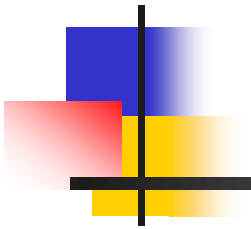
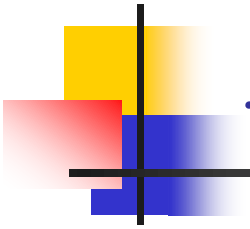


spring 2강-스프링 MVC를 이용한 웹 요청 처리

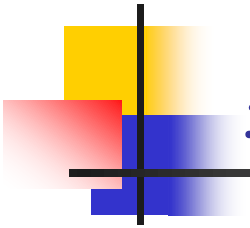
양 명 숙

[now4ever7@gmail.com]



목차

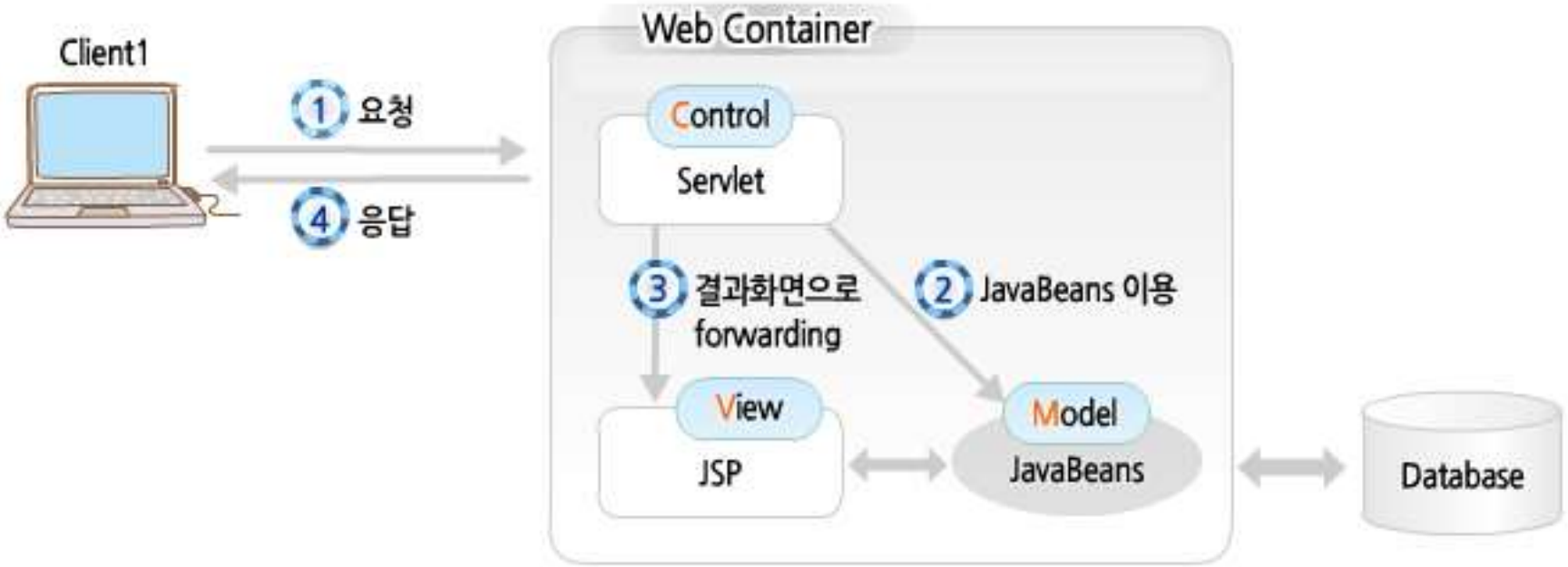
- 스프링 MVC 의 주요 구성 요소 및 처리 흐름
- DispatcherServlet과 ApplicationContext
- 컨트롤러 구현



스프링 MVC

- 스프링 프레임워크는 DI 나 AOP 와 같은 기능뿐만 아니라 웹 개발을 위한 MVC 프레임워크도 함께 제공함
- 스프링 MVC 프레임워크
  - 스프링을 기반으로 하고 있기 때문에 스프링이 제공하는 트랜잭션 처리나 DI, AOP 등을 손쉽게 사용할 수 있음
  - 스트럿츠와 같은 프레임워크와 스프링 프레임워크를 연동하기 위해 추가적인 설정을 하지 않아도 됨
- 모델-뷰-컨트롤러(MVC) 패턴에 기반을 둔 스프링 MVC 를 이용하면, 스프링 프레임워크처럼 **결합도가 낮고 유연한** 웹 기반 애플리케이션을 쉽게 만들 수 있음
- 스프링 MVC 개요
  - 스프링은 디스패처 서블릿, 핸들러 매핑, 컨트롤러, 뷰 리졸버로 요청을 전달함

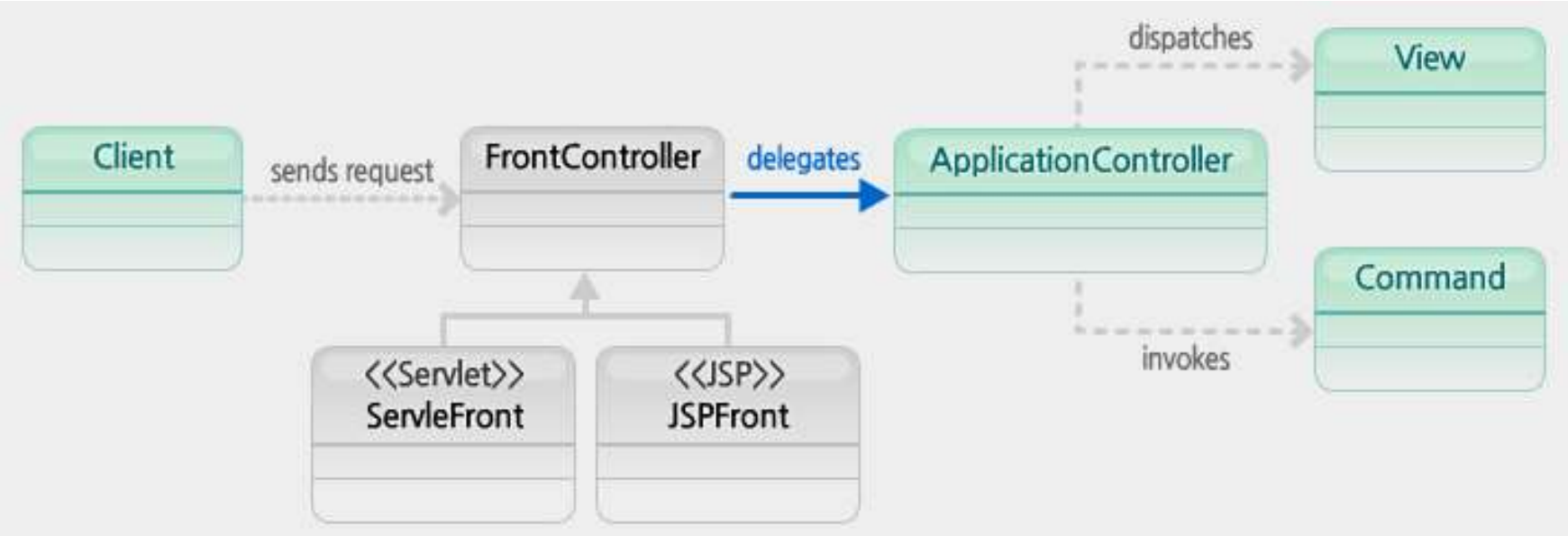
Model2 아키텍처의 처리 흐름



클라이언트의 **모든 요청을 한 곳으로 집중** 시키기 위해서 사용하는 디자인 패턴

FrontController 패턴

- FrontController 패턴
  - Presentation Layer 를 위한 유명한 패턴
  - **클라이언트의 요청을 중앙 집중으로 관리하고자 할 때** 또는 전체 어플리케이션을 통합 관리할 목적으로 적용됨
  - FrontController 는 클라이언트의 요청을 받아서 ApplicationController 로 전달하는 역할을 수행하는데
  - 이런 FrontController 는 Servlet이나 JSP로 구현할 수 있음



Model2 구현

[DispatcherServlet.java]

```
package com.testmall.view.controller;

public class DispatcherServlet extends HttpServlet {
        protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
                //1. URL을 분석하여 요청의 종류를 알아낸다.

                //2. 요청 Path에 따라 적절한 Controller를 찾아서 메소드를 실행한다.

                //3. 적절한 화면으로 이동한다.
        }

        protected void doPost(HttpServletRequest request,
             HttpServletResponse response) throws ServletException, IOException {
                request.setCharacterEncoding("utf-8");
                doGet(request, response);
        }
}
```

- FrontController 기능의 DispatcherServlet
  - 클라이언트의 요청 URL을 분석하여 요청의 종류를 알아냄
  - 요청에 해당하는 적절한 Controller의 메서드를 수행하여 로직을 처리하고, 최종적으로 사용자에게 화면 정보를 응답으로 보내주면 됨

• 앞에서 작성한 Servlet 객체가 클라이언트의 모든 요청에 대해서 동작하도록 하기 위해서는 어떻게 해야 할까?
=> Web Application Deployment Descriptor 설정이 필요함

[web.xml]

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>
            com.testmall.view.controller.DispatcherServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcherServlet</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

        <welcome-file-list>
            <welcome-file>index.jsp</welcome-file>
        </welcome-file-list>
</web-app>
```

DispatcherServlet 이 모든 ".do" 형태의 URL 요청에 대해서 동작하도록 등록한 설정임

Controller interface 정의

- 모든 Command 의 최상위 interface를 Controller 로 정의함
- [Controller.java]

```
package com.testmall.view.controller;

public interface Controller {
        public String handleRequest(HttpServletRequest request,
                HttpServletResponse response) throws Exception;
}
```

구체적인 Controller 클래스들 구현

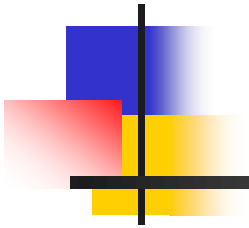
- 모든 Command 의 최상위 인터페이스를 Controller 로 정의한 후, 이 Controller 를 적절하게 구현한 ConcreateCommand에서 사용자 요청 처리 로직을 구현하면 됨

[LoginController.java]

```
package com.testmall.view.system.controller;
public class LoginController implements Controller {
	public String handleRequest(HttpServletRequest request,
			HttpServletResponse response) throws Exception {
	  //1. 사용자 입력정보 추출
	  //2. DB 연동
	  //3. 적절한 화면 URL정보 리턴
	}
}
```

[LogoutController.java]

```
package com.testmall.view.system.controller;
public class LogoutController implements Controller {
	public String handleRequest(HttpServletRequest request,
			HttpServletResponse response) throws Exception {
	  //1. 사용자 입력정보 추출
	  //2. DB 연동
	  //3. 적절한 화면 URL정보 리턴
	}
}
```



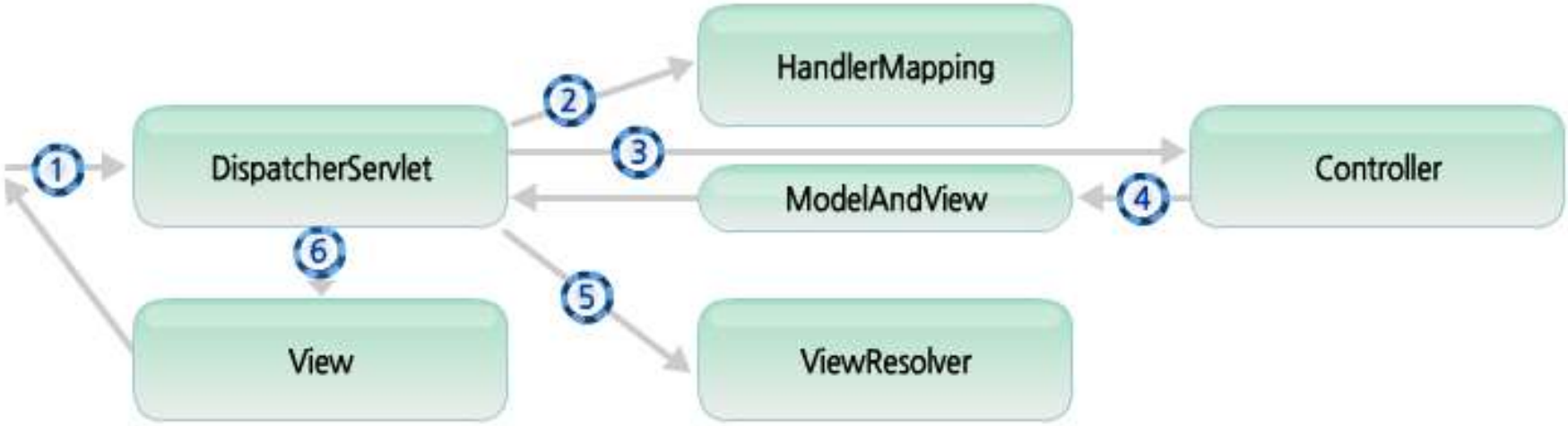
Spring MVC

스프링 MVC의 주요 구성 요소 및 처리 흐름

- 다른 MVC 기반의 프레임워크와 마찬가지로 스프링 MVC도 컨트롤러를 사용하여 클라이언트의 요청을 처리함
- 스프링에서 DispatcherServlet 이 MVC에서 Controller 부분을 처리함
- 스프링 MVC의 주요 구성 요소

구성 요소	설 명
DispatcherServlet	클라이언트의 요청을 전달받는다. 컨트롤러에게 클라이언트의 요청을 전달하고, 컨트롤러가 리턴한 결과값을 View에게 전달하여 알맞은 응답을 생성하도록 함
HandlerMapping	클라이언트의 요청 URL을 어떤 컨트롤러가 처리할지를 결정함
Controller	클라이언트의 요청을 처리한 뒤, 그 결과를 DispatcherServlet 에 알려 줌. 스트럿츠의 Action과 동일한 역할을 수행
ModelAndView	컨트롤러가 처리한 결과 정보 및 뷰 선택에 필요한 정보를 담는다
ViewResolver	컨트롤러의 처리 결과를 생성할 뷰를 결정함
View	컨트롤러의 처리 결과 화면을 생성함. JSP 등을 뷰로 사용함

Spring MVC의 처리 흐름



- DispatcherSevlet을 시작으로 정의된 HandlerMapping 에 의해 Controller를 호출하고, ViewResolver에 의해 View를 찾아 화면을 보여주는 구조임

스프링 MVC의 주요 구성 요소 및 처리 흐름

- 스프링 MVC의 클라이언트 요청 처리 과정

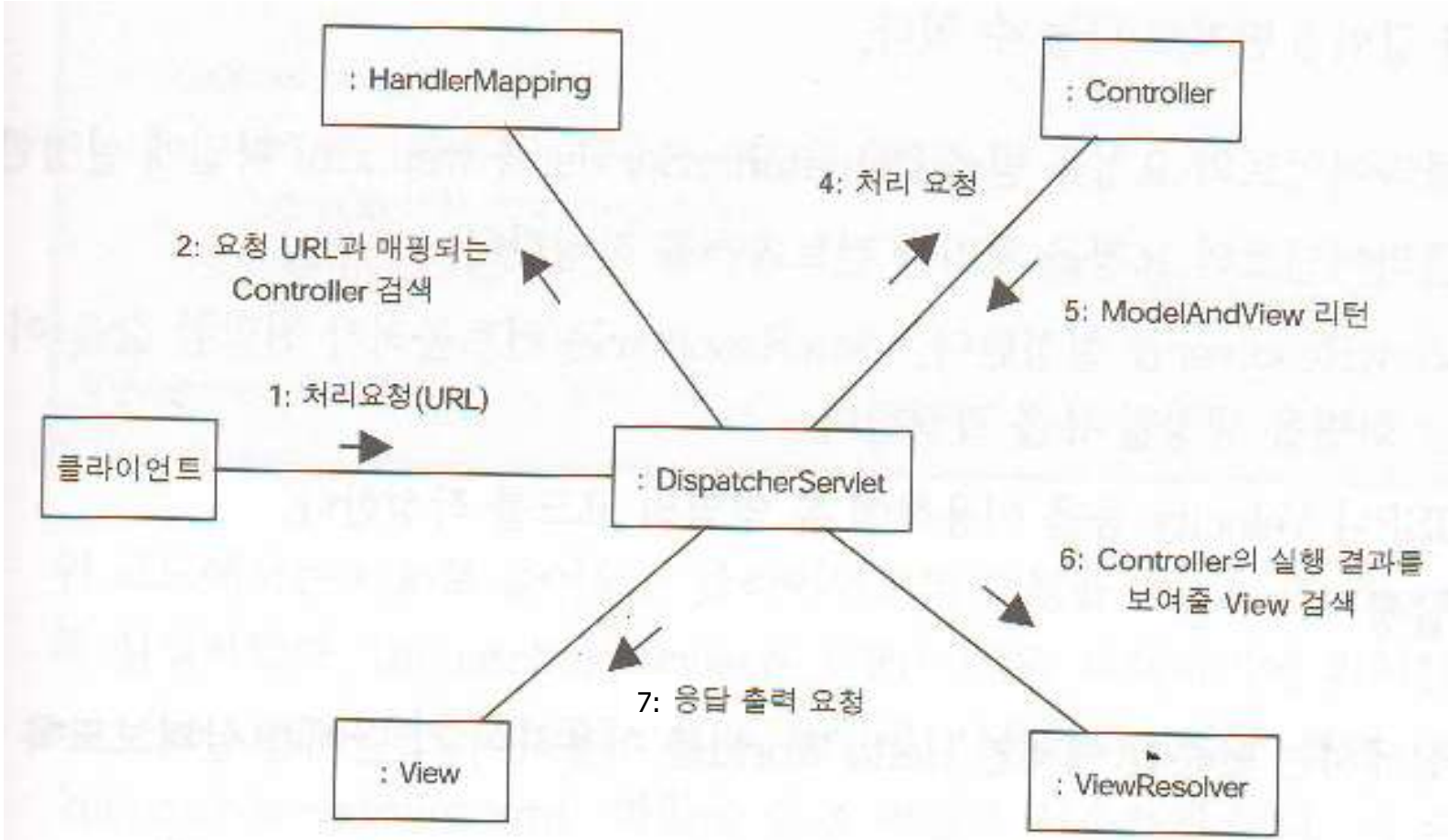
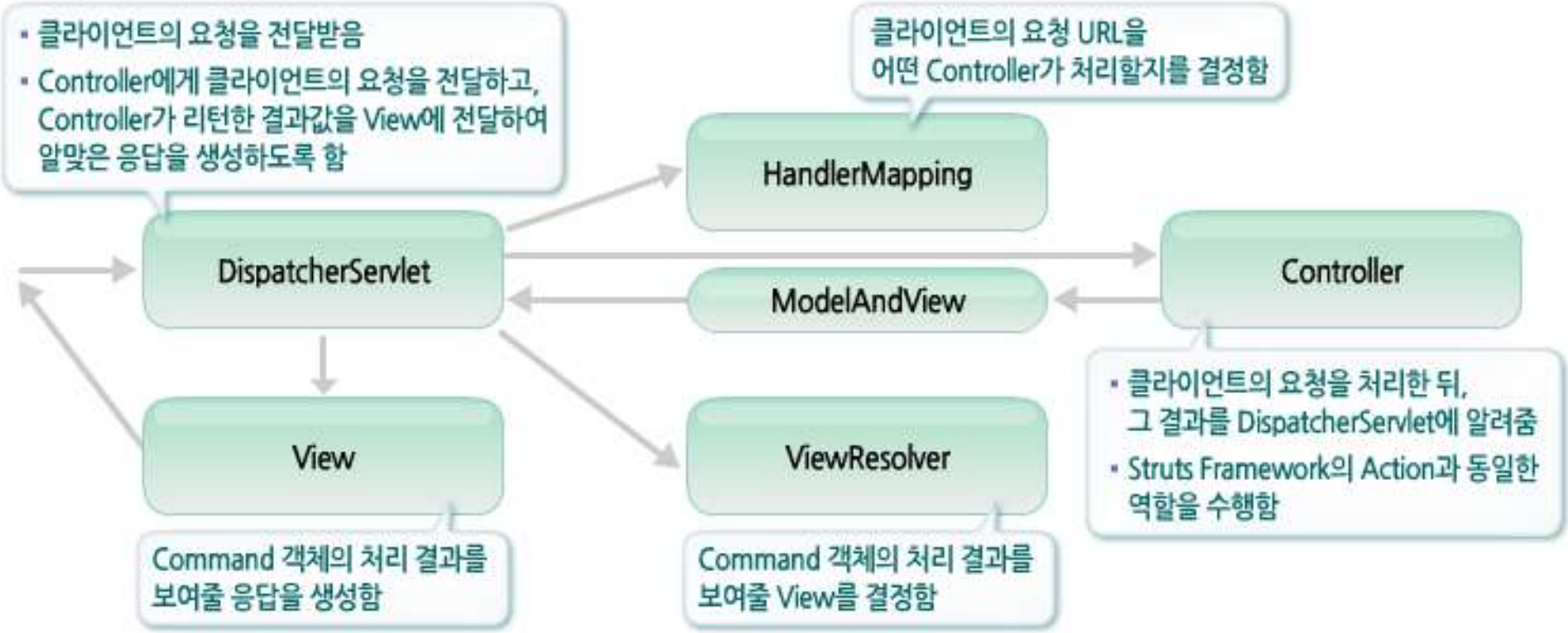


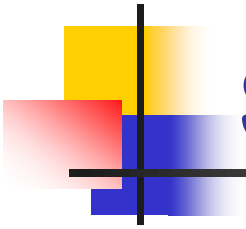
그림 6.1 스프링 MVC의 클라이언트 요청 처리 과정

Spring MVC의 주요 구성 요소



스프링 MVC의 주요 구성 요소 및 처리 흐름

- 1. 클라이언트의 요청이 DispatcherServlet 에 전달됨
- 2. DispatcherServlet 은 HandlerMapping 을 사용하여 클라이언트의 요청을 처리할 컨트롤러 객체를 구함
- 3. DispatcherServlet 은 컨트롤러 객체를 이용해서 클라이언트의 요청을 처리함
- 4. 컨트롤러는 클라이언트의 요청 처리 결과 정보를 담은 ModelAndView 객체를 리턴함
- 5. DispatcherServlet 은 ViewResolver 로부터 응답 결과를 생성할 뷰 객체를 구함
- 6. 뷰는 클라이언트에 전송할 응답을 생성함
- => 개발자가 직접 개발해야 할 부분 – 클라이언트의 요청을 처리할 **컨트롤러**와 클라이언트에 응답 결과 화면을 전송할 **jsp 등의 뷰** 코드임
- 나머지, DispatcherServlet 이나 HandlerMapping, ViewResolver 등은 스프링이 기본적으로 제공하는 구현 클래스를 사용



Spring MVC 적용 절차

- [1] web.xml 파일에 DispatcherServlet 등록
- [2] 클라이언트의 요청에 대한 Controller 작성
- [3] Spring 설정 파일에 HandlerMapping, Controller, ViewResolver 등록
- [4] jsp 작성

[1] web.xml 파일 수정하기

- 1.1 web.xml 파일에 DispatcherServlet 등록

```
[web.xml]
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
<display-name>SpringMVC</display-name>
        <servlet>
                <servlet-name>dispatcher</servlet-name>
                <servlet-class>
                  org.springframework.web.servlet.DispatcherServlet</servlet-class>
        </servlet>

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

•org.springframework.web.servlet 패키지에서 제공하는 DispatcherServlet을 등록하는 설정
•DispatcherServlet 은 클라이언트의 모든 ".do" 요청에 대해서 동작하도록 설정함

[1] web.xml 파일 수정하기

- 1.2 web.xml 파일에 Spring MVC 설정파일 등록

[web.xml]

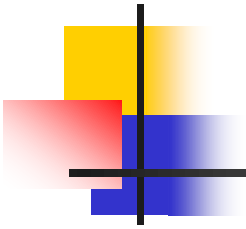
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
<display-name>SpringMVC</display-name>
  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>
       org.springframework.web.servlet.DispatcherServlet   <---java.main 대신 dispatcherservlet이 메인 역을 해준다
    </servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/config/presentation/spring-mvc.xml</param-value>
    </init-param>
 </servlet>
 <servlet-mapping>
      <servlet-name>dispatcher</servlet-name>
      <url-pattern>*.do</url-pattern>
 </servlet-mapping>
</web-app>
```

• <servlet>태그에 <init-param> 태그를 이용해서 DispatcherServlet 객체에게 초기화 파라미터 정보를 알려줌
• 이때 사용하는 파라미터가 **contextConfigLocation** 이며, 이 파라미터에 Spring MVC 설정 파일의 위치를 지정해주면 됨

• DispatcherServlet 객체는 자신이 이용해야 하는 HandlerMapping, Controller, ViewResolver 객체들을 인지하고 있어야 클라이언트 요청에 대해서 적절히 처리할 수 있음

[1] web.xml 파일 수정하기

- <init-param> 태그를 이용해서 초기화 파라미터 정보를 알려주지 않은 경우는?
  - 기본적으로 WEB-INF 폴더에 있는 [servlet-name]-servlet.xml 파일을 Spring MVC 설정 파일로 인식하게 됨
  - 위 경우<init-param> 태그를 생략하면 기본으로 WEB-INF/dispatcher-servlet.xml 파일을 검색하게 되며, 이렇게 로딩된 설정 파일에 모든 Spring MVC 관련 클래스들을 Bean으로 등록해야 함



[1] web.xml 파일 수정하기

- 1.3 web.xml 파일에 인코딩 설정
  - 사용자 입력정보에 한글이 포함되어 있으면 한글을 인코딩해야 하는데, request.setCharacterEncoding("utf-8"); 메서드를 사용하면 간단하게 한글 인코딩을 처리할 수 있다
  - 한글 인코딩 관련 코드가 모든 Controller 클래스마다 존재하는 경우 처리방법
    - web.xml에서 Spring MVC가 제공해주는 Encoding Filter를 사용하면 간단하게 한글을 처리할 수 있음

[1] web.xml 파일 수정하기

- 1.3 web.xml 파일에 인코딩 설정

[web.xml]

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
<display-name>SpringMVC</display-name>
  <filter>
    <filter-name>Encoding Filter</filter-name>
    <filter-class>
       org.springframework.web.filter.CharacterEncodingFilter
    </filter-class>
    <init-param>
       <param-name>encoding</param-name>
       <param-value>utf-8</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>Encoding Filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>
```

• CharacterEncodingFilter 를 등록하고 초기 파라미터 정보로 encoding 파라미터를 "utf-8"로 설정하면, 모든 한글 인코딩은 utf-8 로 설정됨

• <filter-mapping> 에서 <url-pattern> 을 "/*" 로 설정했기 때문에 모든 클라이언트의 요청에 포함된 한글을 인코딩하게 됨

[2] Controller 클래스 작성하기

- Spring MVC에서 제공하는 Controller 인터페이스를 implements 하여 사용자의 요청을 처리하는 로직을 구현함

[LoginController.java]

```
public class LoginController implements Controller {
    public ModelAndView handleRequest(HttpServletRequest request,
                                      HttpServletResponse response) throws Exception {
        //1. 사용자 입력정보(id, password)추출
        String id = request.getParameter("id");
        String pwd = request.getParameter("password");

        //2. DB 연동
        UserDAO dao = new UserDAO();
        UserVO user = dao.getUser(id, pwd);

        //3. 적절한 화면으로 이동
        ModelAndView mav = new ModelAndView();
        if(user != null){
            mav.addObject("message", "로그인 성공");
            mav.addObject("userVo", user);
            mav.setViewName("loginResult");
        }else{
            mav.addObject("message", "로그인 실패");
            mav.setViewName("login");
        }
        return mav;
```

•ModelAndView 객체는 Model 정보와 View 정보를 담고 있는 객체임
-"loginResult" 라는 View 에서 UserVO 객체를 사용할 수 있도록 ModelAndView 객체에 저장함
• **handleRequest()** 메서드는 기본적인 요청 처리를 위해 HttpServletRequest, HttpServletResponse 객체를 매개변수로 받아들임
- 로직 수행의 결과를 ModelAndView 객체에 담아서 리턴함

[2] Controller 클래스 작성하기

- 컨트롤러가 특정 뷰에 종속되지 않도록 하기 위해서 컨트롤러는 직접 특정 JSP를 식별하지 않고, 뷰 이름을 DispatcherServlet으로 돌려보냄
- 뷰 이름은 결과를 만들어 낼 실제 뷰를 찾는데 사용되는 **논리적 이름만 전달**할 뿐이다
- DispatcherServlet 은 뷰 리졸버(view resolver)에게 논리적 뷰 이름을 실제 뷰 구현체에 매핑하도록 요청함

Controller 클래스

```
public class HelloController implements Controller {
    public ModelAndView handleRequest(HttpServletRequest request,
                                      HttpServletResponse response) throws Exception {
                        //1. 파라미터 읽어오기
                        //2. db작업 - 비즈니스 로직 처리
                        String result = getGreeting();

                        //3. 결과, 뷰페이지 저장
                        ModelAndView mav = new ModelAndView();
                        mav.setViewName("hello"); //뷰페이지
                        mav.addObject("greeting", result); //결과 저장

                        return mav;
            }
            private String getGreeting(){
                        Calendar cal = Calendar.getInstance();
                        int hour = cal.get(Calendar.HOUR_OF_DAY);
                        String result="";
                        if(hour>=6 && hour<=10){
                                    result="좋은 아침입니다!";
                        }else if(hour>=12 && hour<=15){
                                    result="점심식사는 하셨나요?";
                        }else if(hour>=18 && hour<=22){
                                    result="좋은 밤 되세요";
                        }else{
                                    result="안녕하세요";
                        }
                        return result;
            }
}//class
```

[3] Spring MVC 설정 파일 작성

- Spring MVC 설정 파일인 spring-mvc.xml 파일에 HandlerMapping, Controller, ViewResolver 를 등록함

[WEB-INF/spring-mvc.xml]

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <!-- HandlerMapping -->
    <bean id="handlerMapping"
            class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
            <property name="mappings">
                    <props>
                            <prop key="/hello.do">helloController</prop>
                    </props>
            </property>
    </bean>

    <!-- Controller -->
    <bean id="helloController"
            class="com.mymall.test.controller.HelloController"></bean>
```

[3] Spring MVC 설정 파일 작성

- <!-- HandlerMapping -->
  - SimpleUrlHandlerMapping 클래스는 클라이언트의 요청 URL과 매핑되는 Controller 를 찾음
  - "/hello.do" 라는 요청에 대해서 "helloController" 라는 이름의 Controller 가 요청을 처리한다
- <!-- Controller -->
  - 앞서 작성한 HelloController를 Bean 으로 등록하면 됨
  - HandlerMapping에 설정한 Controller 이름과 동일한 id 속성값으로 등록해야 함

[3] Spring MVC 설정 파일 작성

```
<!-- ViewResolver -->
<bean id="viewResolver"   <--뷰 이름
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix">
        <value>/WEB-INF/view/</value>    <--ex) voard/detail 등을 넣는다
    </property>
    <property name="suffix">
        <value>.jsp</value>   <--위의 value 두ㅣ에 붙음
    </property>
</bean>
</beans>
```

- <!-- ViewResolver -->
  - 일반적으로 jsp 를 view 로 사용할 경우, InternalResourceViewResolver를 사용함
  - InternalResourceViewResolver 는 접두사(prefix)와 접미사(suffix)를 적절히 지정하면 이 설정을 이용해서 실행할 View 정보를 완성하게 됨
  - 예) ModelAndView 에서 View 이름을 "loginResult" 라고 설정했다면, InternalResourceViewResolver 에 의해 완성된 View 정보는 "/WEB-INF/view/loginResult.jsp" 가 됨

[4] jsp 작성

- view에 해당하는 jsp는 일반적인 jsp 와 동일하게 구현함
- ModelAndView 에 "greeting" 라는 이름으로 등록된 정보를 사용하는 jsp 페이지를 구현함

[loginResult.jsp]

```
<%@page import="com.testmall.biz.user.vo.UserVO"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>로그인 결과</title>
</head>
<body>
<center>
<h3>로그인 결과</h3>
<hr>
메시지 : ${greeting}
<hr>
</center>
</body>
</html>
```

ModelAndView 에 설정된 Model 정보를 "${...}" 과 같은 EL 구문을 이용해서 사용할 수 있음

SpringMVC Framework의 Controller 클래스

- SpringMVC를 이용하면서 Controller 클래스를 개발해야 하는 건 필수임
  - => 기본으로 사용자의 요청 하나당 하나의 Controller 클래스가 매핑되므로 다수의 Controller 클래스를 작성해야 함
  - SpringMVC에서의 Controller 클래스는 Spring에서 제공되는 Controller 인터페이스를 구현한 클래스를 의미함

```
[Controller.java]
public interface Controller {
    ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception;
}
```

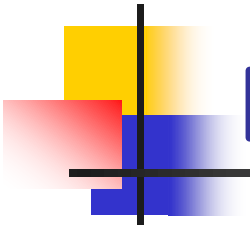
  - Controller 인터페이스를 구현하고, handleRequest()메서드를 오버라이드하여 각각의 Controller 클래스를 구현하면 됨
    - Model과의 연결 부분을 구현하여 클라이언트 요청에 의해 적절한 로직이 실행될 수 있도록 함
  - handleRequest()메서드는 컨테이너에 의해서 자동으로 호출됨

handleRequest()메서드를 오버라이드

[LoginController.java]

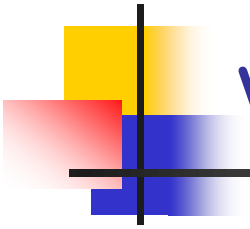
```
public class LoginController implements Controller {
    public ModelAndView handleRequest(HttpServletRequest request,
                                      HttpServletResponse response) throws Exception {
        System.out.println("로그인 처리...");
        return null;
    }
}
```

handleRequest()메서드를 오버라이드하여 로그인 처리 로직을 구현함



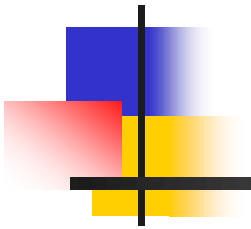
HandlerMapping

- 클라이언트의 요청을 SpringMVC에서 처리할 Controller를 찾을 때 DispatcherServlet은 HandlerMapping에 이를 의뢰함
- SpringMVC에서는 몇 개의 HandlerMapping 클래스를 제공해줌
  - 별도의 설정 없이 사용하면 기본으로 적용되는 클래스
  - BeanNameUrlHandlerMapping
    - URL과 Controller의 Bean 이름을 매핑함
    - Bean 이름을 URL과 동일하게 /XXX.do 형식으로 만들어야 함
    - Bean을 등록할 때 id 속성 대신 name 속성을 사용해야만 '/' 같은 특수문자를 입력할 수 있음

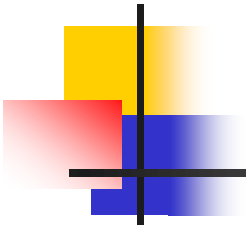


ViewResolver

- ViewResolver
  - ModelAndView 객체가 리턴한 View 의 이름을 이용하여 사용자에게 보여줄 뷰(jsp)를 결정하는 역할을 함

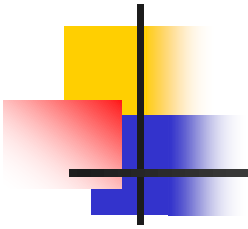


@Controller, @RequestMapping 어노테이션 사용



어노테이션(Annotation, Metadata)

- 어노테이션
  - 클래스나 메서드 등의 선언시에 @를 사용하는 것
    - JDK 5.0 부터 등장
  - [1] 컴파일러에게 정보를 알려주거나
  - [2] 컴파일할 때와 설치(deployment) 시의 작업을 지정하거나
  - [3] 실행할 때 별도의 처리가 필요할 때 사용함
  - 어노테이션은 클래스, 메소드, 변수 등 모든 요소에 선언할 수 있음



어노테이션(Annotation, Metadata)

- 자바에 미리 정해져 있는 어노테이션
  - @Override
    - 해당 메서드가 부모 클래스에 있는 메서드를 Override 했다는 것을 명시적으로 선언함
    - 오버라이딩 문법에 맞지 않게 잘못 코딩하면 컴파일러가 에러를 발생함
  - @Deprecated
  - @SuppressWarnings

Annotation 사용

- 과도한 xml 설정으로 인한 불편함을 해결하기 위해서 Annotation 기반의 설정 방법을 적용함
- Spring MVC 도 xml 설정을 최소화할 수 있도록 Annotation 기반 설정을 지원하는데, 기본 설정은 DI 설정과 동일함
- [1] @Controller 사용
  - Controller 클래스를 Controller Bean 으로 인식시키기 위해 사용되는 @Controller 어노테이션
  - @Controller 어노테이션이 적용된 클래스는 어떤 Controller 인터페이스 클래스도 상속할 필요가 없음
    - Controller 클래스는 POJO로 구현될 수 있다는 의미

Annotation 사용

- @Controller 사용시

```
[LoginController.java]
import org.springframework.stereotype.Controller;

@Controller
public class LoginController {
    public String loginUser(UserVO vo){
        return null;
    }
}
```

- 어노테이션 사용하지 않는 경우

```
public class LoginController implements Controller {
      public ModelAndView handleRequest(HttpServletRequest request,
                                  HttpServletResponse response) throws Exception {
                ...
}
```

@RequestMapping 어노테이션

- @RequestMapping 을 Controller 클래스의 **메소드에 설정**할 때, 요청 URI와 적절하게 매핑하면 됨
  - @RequestMapping 어노테이션을 이용하면 클라이언트의 요청방식(GET/POST)에 따라서 수행될 메서드를 다르게 설정할 수 있음

[LoginController.java]

```
package com.testmall.view.user.controller;

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

@Controller
public class LoginController {
    @RequestMapping("/login.do")
    public ModelAndView login(HttpServletRequest request,
        HttpServletResponse response) {
    }
}
```

[WEB-INF/spring-mvc.xml]

```
<!-- Controller -->
<bean id="loginController" class="com.testmall.view.user.LoginController"/>
```

@RequestMapping 어노테이션

- 어노테이션을 사용하지 않고, xml 설정 파일에 설정한다면

```
[WEB-INF/spring-mvc.xml]
<!-- HandlerMapping -->
<bean id="handlerMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
            <props>
                <prop key="/login.do">loginController</prop>
            </props>
    </property>
</bean>

<!-- Controller -->
<bean id="loginController" class="com.testmall.view.user.LoginController"/>
```

@RequestMapping 어노테이션

- 클라이언트 요청 방식에 따른 @RequestMapping 사용방법
- [1] 클라이언트가 get방식으로 입력 폼을 요청하는 경우 – 입력 화면을 보여주고, 입력화면에서 submit 버튼을 클릭하여 post 방식으로 요청할 경우의 처리 방법

[LoginController.java]

```
package com.multicampus.view.user.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class LoginController {
    @RequestMapping(value="/login.do", method=RequestMethod.GET)
    public String loginView(HttpServletRequest request,
        HttpServletResponse response) {
    }

    @RequestMapping(value="/login.do", method=RequestMethod.POST)
    public String login(HttpServletRequest request,
        HttpServletResponse response) {
    }
}
```

•/login.do 요청에 대해서 GET방식으로 요청이 들어오면 loginView() 메서드를 실행하여 로그인 화면을 보여줌
•/login.do 요청에 대해서 POST 방식으로 요청이 들어오면 login()메서드를 실행하여 실질적인 로그인 처리 작업을 수행하도록 함

@RequestMapping 어노테이션

- [2] 클라이언트의 요청이 동일한 URL을 사용할 경우

[LoginController.java]

```
package com.multicampus.view.user.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/login.do")
public class LoginController {
    @RequestMapping(method=RequestMethod.GET)
    public String loginView(HttpServletRequest request,
        HttpServletResponse response) {
    }

    @RequestMapping(method=RequestMethod.POST)
    public String login(HttpServletRequest request,
        HttpServletResponse response) {
    }
}
```

• 동일한 /login.do 요청에 대해서 GET방식으로 요청이 들어오면 loginView() 메서드를 실행하여 로그인 화면을 보여줌
• POST 방식으로 요청이 들어오면 login()메서드를 실행하여 실질적인 로그인 처리 작업을 수행하도록 함

@RequestMapping 어노테이션에 method 속성을 설정하지 않을 경우, get, post 등 모든 http 전송 방식을 처리하게 됨

클라이언트 요청 처리

- 대부분의 Controller 가 Model에 해당하는 컴포넌트의 메서드를 호출하는데, 이때 사용되는 사용자의 입력정보는 어떻게 추출할까?
  - request.getParameter()
  - 또는 request.getParameterValues() 메서드 사용
- 예) 사용자로부터 아이디와 비밀번호를 입력 받아서 "/login.do" 로 매핑된 Controller 를 호출하는 경우.

```
<form name="frm1" method="post" action="/login.do">
        아이디 : <input type="text" name="id"><br>
        비밀번호 : <input type="password" name="password"><br>
        <input type="submit" value="전송">
</form>
```

public String login(**@RequestParam("id") String id, @RequestParam("password") String password**)

클라이언트 요청 처리

[LoginController.java]

```
@Controller
public class LoginController {
        @RequestMapping("/login.do")
        public String login(HttpServletRequest request) {
                String id = request.getParameter("id");
                String password = request.getParameter("password");
                UserVO vo = new UserVO();
                vo.setId(id);
                vo.setPassword(password);

                int n = userService.insertUser(vo);
        }
}
```

```
@Controller
public class LoginController {
    @RequestMapping("/login.do")
    public String login(@RequestParam String id, @RequestParam  String password)
throws Exception {
                UserVO vo = new UserVO();
                vo.setId(id);
                vo.setPassword(password);
                int n = userService.insertUser(vo);
    }
}
```

클라이언트 요청 처리

- 복잡하고 길게 작성되는 login() 메서드를 자바빈을 이용해서 구현하면 간단하게 구현할 수 있음

```
[LoginController.java]
@Controller
public class LoginController {
    @RequestMapping("/login.do")
    public String login(@ModelAttribute UserVO vo) throws Exception {
        int n = userService.insertUser(vo);
    }
}
```

- login() 메서드의 매개변수로 사용자가 입력한 값을 매핑할 수 있는 자바빈을 등록하면 Spring 컨테이너가 자바빈을 생성하여 넘겨줌
- 이때 사용자가 입력한 값을 자바빈의 property에 자동으로 채워주기까지 함
- => 사용자가 입력한 값 추출과 자바빈 생성 및 값 설정 과정을 컨테이너에 의해서 자동으로 처리할 수 있는데, 이런 자바빈을 Command 객체라고 함

클라이언트 요청 처리

- Command 객체가 어떻게 사용되는지 보자

```
[login.jsp]
<%@page contentType="text/html; charset=utf-8"%>
<html><head><title>로그인</title></head>
<body>
<form action="login.do" method="post">
<table border="1" cellpadding="0" cellspacing="0">
<tr><td>아이디</td><td><input name="id" type="text" /></td></tr>
<tr><td>비밀번호</td><td><input name="password" type="password" /></td></tr>
<tr><td colspan="2" align="center"><input type="submit" value="로그인" /></td></tr>
</table>
</form>
</body>
</html>
```

```
[UserVO.java]
public class UserVO {
      private String id;
      private String password;

        // public Getter/Setter methods
}
```

입력폼에서 name="id"라는 input 박스에 입력한 값은 UserVO객체의 id변수에 자동으로 저장됨

클라이언트 요청 처리

- jsp 에서 Command 객체 활용하기
  - Command 객체에 설정된 정보를 View에 해당하는 jsp 에서 사용하려면?
  - ${..} EL 구문 사용

컨트롤러의 처리 결과를 보여주는 뷰 코드에서는 커맨드 객체의 클래스 이름을 이용해서 커맨드 객체에 접근할 수 있음
즉, **커맨드 객체는 자동으로 모델에 추가됨**(단, 첫 글자는 소문자임)

```
@RequestMapping("/login.do")
public String login(@ModelAttribute UserVO user)
throws Exception {

}
```

변수명을 무시해버린다(user을 취급안함)
클래스명으로 취급한다(userVO.어쩌구...식으로 나옴)

```
<body>
   <h3>${userVO.id}님 로그인 환영합니다.</h3>
</body>
```

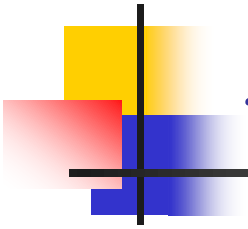
login() 메서드가 실행된 이후에 클라이언트에 전송되는 jsp에서는 Command 객체의 이름을 통해서 프로퍼티에 접근할 수 있음
(단, jsp에서 Command 객체를 사용하기 위해서는 **Command 클래스 이름을 소문자로 해서** 접근해야 함

클라이언트 요청 처리

- Command 클래스의 이름을 다른 이름으로 변경하고자 한다면?

```
@RequestMapping("/login.do")
public String login(@ModelAttribute("myuser") UserVO user) throws Exception {

}
```

```
<body>
   <h3>${myuser.id}님 로그인 환영합니다.</h3>
</body>
```



클라이언트 요청 처리

- Command 객체에는 없는 파라미터의 경우, Controller 클래스에서 어떻게 사용해야 할까?
  - Command 객체를 이용하기 위해서는 요청 파라미터와 동일한 이름의 변수가 Command 클래스에 선언되어 있어야 함
  - Command 객체에는 없는 파라미터를 Controller 클래스에서 사용하려면?
    - Spring MVC에서는 HTTP 요청 파라미터 정보를 추출하기 위해서 @RequestParam 어노테이션을 제공함

클라이언트 요청 처리

- 예) @RequestParam 어노테이션을 이용하여 검색과 관련된 파라미터 사용하기

```
GetUserListController.java]
@Controller
public class getUserListController {
        @RequestMapping("/getUserList.do")
        public String getUserList(
            @RequestParam("searchCondition") String searchCondition,
            @RequestParam("searchKeyword") String searchKeyword){
               System.out.println("검색 조건 : " + searchCondition);
               System.out.println("검색 단어 : " + searchKeyword);
               return null;
        }
}
```

• @RequestParam 은 request 내장객체에서 제공하는 getParameter() 메서드와 동일한 기능의 어노테이션
• @RequestParam 으로 추출한 파라미터 값이 없는 경우에는 **null**이 할당되므로 NullPointerException을 방지하기 위해 null에 대한 적절한 로직을 추가해야 함

```
@RequestMapping("/getUserList.do")
        public String getUserList(@RequestParam  String searchCondition,
                                  @RequestParam  String searchKeyword){ }
```

```
@RequestMapping("/getUserList.do")
        public String getUserList(String searchCondition,  String searchKeyword){
}
```

클라이언트 요청 처리

- null에 대한 적절한 로직

```
public String getUserList(
    @RequestParam("searchCondition") String searchCondition,
    @RequestParam("searchKeyword") String searchKeyword){
    if(searchCondition == null)
        searchCondition = "NAME";
    if(searchKeyword == null)
        searchKeyword = "";
    return null;
}
```

- 기본값 설정도 @RequestParam 을 이용하면 간단하게 처리할 수 있음

```
public String getUserList(
    @RequestParam(value="searchCondition", defaultValue="NAME") String searchCondition,
    @RequestParam(value="searchKeyword", defaultValue="") String searchKeyword){
    return null;
}
```

@RequestParam 어노테이션을 이용한 파라미터 매핑

- 컨트롤러를 구현하면서 가장 많이 사용되는 어노테이션 – @RequestParam 어노테이션
  - Http 요청 파라미터를 메서드의 파라미터로 전달받을 때 사용됨
  - @RequestParam 어노테이션과 Http 요청 파라미터의 관계

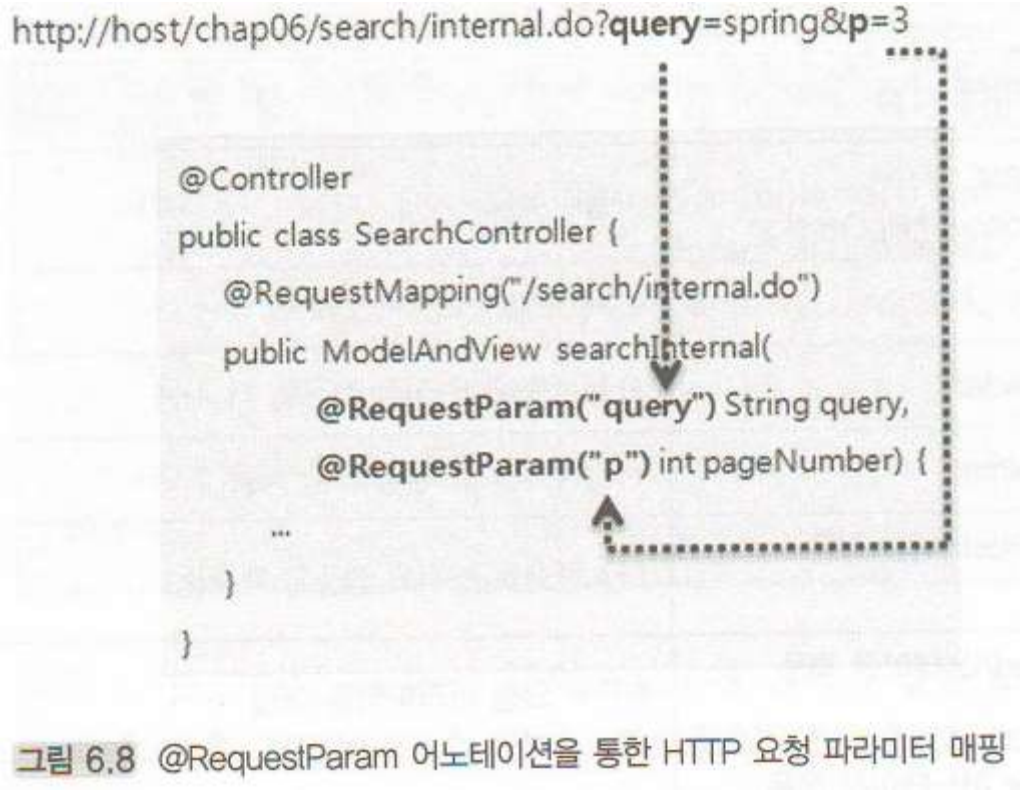


그림 6.8 @RequestParam 어노테이션을 통한 HTTP 요청 파라미터 매핑

서버->톰캣->config 폴더 들어가서 server.xml
아래의 context를 꼭 고치기

@RequestParam 어노테이션을 이용한 파라미터 매핑

- @RequestParam 어노테이션이 적용된 파라미터가 String이 아닐 경우 실제 타입에 따라서 알맞게 타입 변환을 수행함
  - 예) pageNumber 파라미터의 타입은 int, 이 경우 자동으로 문자열을 int 타입으로 변환해줌
- @RequestParam 어노테이션이 적용된 파라미터는 **기본적으로 필수 파라미터**임
  - @RequestParam 어노테이션에 명시된 Http **요청 파라미터가 존재하지 않을 경우** 스프링MVC는 잘못된 요청(Bad Request)을 의미하는 **400 응답 코드**를 웹 브라우저에 전송함
- **필수가 아닌 파라미터**의 경우 **required 속성 값을 false** 로 지정해 주면 됨
  - required 속성의 기본값은 true

```
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class SearchController {

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

@RequestParam 어노테이션을 이용한 파라미터 매핑

- 필수가 아닌 요청 파라미터의 값이 존재하지 않을 경우 null 값을 할당함
- null 을 할당할 수 없는 기본 데이터 타입인 경우에는 타입 변환 에러 발생
  - 예) p 요청 파라미터를 필수가 아닌 파라미터로 설정했는데, p 요청 파라미터를 지정하지 않은 경우 null 을 기본 데이터 타입으로 변환할 수 없다는 예외 발생
- 기본 데이터 타입을 사용할 경우 http 요청 파라미터가 존재하지 않으면 기본 값을 할당하는 경우가 많은데, 이런 경우에는 defaultValue 속성을 이용해서 기본값을 지정할 수 있음

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

defaultValue 속성을 이용해서 기본 값을 지정하면 해당 요청 파라미터를 지정하지 않을 경우 defaulValue 속성에 지정한 문자열을 값으로 이용하게 됨
예) http://localhost:8080/chap06/search/external.do?query=spring
⇒ p 파라미터가 존재하지 않으므로 기본값으로 지정한 1 을 p 파라미터의 값으로 사용
⇒ pageNumber 파라미터의 값은 1

컨트

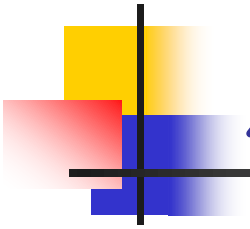
- 컨트롤러의 @RequestMapping 어노테이션이 적용된 메서드는 커맨드 클래스뿐만 아니라 HttpServletRequest, HttpSession, Locale 등 웹 어플리케이션과 관련된 다양한 타입의 파라미터를 가질 수 있음

[표 6.2] 컨트롤러 메서드의 파라미터 타입

파라미터 타입	설 명
HttpServletRequest, HttpServletResponse, HttpSession	서블릿 API
java.util.Locale	현재 요청에 대한 Locale
InputStream, Reader	요청 컨텐츠에 직접 접근할 때 사용
OutputStream, Writer	응답 컨텐츠를 생성할 때 사용
@PathVariable 어노테이션 적용 파라미터	URI 템플릿 변수에 접근할 때 사용
@RequestParam 어노테이션 적용 파라미터	HTTP 요청 파라미터를 매핑
@RequestHeader 어노테이션 적용 파라미터	HTTP 요청 헤더를 매핑
@CookieValue 어노테이션 적용 파라미터	HTTP 쿠키 매핑
@RequestBody 어노테이션 적용 파라미터	HTTP 요청의 몸체 내용에 접근할 때 사용. HttpMessageConverter를 이용해서 HTTP 요청 데이터를 해당 타입으로 변환한다.
Map, Model, ModelMap	뷰에 전달할 모델 데이터를 설정할 때 사용

클라이언트 요청 처리

- Command 객체나 @RequestParam 보다 HttpServletRequest 를 이용하는 방식이 더 효과적일 때가 있음
- HttpServletRequest 와 같이 매개변수로 사용될 수 있는 ServletAPI
- (Spring MVC 에서 Controller 메서드의 매개변수로 지원하는 ServletAPI)
  - [1] ServletRequest/ HttpServletRequest
  - [2] ServletResponse/ HttpServletResponse
  - [3] HttpSession



서블릿 API 직접 사용

- 컨트롤러 클래스의 @RequestMapping 어노테이션이 적용된 메서드는 다음의 5가지 타입의 파라미터를 전달받을 수 있음
  - javax.servlet.http.HttpServletRequest/javax.servlet.ServletRequest
  - javax.servlet.http.HttpServletResponse/javax.servlet.ServletResponse
  - javax.servlet.http.HttpSession
- 서브릿 API를 사용할 필요 없이 스프링 MVC 가 제공하는 어노테이션을 이용해서 요청 파라미터, 헤더, 쿠키, 세션 등의 정보에 접근할 수 있기 때문에, 직접적으로 API를 사용해야 하는 경우는 매우 드물다
- 다음의 경우에는 서블릿 API를 사용하는 것이 더 편리
  - HttpSession 의 생성을 직접 제어해야 하는 경우
  - 컨트롤러에서 쿠키를 생성해야 하는 경우
  - 서블릿 API 사용을 선호하는 경우

서블릿 API 직접 사용

- 예) 조건에 따라서 HttpSession 을 생성해야 하는 경우
  - HttpServletRequest 타입의 파라미터를 전달받아야만 HttpSession 을 조건에 따라 생성하거나 생성하지 않을 수 있음

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

  - HttpSession 타입의 파라미터를 가질 경우 세션이 생성된다는 점에 유의
  - 즉, 기존에 세션이 존재한다면 해당 세션이 전달되고, 그렇지 않다면 새로운 세션이 생성되고, 관련 HttpSession 인스턴스가 파라미터에 전달됨
  - 따라서 HttpSession 타입의 파라미터는 항상 null 이 아님

클라이언트 요청 처리

- 매개변수로 HttpServletRequest 객체를 사용

```
[LoginController.java]
@Controller
public class LoginController {
	@RequestMapping("/login.do")
	public String login(UserVO vo, HttpServletRequest request) {
		UserDAO dao = new UserDAO();
		UserVO user = dao.getUser(vo);
		if(user != null){
			request.getSession().setAttribute("userId", user.getId());
			return "redirect:/getUserList.do";
		}else{
			return "login.jsp";
		}
	}
}
```

```
public String login(UserVO vo, HttpSession session) {
		.....
		session.setAttribute("userId", user.getId());
```

로그인을 인증하고 로그인을 성공했을 경우에만 HttpSession 객체를 생성하여 아이디를 세션에 저장하는 예제
=> 특정 조건에 대해서 세션객체를 생성해야 하는 경우에 매개변수로 HttpServletRequest 객체가 필요함

required 속성의 값을 false로 지정할 경우, 해당 쿠키가 존재하지 않으면 null값을 전달받게 됨

@CookieValue 어노테이션을 이용한 쿠키 매핑

- @CookieValue 어노테이션을 이용하면 쿠키 값을 파라미터로 전달받을 수 있음
- 예) auth 쿠키의 값을 authValue 파라미터를 통해서 전달받도록 작성한 코드

```
import org.springframework.web.bind.annotation.CookieValue;
@Controller
public class CookieController {
        @RequestMapping("/cookie/view.do")
        public String view(@CookieValue(value = "auth") String authValue) {
                System.out.println("auth 쿠키: " + authValue);
                return "cookie/view";
        }
}
```

- @CookieValue 어노테이션은 해당 쿠키가 존재하지 않으면 기본적으로 500 에러를 발생시킴
- 쿠키가 필수가 아닌 경우에 required 속성의 값을 false로 지정

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

@CookieValue 어노테이션을 이용한 쿠키 매핑

- @RequestParam 어노테이션과 마찬가지로 defaultValue 속성을 이용해서 기본 값을 설정

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

• 헤더 정보
host, user-agent, Accept-Language 등

@RequestHeader 어노테이션을 이용한 헤더 매핑

- @RequestHeader 어노테이션을 이용하면 Http 요청 헤더의 값을 메서드의 파라미터로 전달받을 수 있음

```
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HeaderController {

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

- @RequestHeader 어노테이션도 @RequestCookie 어노테이션과 마찬가지로 해당 헤더가 존재하지 않으면 500 응답 에러 코드 전송
- required, defaultValue 속성을 이용해서 필수여부와 기본 값을 설정

Controller 의 리턴 타입

- Controller 의 리턴 타입의 종류

타입	설명
ModelAndView	Model과 View 정보가 저장된 객체임
String	View 정보를 정확한 이름으로 설정함
Model	View에 전달할 객체들을 담고 있는 객체임
Map	View에 전달할 정보가 저장된 Map 객체를 리턴함 - 이때 View 이름은 요청 URL을 통해서 자동으로 결정됨

컨트롤러 메서드의 리턴 타입

- 컨트롤러 메서드는 ModelAndView를 비롯한 몇 가지 리턴 타입을 가질 수 있음

[표6.3] @RequestMapping 메서드의 리턴 타입

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

Controller 의 리턴 타입

- ModelAndView 를 리턴하는 경우

```
public ModelAndView login(@ModelAttribute UserVO vo, HttpServletRequest request) {
        UserDAO dao = new UserDAO();
        UserVO user = dao.getUser(vo);
        ModelAndView mav = new ModelAndView();

        if(user != null){
            mav.setViewName("getUserList");
        }else{
            mav.setViewName("login");
        }

        return mav;
}
```

  - ModelAndView 를 리턴하는 경우는 직접 View 정보를 ModelAndView 에 담아서 알려주겠다는 것

컨트롤러 클래스 자동 스캔

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

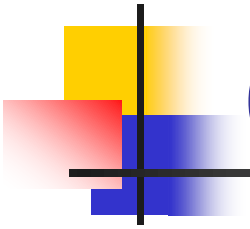
```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

mysite.spring.chap06.controller 패키지로 시작하는 모든 패키지의 클래스들을 스캔함
=> 각 Controller 의 @Controller 어노테이션이 인식됨
=> Controller클래스가 자동으로 빈으로 등록됨

클라이언트 요청 처리

- view 정보를 설정하여 리턴할 때 특정 화면으로 Redirect 하고자 하는 경우
  - "redirect: " 라는 접두사를 붙이면 됨
  - "redirect: " 접두사 없이 view 정보를 설정하면 기본적으로 Forwarding 됨

```
public ModelAndView login(@ModelAttribute UserVO vo, HttpServletRequest request) {
        UserDAO dao = new UserDAO();
        UserVO user = dao.getUser(vo);
        ModelAndView mav = new ModelAndView();
        if(user != null){
                mav.setViewName("redirect:/getUserList.do");
        }else{
                mav.setViewName("login");
        }
        return mav;
}
```



예제-스프링 MVC Hello World

- 스프링 MVC를 이용하여 웹 어플리케이션을 개발하는 과정
  - [1] 클라이언트의 요청을 받을 DispatcherServlet 을 web.xml에 설정함
  - [2] 클라이언트의 요청을 처리할 컨트롤러를 작성함
  - [3] ViewResolver를 설정함
    - ViewResolver는 컨트롤러가 전달한 값을 이용해서 응답 화면을 생성할 뷰를 결정함
  - [4] jsp 등을 이용하여 뷰 영역의 코드를 작성함

1단계- DispatcherServlet 설정 및 스프링 컨텍스트 설정

- web.xml 파일에 다음 2가지 정보를 추가
- [1] 클라이언트의 요청을 전달받을 DispatcherServlet 설정
  - 서블릿과 서블릿 매핑 정보를 추가
- [2] 공통으로 사용할 어플리케이션 컨텍스트 설정

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

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

• *.do 로 들어오는 클라이언트의 요청을 DispatcherServlet 이 처리하도록 설정
• DispatcherServlet 은 WEB-INF/디렉토리에 위치한 [서블릿 이름]-servlet.xml 파일을 스프링 설정 파일로 사용함
• 예) 위 코드의 경우 dispatcher-servlet.xml 파일을 설정 파일로 사용하게 됨
 - 이 파일에서 스프링 MVC의 구성요소인 Controller, ViewResolver, View 등의 빈을 설정하게 됨

2단계– 컨트롤러 구현 및 설정 추가

- 컨트롤러를 구현하려면 먼저 @Controller 어노테이션을 클래스에 적용함
- 그리고, @RequestMapping 어노테이션을 이용해서 클라이언트의 요청을 처리할 메서드를 지정함

```
package mysite.spring.chap06.controller;
import java.util.Calendar;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
@Controller
public class HelloController {
        @RequestMapping("/hello.do")
        public ModelAndView hello() {
                ModelAndView mav = new ModelAndView();
                mav.setViewName("hello");
                mav.addObject("greeting", getGreeting());
                return mav;
        }
        private String getGreeting() {
                int hour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
                if (hour >= 6 && hour <= 10) {
                        return "좋은 아침입니다.";
                } else if (hour >= 12 && hour <= 15) {
                        return "점심 식사는 하셨나요?";
  } else if (hour >= 18 && hour <= 22) {
                return "좋은 밤 되세요";
  }
                return "안녕하세요";
  }
}
```

2단계- 컨트롤러 구현 및 설정 추가

- @Controller 어노테이션 – 해당 클래스가 스프링 MVC의 컨트롤러를 구현할 클래스라는 것을 지정함
- @RequestMapping 어노테이션 – 값으로 지정한 요청 경로를 처리할 메서드를 설정함
  - @RequestMapping("/hello.do") => http://host:port/컨텍스트 경로/hello.do 요청을 HelloController 클래스의 hello() 메서드가 처리하게 됨
- ModelAndView – 컨트롤러의 처리결과를 보여줄 뷰와 뷰에서 출력할 모델을 지정할 때 사용됨
  - 사용할 뷰 이름으로 "hello" 를 지정, 모델에 "greeting" 이라는 이름으로 String 타입의 값을 추가
- 스프링 MVC는 ModelAndView 뿐만 아니라 String, ModelMap, Map 과 같은 타입을 이용해서 뷰 이름과 모델 정보를 설정할 수 있음
- DispatcherServlet 은 스프링 컨테이너에서 컨트롤러 객체를 검색하기 때문에 스프링 설정 파일에 컨트롤러를 빈으로 등록해 주어야 함

2단계- 컨트롤러 구현 및 설정 추가

- dispatcher-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd>

            <bean id="helloController" class="mysite.spring.controller.HelloController" />

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

3단계-설정 파일에 ViewResolver 설정 추가

- 컨트롤러 클래스는 직접 또는 간접적으로 ModelAndView 객체를 생성하게 됨
  - 앞에서 ModelAndView.setViewName() 메서드를 이용해서 컨트롤러의 처리 결과를 보여줄 뷰 이름을 "hello" 로 지정하였는데, DispatcherServlet 은 이 뷰 이름과 매칭되는 뷰 구현체를 찾기 위해 ViewResolver 를 사용함
  - 스프링 MVC는 jsp, velocity, FreeMarker 등의 뷰 구현 기술과의 연동을 지원함
  - jsp 를 뷰 기술로 사용할 경우 다음과 같이 InternalResourceViewResolver 구현체를 빈으로 등록해 주면 됨

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

prefix 프로퍼티와 suffix 프로퍼티의 값으로 각각 "/WEB-INF/view/" 와 ".jsp" 를 설정
⇒이 ViewResolver 가 "/WEB-INF/view/뷰이름.jsp" 를 뷰 jsp 로 사용한다는 것을 의미함
⇒ HelloController 는 뷰 이름으로 hello 를 리턴하므로, 실제로 사용되는 뷰 파일은
"/WEB-INF/view/**hello**.jsp" 파일이 됨

4단계-뷰 코드 구현

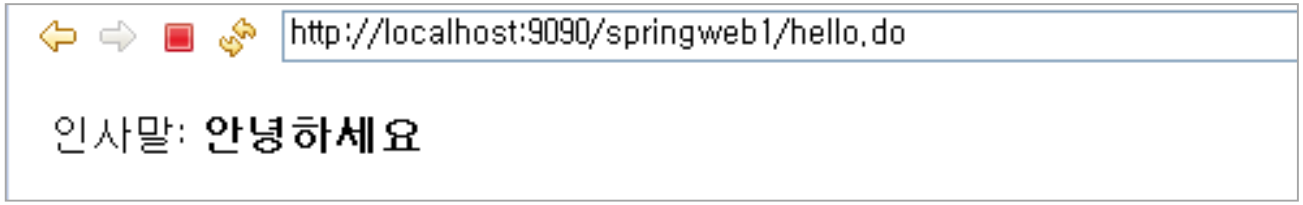
- HelloController 의 처리 결과를 보여줄 jsp 코드

```
<%@ page language="java" contentType="text/html; charset=utf-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>인사</title>
</head>
<body>
인사말: <strong>${greeting}</strong>
</body>
</html>
```

- ${greeting} => greeting 이라는 이름은 HelloController 에서 추가한 모델의 이름과 동일
- 뷰 코드에서는 모델을 추가할 때 사용한 이름을 이용해서 해당 모델의 값을 출력할 수 있게 됨

5단계-실행

- http://localhost:8080/springweb/hello.do



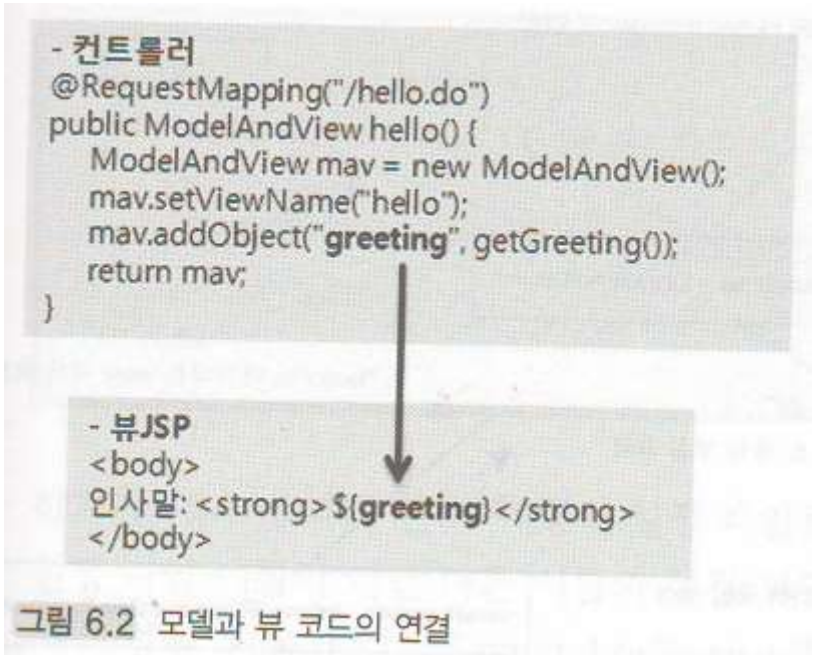


그림 6.2 모델과 뷰 코드의 연결

실행 흐름 정리

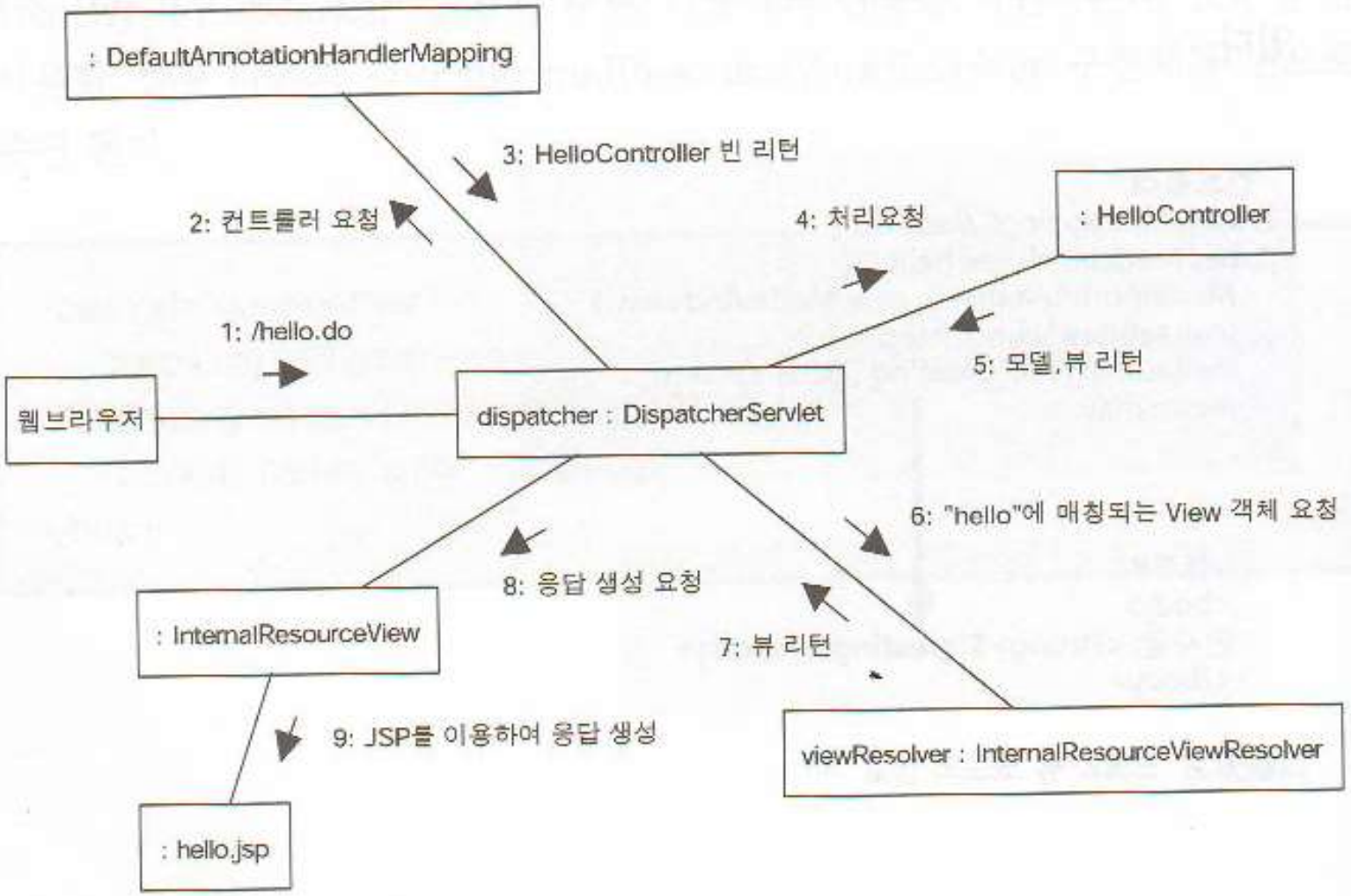
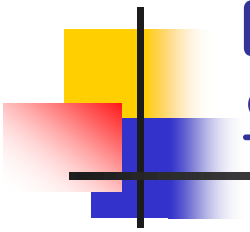


그림 6.4 /hello.do 요청 처리 과정



DispatcherServlet 설정과 ApplicationContext의 관계

- DispatcherServlet – 클라이언트의 요청을 중앙에서 처리하는 스프링 MVC의 핵심 구성 요소
  - web.xml 파일에 한 개 이상의 DispatcherServlet 을 설정할 수 있으며, 각 DispatcherServlet 은 한 개의 WebApplicationContext 를 갖게 됨
  - 또한 각 DispatcherServlet 이 공유할 수 있는 빈을 설정할 수도 있음

DispatcherServlet 설정

- DispatcherServlet 은 기본적으로 웹 어플리케이션의 /WEB-INF/디렉토리에 위치한 [서블릿이름]-servlet.xml 파일로부터 스프링 설정 정보를 읽어옴
- 한 개 이상의 설정 파일을 사용해야 하는 경우나 기본 설정 파일 이름이 아닌 다른 이름의 설정 파일을 사용하고 싶은 경우
  - DispatcherServlet 을 설정할 때 contextConfigLocation 초기화 파라미터에 설정 파일 목록을 지정하면 됨

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

- contextConfigLocation 초기화 파라미터는 설정 파일 목록을 값으로 갖는데,
- 각 설정파일은 콤마(,), 공백 문자(" "), 탭(□t), 줄 바꿈(□n), 세미콜론(";") 을 이용하여 구분
- 각 설정 파일의 경로는 웹 어플리케이션 루트 디렉토리를 기준으로 함

웹 어플리케이션을 위한 ApplicationConext 설정

- DispatcherServlet 은 그 자체가 서블릿이기 때문에 한 개 이상의 DispatcherServlet 을 설정하는 것이 가능
- 예) 웹 페이지를 위한 DispatcherServlet 과 REST 기반의 웹 서비스 연동을 위한 DispatcherServlet 을 나누어 설정했다면.

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

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

웹 어플리케이션을 위한 ApplicationConext 설정

- 이 경우 두 DispatcherServlet은 각각 별도의 WebApplicationContext를 생성하게 됨
- front DispatcherServlet 은 front.xml 설정 파일을 사용하고, rest DispatcherServlet 은 rest.xml 설정 파일을 사용=> front.xml 에서는 rest.xml 에 설정된 빈 객체를 사용할 수 없게 됨
- 웹 어플리케이션에서 컨트롤러는 클라이언트의 요청을 비즈니스 로직을 구현한 서비스 레이어를 이용하여 처리하는 것이 일반적
- 서비스 레이어는 영속성 레이어(db연동)를 사용해서 데이터 접근을 처리
- front 관련 컨트롤러와 rest 관련 컨트롤러는 동일한 서비스 레이어에 대한 의존 관계를 가질 것임

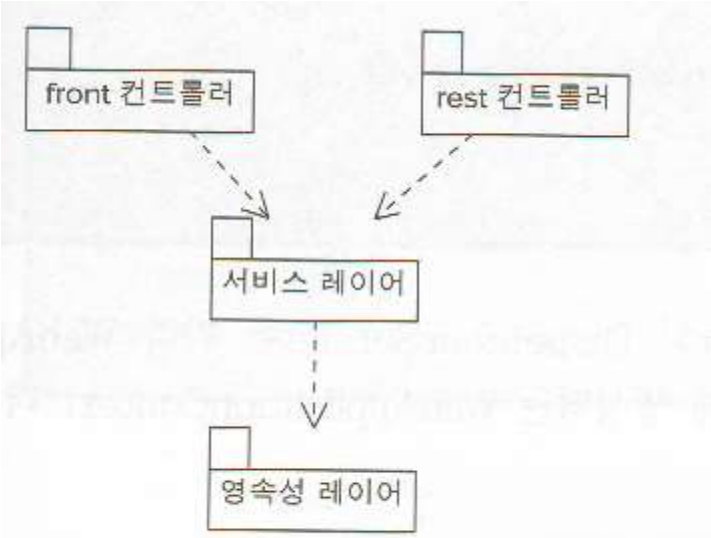


그림 6.5 웹 어플리케이션의 전형적인 레이어 구성

웹 어플리케이션을 위한 ApplicationConext 설정

- 서로 다른 DispatcherServlet 이 공통 빈을 필요로 하는 경우
  - ContextLoaderListener 를 사용하여 공통으로 사용될 빈을 설정할 수 있게 됨
  - ContextLoaderListener 를 ServletListener 로 등록하고, contextConfigLocation 컨텍스트 파라미터를 이용하여 공통으로 사용될 빈 정보를 담고 있는 설정 파일 목록을 지정하면 됨

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

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

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

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

웹 어플리케이션을 위한 ApplicationConext 설정

- ContextLoaderListener 와 DispatcherServlet 은 각각 WebApplicationContext 객체를 생성하는데, 이때 생성되는 WebApplicationContext 객체 간의 관계는 아래 그림과 같다

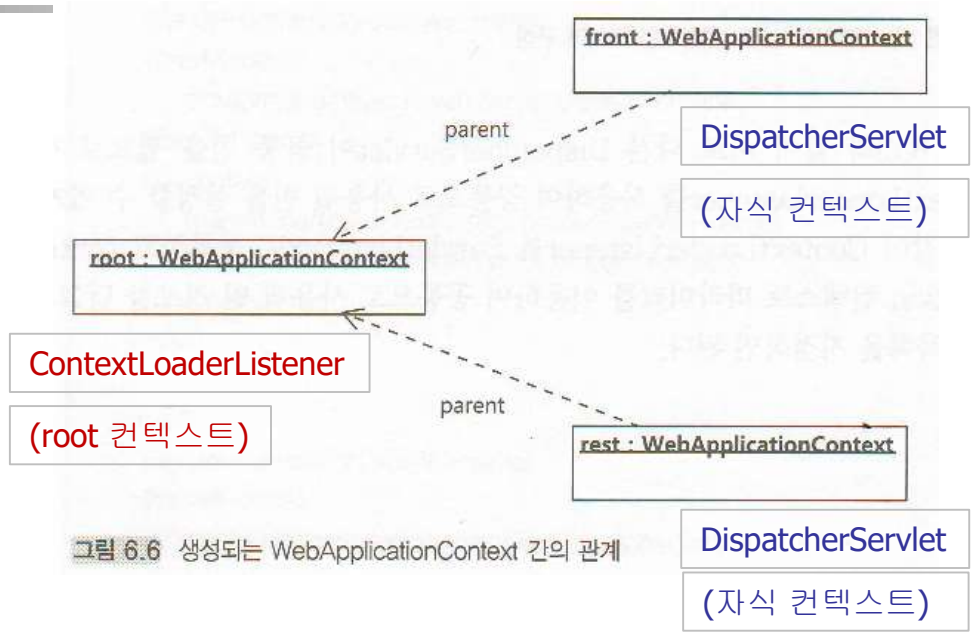


그림 6.6 생성되는 WebApplicationContext 간의 관계

- ContextLoaderListener 가 생성하는 WebApplicationContext 는 웹 어플리케이션에서 **루트 컨텍스트**가 되며, DispatcherServlet 이 생성하는 WebApplicationContext 는 루트 컨텍스트를 부모로 사용하는 자식 컨텍스트가 됨
  - 이때 **자식은 root 가 제공하는 빈을 사용할 수** 있음
  - ContextLoaderListener 는 contextConfigLocation 컨텍스트 파라미터를 명시하지 않으면 /WEB-INF/applicationContext.xml 을 설정 파일로 사용함
  - **클래스패스에 위치한 파일로부터 설정 정보를 읽어오고 싶으면 classpath: 접두어 사용**

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

• web.xml

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath*:config/spring/context-*.xml</param-value>
</context-param>
<listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
        </init-param>
</servlet>
<servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

캐릭터 인코딩 처리를 위한 필터 설정

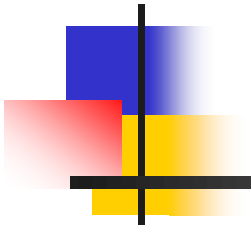
- 요청 파라미터의 캐릭터 인코딩이 ISO-8859-1 이 아닌 경우, request.setCharacterEncoding() 메서드를 사용해서 요청 파라미터의 캐릭터 인코딩을 설정해 주어야 함

```
request.setCharacterEncoding("UTF-8")
```

- 모든 컨트롤러에서 위 코드를 실행하는 것보다는, 서블릿 필터를 이용해서 원하는 요청에 위 코드를 적용하는 것이 더 편리함
- 스프링은 요청 파라미터의 캐릭터 인코딩을 설정할 수 있는 필터 클래스인 CharacterEncodingFilter 클래스를 제공
- web.xml 에 CharacterEncodingFilter 클래스를 설정하자

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

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

요청 파라미터의 캐릭터 인코딩은 encoding 초기화 파라미터를 통해서 지정



PD 테이블 이용

예제-dispatcher-servlet.xml

```
<!-- pd  Controller-->
<bean id="pdInsertController"
class="com.mysite.pd.controller.PdInsertController"
p:pdDAO-ref="pdDAO" />

<bean id="pdListController"
class="com.mysite.pd.controller.PdListController"
p:pdDAO-ref="pdDAO" />

<bean id="pdDetailController"
class="com.mysite.pd.controller.PdDetailController"
p:pdDAO-ref="pdDAO" />

<bean id="pdEditController"
class="com.mysite.pd.controller.PdEditController"
p:pdDAO-ref="pdDAO" />

<bean id="pdDeleteController"
class="com.mysite.pd.controller.PdDeleteController"
p:pdDAO-ref="pdDAO" />
```

```
<!-- dao -->
   <bean id="pdDAO"
      class="com.mysite.pd.model.PdDAO"
      p:pool-ref="pool" />

   <bean id="pool"
      class="com.herbmall.db.ConnectionProvider" />
```

PdInsertController

```
package com.mysite.pd.controller;
@Controller
@RequestMapping("/pd/pdInsert.do")
public class PdInsertController {
		private PdDAO pdDAO;

		public PdInsertController() {
			System.out.println("PdInsertController()생성");
		}

		public void setPdDAO(PdDAO pdDAO) {
			System.out.println("setPdDAO()호출");
			this.pdDAO = pdDAO;
		}

		@RequestMapping(method=RequestMethod.GET)
		public ModelAndView insertGet(){
			System.out.println("get : insertGet()");
			ModelAndView mav=new ModelAndView();
```

PdInsertController

```
		mav.setViewName("/pd/pdWrite");
		//WEB-INF/view   /pd/pdWrite   .jsp
		return mav;
	}
	@RequestMapping(method=RequestMethod.POST)
	public ModelAndView insertPost(@ModelAttribute("pdBean") PdBean pdBean){
		//파라미터값?, 사용자가 입력한 값은?
		System.out.println("POST : insertPost(), pdBean :"+pdBean);

		//DB처리작업
		//PdDAO dao=new PdDAO();
		//dao.insertPd(pdBean);
		try {
			int n = pdDao.insertPd(pdBean);
			System.out.println("상품 등록 성공, n="+n);
		} catch (SQLException e) {
			System.out.println("상품 등록 실패!");
			e.printStackTrace();
		}
		ModelAndView mav=new ModelAndView();
		mav.setViewName("redirect:/pd/pdList.do");
		return mav;
	}
}//class
```

PdListController

```
@Controller
public class PdListController {
	private PdDAO pdDAO;
	public PdListController() {
		System.out.println("PdListController()생성");
	}
	public void setPdDAO(PdDAO pdDAO) {
		System.out.println("setPdDAO()호출");
		this.pdDAO = pdDAO;
	}
	@RequestMapping("/pd/pdList.do")
	public ModelAndView getPdList(){
		System.out.println("getPdList() :");
		List<PdBean> list = pdDAO.listPdAll();
		ModelAndView mav=new ModelAndView();
		mav.addObject("list", list);
		mav.setViewName("/pd/pdList");
		return mav;
	}
}
```

PdDetailController

```
@Controller
public class PdDetailController {
	private PdDAO pdDAO;
	public PdDetailController() {
		System.out.println("PdDetailController()생성");
	}
	public void setPdDAO(PdDAO pdDAO) {
		System.out.println("setPdDAO()호출");
		this.pdDAO = pdDAO;
	}
	@RequestMapping("/pd/pdDetail.do")
	public ModelAndView getPdDetail(@RequestParam(value="no", defaultValue="0") int no){
		System.out.println("getPdDetail() :");
		PdBean bean = pdDAO.selectPdByNo(no);
		ModelAndView mav=new ModelAndView();
		mav.addObject("bean", bean);
		mav.setViewName("/pd/pdDetail");
		return mav;
	}
}
```

PdEditController

```
@Controller
@RequestMapping("/pd/pdEdit.do")
public class PdEditController {
		private PdDAO pdDAO;
		public PdEditController() {
			System.out.println("PdEditController()생성");
		}
		public void setPdDAO(PdDAO pdDAO) {
			System.out.println("setPdDAO()호출");
			this.pdDAO = pdDAO;
		}
		@RequestMapping(method=RequestMethod.GET)
		public ModelAndView editGet(int no){
			System.out.println("get : editGet()");
			PdBean bean = pdDAO.selectPdByNo(no);
			ModelAndView mav=new ModelAndView();
			mav.addObject("bean", bean);
			mav.setViewName("/pd/pdEdit");
			return mav;
		}
```

PdEditController

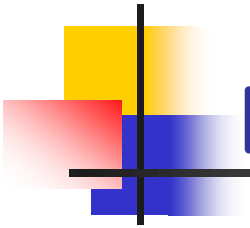
```
@RequestMapping(method=RequestMethod.POST)
public ModelAndView editPost(PdBean pdBean){
        System.out.println("POST : editPost(), pdBean :"+pdBean);

        int n = pdDAO.updatePd(pdBean);
        String msg="", url="";
        if(n>0){
                url="/pd/pdDetail.do?no="+ pdBean.getNo();
                msg="수정 성공";
        }else{
                url="/pd/pdEdit.do?no="+ pdBean.getNo();
                msg="수정 실패!!";
        }

        ModelAndView mav=new ModelAndView();
        mav.addObject("msg", msg);
        mav.addObject("url", url);
        mav.setViewName("/inc/message");
        return mav;
}
}//class
```

PdDeleteController

```
@Controller
public class PdDeleteController {
		private PdDAO pdDAO;
		public PdDeleteController() {
			System.out.println("PdDeleteController()생성");
		}
		public void setPdDAO(PdDAO pdDAO) {
			System.out.println("setPdDAO()호출");
			this.pdDAO = pdDAO;
		}
		@RequestMapping("/pd/pdDelete.do")
		public ModelAndView delete(int no){
			System.out.println("delete(), no:" + no);
			int n = pdDAO.deletePd(no);
			String msg="", url="";
			if(n>0){
				url="/pd/pdList.do";
				msg="삭제 성공";
			}else{
				url="/pd/pdDetail.do?no="+ no;
				msg="삭제 실패!!";
			}
```

PdDeleteController

```
                ModelAndView mav=new ModelAndView();
                mav.addObject("msg", msg);
                mav.addObject("url", url);

                mav.setViewName("/inc/message");
            return mav;
        }
}//class
```



pdWrite.jsp

```
<form name="frm" method="POST"
action="<c:url value='/pd/pdInsert.do'/>" >
  상품명: <INPUT TYPE="text" NAME="pdName"><br>
  가격:<INPUT TYPE="text" NAME="price"><br>
<INPUT TYPE="submit" value="등록">
<INPUT TYPE="reset" value="취소">
</form>
```

pdList.jsp

```
<%
    List<PdBean> list = (List<PdBean>)request.getAttribute("list");
%>
<h2>상품목록</h2>
<table width="600" border="1">
    <tr>
        <th>번호</th>
        <th>상품명</th>
        <th>가격</th>
        <th>등록일</th>
    </tr>

    <!-- 반복문 시작 -->
    <%
        for(int i=0;i<list.size();i++){
            PdBean bean = list.get(i);
    %>
```

pdList.jsp

```
            <tr>
                        <td><%=bean.getNo() %></td>
                        <td><a href="pdDetail.do?no=<%=bean.getNo()%>">
                                    <%=bean.getPdName() %></a>
                        </td>
                        <td><%=bean.getPrice() %></td>
                        <td><%=bean.getRegdate() %></td>
            </tr>
      <%
            }//for
      %>
      <!-- 반복문 끝 -->

</table>

<br><br>
<a href="<c:url value='/pd/pdInsert.do'/>">등록</a>
```

pdDetail.jsp

```
<script>
    function del(no){
        var result=confirm("삭제하시겠습니까?");
        //확인=>true, 취소=>false
        if(result){
            //삭제페이지로 이동
            location.href="pdDelete.do?no=" + no;
        }
    }
</script>
<%
    //1. get방식으로 보낸 parameter 값 읽어오기
    //http://localhost:9090/mystudy/pdtest/pdDetail.jsp?no=32
    String no= request.getParameter("no");

    PdBean bean = (PdBean)request.getAttribute("bean");
    DecimalFormat df = new DecimalFormat("#,###");
    //3. 상품정보를 화면에  출력하기
%>
```

pdDetail.jsp

```
<br><br>
<%=no %>번 클릭했습니다<br><br>
상품명 : <%=bean.getPdName() %><br>
가격 : <%=df.format(bean.getPrice()) %>원<br>
등록일 : <%= bean.getRegdate() %><br>
<br>
<hr>
<a href="pdList.do">목록</a> |
<a href="pdEdit.do?no=<%=no%>">수정</a> |
<a href="#" onclick="del(<%=no%>)">삭제</a>
```

pdEdit.jsp

```
<%
    //파라미터로 넘어온 no에 해당하는 상품을 조회
    PdBean bean = (PdBean)request.getAttribute("bean");
%>
<html><head>
<script type="text/javascript" >
    function send(){
        if(frm1.pdName.value==""){
            alert("상품명을 입력하세요");
            frm1.pdName.focus();
            return;
        }
        if(!frm1.price.value){
            alert("가격을 입력하세요");
            frm1.price.focus();
            return;
        }
        frm1.submit();
    }
</script></head>
<body>    <h2>상품 수정</h2>
    <form name="frm1" method="post" action="pdEdit.do">
```

pdEdit.jsp

```
        <input type="hidden" name="no" value="<%=bean.getNo()%>">
        <table width="300" border="0">
                <tr><td>상품명</td>
                        <td><input type="text" name="pdName"
                                        value="<%=bean.getPdName()%>">
                        </td>
                </tr>
                <tr><td>가격</td>
                        <td><input type="text" name="price"
    value="<%=bean.getPrice()%>">
                        </td>
                </tr>
                <tr><td colspan="2" align="center">
                                <input type="button" value="수정" onclick="send()">
                                <input type="reset" value="취소">
                        </td>
                </tr>
        </table>
    </form><br>
    <a href="pdList.do">상품목록</a>
</body>
```

message.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:if test="${!empty msg}">
      <script type="text/javascript">
            alert("${msg}");
            location.href="<c:url value='${url}' />";
      </script>
</c:if>
<c:if test="${empty msg}">
      <script type="text/javascript">
            location.href="<c:url value='${url}' />";
      </script>
</c:if>
```