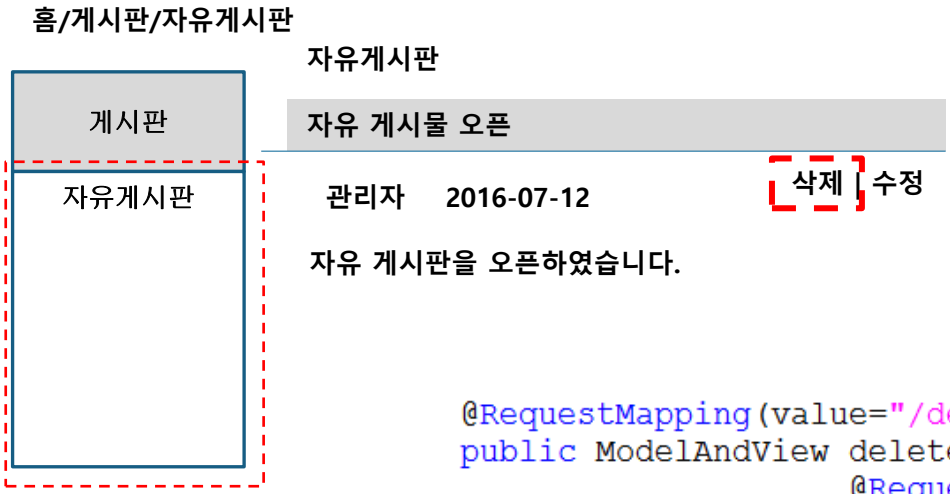
- **게시물 수정 화면에서 내용을 수정 후, [저장] 버튼을 누르면 게시물을 수정**
- **화면에서 파라미터로 전달된 값을 메소드 매개변수로 받기 위해 @RequestParam 어노테이션 적용**




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

    postingService.modifyPosting(boardUsid, nameValues);

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

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

4-6. 게시물 삭제

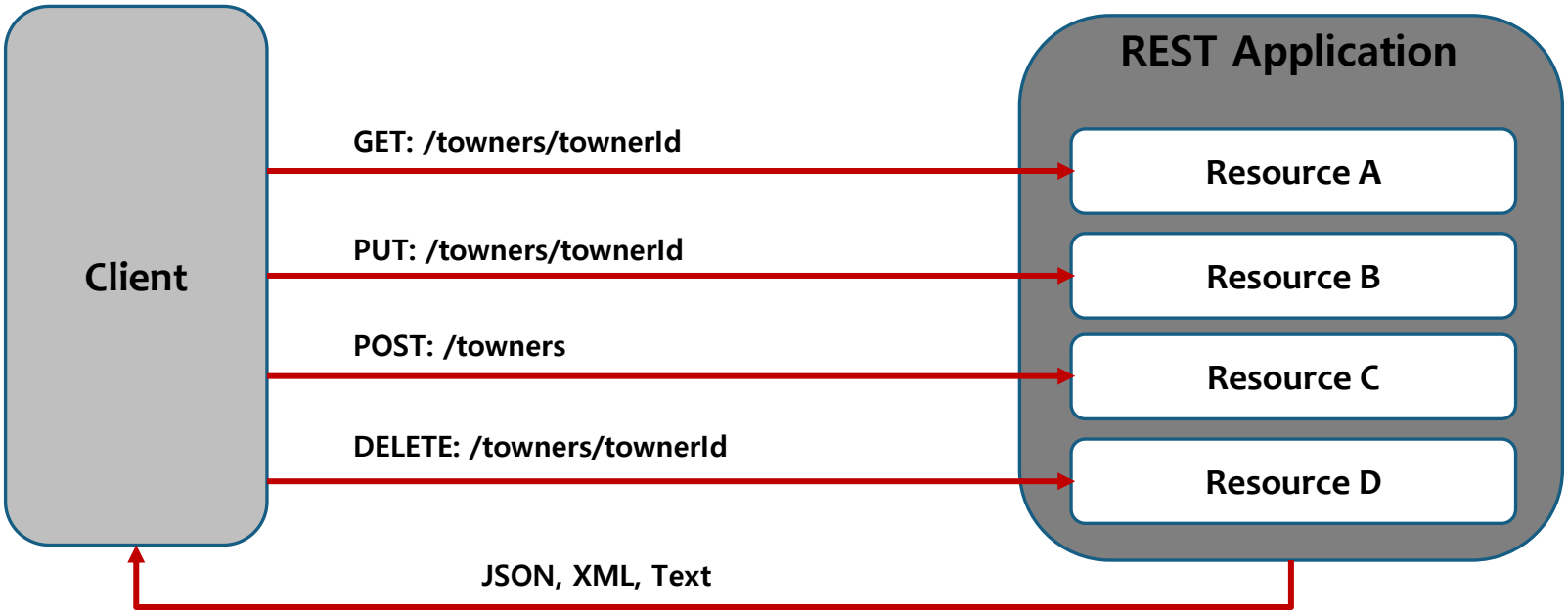
- **게시물 상세조회 화면에서 [삭제] 버튼을 누르면 게시물을 삭제**




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

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

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

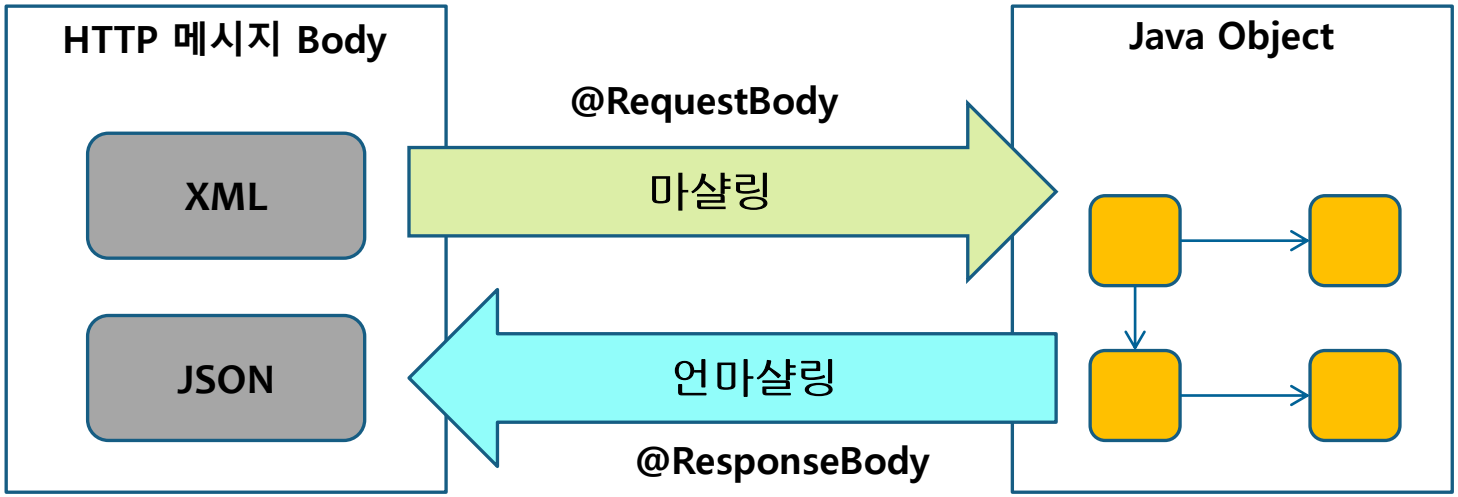
5-1. RESTful 웹 서비스 - 개요

- **REST는 REpresentational State Transfer**
  - **URI를 통해 리소스를 고유하게 식별되게 정의하는 아키텍처 스타일**
- **REST에서는 HTTP Method(GET, POST, PUT, DELETE 등)를 통해 리소스와 상호작용**
- **RESTful 웹 서비스**
  - **REST 스타일을 준수하는 서비스**
  - **응답과 요청에 JSON, XML, 일반텍스트 등을 사용**



5-1. RESTful 웹 서비스 – 메시지 변환

- **웹 시스템 간의 XML, JSON 형식의 데이터를 주고받는 경우 다음과 같은 메시지 변환이 생김**
  - **언마샬링: XML이나 JSON 과 같은 전문을 Java 객체로 변환하는 과정**
  - **마샬링: Java 객체를 XML이나 JSON과 같은 전문으로 변환하는 과정**
- **@RequestBody, @ResponseBody 어노테이션을 메소드에 추가하면 Spring MVC에서 메시지를 변환**



5-1. RESTful 웹 서비스 – @RestController

- **Spring은 RESTful 스타일을 지원하는 @RestController 제공 (Spring 4+)**
- **@RestController 어노테이션이 적용된 모든 컨트롤러는 View 대신 특정 객체 리턴**
- **컨트롤러가 리턴한 객체는 메시지 변환 설정에 따라 변환 됨**
  ex. 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 rest) {
        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 – @PathVariable

- **@RequestMapping에 설정한 URI의 특정 부분에 접근할 때, URI 템플릿을 지정**
- **URI 템플릿은 {, } 안에 변수 이름을 사용해서 표현**
- **@PathVariable은 URI 템플릿 변수 값을 메소드 파라미터에 할당할 때 사용**

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

@Controller
@RequesMapping("/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 – @RequestBody, @ResponseBody

- **HTTP 프로토콜은 header와 body로 구성**
- **@RequestBody는 *HTTP 요청 메시지 Body*를 *자바 객체*로 변환**
- **@ResponseBody는 *자바 객체*를 *HTTP 응답 메시지 Body*로 변환**
- **SpringMVC는 HttpMessageConverter를 사용하여 자바 객체와 요청/응답 Body 사이의 변환 처리**

```
<script type="text/javascript">
$(document).ready(function() {
    $("#getjson").click(function() {
        $.ajax({
            url: "/towners/json",
            method: "post",
            contentType: "application/json",
            success: function() {
                alert("전송 성공");
            },
            error: function(xhr, status, error) {
                alert("Error: " + status);
                alert("Error: " + error);
            }
        });
    });
});
</script>

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

5-1. RESTful 웹 서비스 – @RequestBody, @ResponseBody

- **@RequestBody 어노테이션을 사용하면, 적절한 HttpMethodConverter 구현을 통해 해당 객체로 변환**
  - **요청 body를 @RequestBody 어노테이션이 적용된 자바 객체로 변환할 때에는 HTTP 요청 헤더의 Content-Type 헤더에 명시된 미디어 타입(MIME)을 지원하는 HttpMesageConverter를 구현체로 사용**
  - **@ResponseBody 어노테이션을 이용해서 리턴하는 객체를 HTTP 메시지의 body로 변환할 때에는 HTTP 요청 헤더의 Accept 헤더에 명시된 미디어 타입을 지원하는 HttpMessageConverter 구현체를 선택**

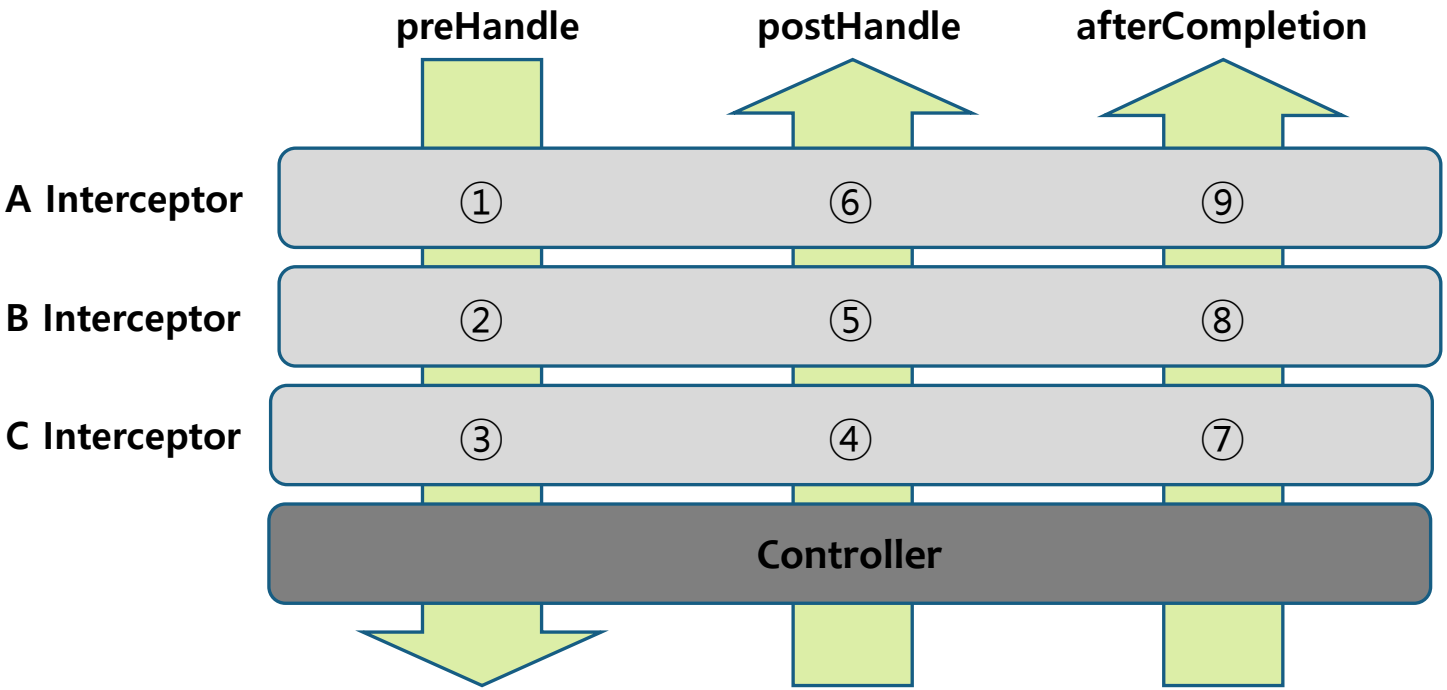
구현 클래스	설명
ByteArrayHttpMessageConverter(*)	HTTP 메시지와 byte 배열 사이의 변환 처리 컨텐츠 타입은 applicaion/octet-stream
StringHttpMessageConverter(*)	HTTP 메시지와 String 사이의 변환 처리 컨텐츠 타입은 text/plain;charset=ISO-8859-1
FormHttpMessageConverter(*)	HTTP 폼 데이터를 MultiValueMap으로 전달 받을 때 사용 컨텐츠 타입은 application/x-www-form-urlencoded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

- **서버에서 제공하는 자원에 대한 REST API 설계**
  - **자원을 나타내는 URI + HTTP Method**
- **@RestController 어노테이션을 적용하여 RESTful 웹 서비스를 제공하는 컨트롤러 클래스를 개발**

기능	자원을 나타내는 URI	HTTP Method
게시판 개설	/ws/board	POST
게시판 수정	/ws/board/{boardUsid}	PUT
게시판 삭제	/ws/board/{boardUsid}	DELETE
게시물 목록 조회	/ws/board/{boardUsid}/postings	GET
게시물 상세 조회	/ws/board/{boardUsid}/postings/{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

5-2. 요청과 응답 가로채기 – 개요

- 인터셉터
  - DispatcherServlet이 컨트롤러를 호출하기 전과 후에 요청과 응답을 참조하여 가공하는 기능
  - 컨트롤러 호출 전과, 호출 후, 뷰 호출 후의 3곳에서 요청과 응답을 가로 챌 수 있음
- 로깅이나 모니터링 정보 수집, 접근제어 처리 등과 같이 여러 컨트롤러에 공통르로 적용할 때 유용
- 인터셉터를 체인형태로 여러 개 적용할 수 있음 (순서주의!)



5-2. 요청과 응답 가로채기 – HandlerInterceptor

- **Spring은 여러 컨트롤러에 공통으로 적용되는 기능을 쉽게 구현할 수 있도록 HandlerInterceptor 제공**
- **HandlerInterceptor는 요청과 응답을 참조하여 가공할 수 있는 필터의 일종**
- **HandlerInterceptor 인터페이스를 구현하거나 HandlerInterceptorAdaptor 클래스를 상속**
- **HandlerInterceptor 인터페이스는 처리 시점에 따른 3가지 메소드 제공**

```
public class LoggingInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest req,
            HttpServletResponse res, Object handle) throws Exception {
        System.out.println("intercept!! prehandle");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest req,
            HttpServletResponse res, Object handle,
            ModelAndView mav) throws Exception {
        System.out.println("intercept!! postHandle");
    }

    @Override
    public void afterComplete(HttpServletRequest req,
            HttpServletResponse res, Object handle,
            Exception ex) throws Exception {
        System.out.println("intercept!! afterComplete");
    }
}
```

컨트롤러가 호출되기 전에 호출
false를 리턴하면 request를 바로 종료

컨트롤러가 실행되고 난 후 호출
prehandle에서 false를 리턴하면 실행되지 않음

모든 작업이 완료된 후에 호출
주로 요청 처리 중에 사용한 자원을 반납

5-2. 요청과 응답 가로채기 – HandlerInterceptor

- **HandlerInterceptor를 Spring 설정파일에 등록하면, 모든 컨트롤러에 적용 됨**
- **컨트롤러 메소드 전/후/응답완료 후 호출되는 것 확인 가능**

servlet-context.xml

```
<interceptor>
    <beans:bean class="com.lectopia.springmvc.cookbook.interceptor.LoggingInterceptor" />
</interceptor>
```

임의의 메소드 실행 결과

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

5-2. 요청과 응답 가로채기 – URL 매핑

- **HandlerInterceptor는 기본적으로 모든 요청에 대해서 적용 됨**
- **특정 요청 URL에 대해서만 인터셉터를 적용하고 싶은 경우, interceptor 요소를 사용 함**
- **HandlerInterceptor 요청 경로 패턴은 mapping 요소를 사용해서 지정**

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

5-3. 예외 처리 – 개요

- **컨트롤러 작업 중에 발생한 예외를 적절히 처리하는 것은 UX 측면에서 매우 중요**
- **별다른 처리를 하지 않으면 브라우저는 HTTP 500 에러와 서블릿 컨테이너가 출력한 에러 페이지를 표시하게 됨**
- **SpringMVC의 HandlerExceptionResolver를 사용해서 예외 발생 시 적절한 안내 페이지를 보여줄 수 있음**
- **SpringMVC는 4개의 HandlerExceptionResolver 구현체를 제공**

구현 클래스	설명
AnnotationMethodHandlerExceptionResolver	예외가 발생한 컨트롤러 내의 메소드 중에서 @ExceptionHandler 어노테이션이 적용된 메소드를 찾아서 예외 처리를 위임해주는 HandlerExceptionResolver
DefaultHandlerExceptionResolver	다른 HandlerExceptionResolver에서 처리하지 못한 예외에 대해서 처리하는 표준 예외처리 핸들러 Spring MVC 내부에서 발생하는 예외 처리
SimpleMappingExceptionResolver	web.xml의 <error-page>와 유사하게 예외 타입 별로 처리할 뷰 이름 지정
ResponseStatusExceptionResolver	예외 발생 시 해당 예외 코드를 클라이언트에게 돌려주는 것이 아니라, 특정 HTTP 응답 상태 코드로 전환하여 의미 있는 HTTP 응답 상태를 리턴

5-3. 예외 처리 – @ExceptionHandler

- **예외 발생 시, @ExceptionHandler 어노테이션이 붙은 메소드가 호출되어 예외를 처리**
- **@ExceptionHandler가 붙은 예외처리 메소드는 @RequestMapping 메소드와 유사하게 구현**
- **예외타입 지정 시, 해당 예외를 포함한 하위 타입의 예외까지 처리**
- **메소드에서 처리한 예외 객체는 뷰 코드에서 Exception 기본 객체를 이용해서 예외 객체에 접근할 수 있음**

```
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" %>
<! DOCTYP html>
<html>
<head>
<title>Insert title</title>
</head>
<body>
    <h3>에러 메시지</h3>
    <p>${exception}</p>
    <p>${exception.message}</p>
</body>
</html>
```

5-3. 예외 처리 – @ControllerAdvice

- **@ControllerAdvice를 활용하여, 동일한 타입의 예외를 하나의 코드에서 공통으로 처리 가능**
- **@ControllerAdvice를 적용한 클래스를 빈으로 등록해야 예외 처리가 가능**
- **예외가 발생하면, 동일 클래스의 @ExceptionHandler 메소드 중에서 발생한 예외처리 가능한 메소드를 검색**
- **처리 가능한 메소드가 없을 경우, @ControllerAdvice 클래스에 위치한 @ExceptionHandler 메소드를 검색**

```
@ControllerAdvice("com.lectopia.board.web.controller.ws")
public class WsExceptionHandler {
    @ExceptionHandler(LectopiaException.class)
    @ResponseBody public ResponseMessage exceptionHandler(LectopiaException.class) {
        //
        return new ResponseMessage(e);
    }
}
```

com.lectopia.board.web.controller.ws 패키지 아래에 있는 컨트롤러 수행 중 예외가 발생하는경우, 이 핸들러 클래스에서 예외 처리

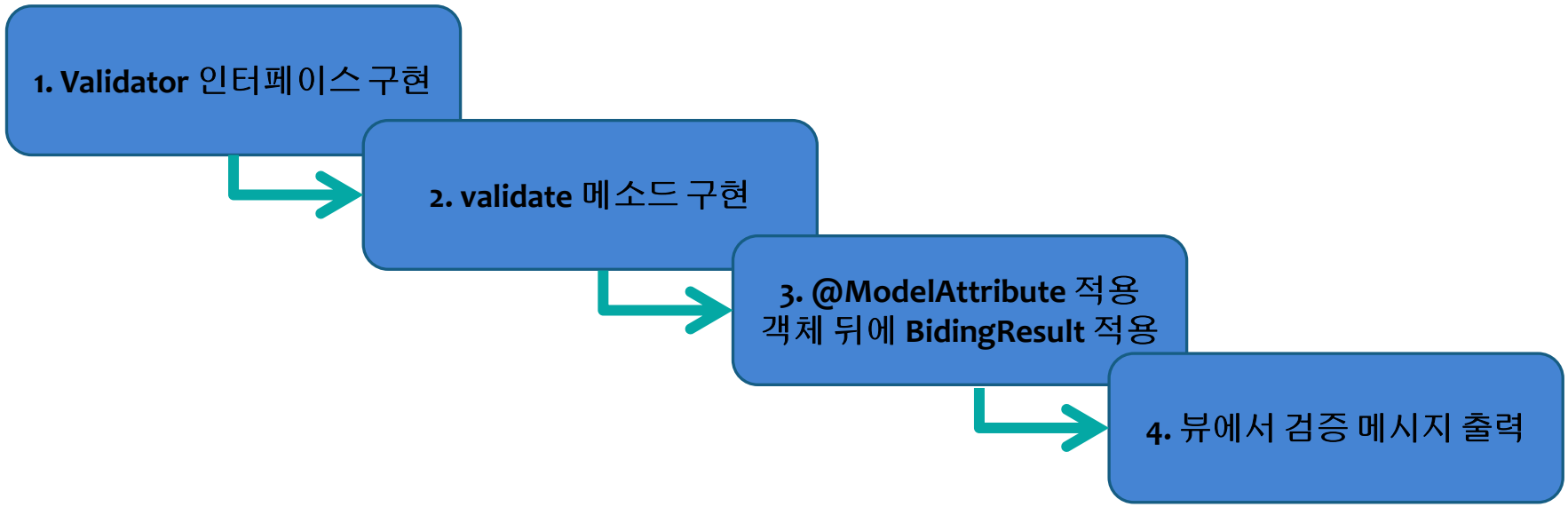
5-3. 예외 처리 – SimpleMappingExceptionResolver

- **SimpleMappingExceptionResolver는 예외 타입별로 에러 페이지를 지정 가능**
- **특정 타입의 예외 발생 시 지정한 에러 페이지를 뷰 페이지로 사용**
- **[실습] com.lectopia.spring.board.web 프로젝트에 LectopiaException 예외가 발생할 때 에러페이지를 보여주도록 추가**
  - **공통 에러페이지는 /views/errors/error.jsp**

```
<beans:bean
    class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <beans:property name="excpetionMapping">
        <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 – 개요

- 화면에서는 서버로 입력 값을 전달하기 전에 JavaScript를 사용하여 1차적으로 오류 검증
- 이와는 별개로 SpringMVC에서 폼 입력 값으로 전달받은 파라미터 값을 검증하는 기능 제공
  - @ModelAttribute로 바인딩 된 객체를 검증하는데 사용
- Spring은 유효성 검사를 위한 Validator 인터페이스와 검사 결과를 저장 할 Errors 인터페이스 제공



5-4. 폼 입력 값 검증 – Validator

- **Validator 인터페이스와 구현 클래스 예시**
- **입력 값 검사 후 통과하지 못한 경우 페이지에 검증 오류 메시지를 보여주도록 구현**
- **supports와 validate 메소드를 overriding하여 구현**

```
public interface Validator {
    // 해당 클래스 validation 지원여부
    boolean supports(Class<?> clazz);

    // 검증 결과 문제가 있는 경우 error 개게에 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
    void validate(Object target, Errors errors) {
        Recipe recipe = (Recipe)target;
        if (recipe.getName() == null) ||
                recipe.getName().length() == 0) {
            errors.reject("recipeName", "레시피 이름은 반드시 입력되어야 합니다.");
        }
    }
}
```

5-4. 폼 입력 값 검증 – Validator → Controller & JSP

- 메소드에 적요할 때, 반드시 검증하려는 객체 바로 뒤에 BidingResult를 추가
- 메소드에서 validate 메소드를 호출해서 입력 값에 대한 검증 수행
- 검증 결과 오류가 발생한 경우
  - bindingResult에 해당 오류가 담겨서 리턴
- 뷰에서는 bindingResult를 받아서 발생한 오류 메시지 화면에 표시

```
...
<body>
    <h1>레시피 등록</h1>
    <form action="${ctx}/recipe" method="post">
        ...
        <input type="submit" value="레시피 등록" />
        <input type="button" value="취소" onclick="javascript:history.back(-1);" />
    </form>

    <c:foreach var="error" items="${errors.allErrors}">
        ${error.defaultMessage} <br/>
    </c:foreach>
</body>
...
```

```
@Controller
public class RecipeController {
    @Autowired
    private RecipeValidator validator;
    ...

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

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

레시피 등록

레시피 명

재료

...

레시피 등록 취소

레시피 이름은 반들시 입력되어야 합니다.

5-5. 요약

- **RESTful 웹 서비스에 필요한 어노테이션은?**
- **여러 컨트롤러에 공통으로 적용되는 기능을 쉽게 구현하기 위한 방법은?**
- **컨트롤러에 발생한 예외를 적절한 형태로 처리 할 수 있는 클래스는?**
- **입력 값 유효성을 검사하기 위해 사용되는 인터페이스는?**