

The background features a large, flowing green wave that curves across the frame. A semi-transparent white horizontal band is positioned in the middle, serving as a backdrop for the text. The bottom of the image is a solid dark green bar.

스프링 MVC

Quickstart (프로젝트 생성)

■ 웹 프로젝트에 필요한 폴더 (강조된 항목은 웹에 추가되는 폴더)

- src/main/java
- src/main/resources
- **src/main/webapp/WEB-INF/lib**
- **src/main/webapp/WEB-INF/views**

■ pom.xml 파일에 추가되는 항목

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>sp5</groupId>
  <artifactId>sp5-chap09</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
```

```
<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <version>2.3.2-b02</version>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.0.2.RELEASE</version>
  </dependency>
</dependencies>
```

Quickstart (이클립스 톰캣 설정)

■ 톰캣 설치

- 톰캣 다운로드 → 압축풀기
- `conf/server.xml` 파일 수정 (`port : 8080 → 8081`)

■ 이클립스 톰캣 등록

- 주메뉴 Window → Preferences
- Server → Runtime Environments → Add
- 단계별 마법사에서 다운로드한 톰캣 설치 경로 지정

■ 서버 등록

- JavaEE or Spring Perspective → Server Tab → New (서버 등록)

설치 상세 내역은 별도 톰캣 설치 파일 참고

Quickstart (설정)

▪ web.xml (xml based spring configuration)

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/spring/root-context.xml</param-value>
</context-param>
```

루트 웹 애플리케이션 컨텍스트 설정

```
<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>
  <load-on-startup>1</load-on-startup>
</servlet>
```

프론트 컨트롤러 서블릿 설정

```
<servlet-mapping>
  <servlet-name>appServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

서블릿 웹 애플리케이션 컨텍스트 설정

Quickstart (설정)

- web.xml (xml based spring configuration)
 - Root Web Application Context
 - » 웹 애플리케이션이 시작될 때 최상위 전역 웹 애플리케이션 컨텍스트 생성
 - » 모든 서블릿에서 공통으로 사용되는 빈을 설정하는 용도로 사용
 - Servlet Web Application Context
 - » 서블릿이 처음 요청될 때 웹 애플리케이션 컨텍스트 생성
 - » 이후 해당 서블릿에 대한 요청이 발생할 때 동작

Quickstart (설정)

▪ web.xml (code based spring configuration)

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextClass</param-name>
    <param-value>
      org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
  </init-param>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      config.MvcConfig
      config.ControllerConfig
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

FrontController Servlet 등록
IoC Container 설정

Quickstart (설정)

- servlet-context.xml 파일 (xml based spring web mvc configuration)

@Controller, @RequestMapping 활성화

```
<!-- DispatcherServlet Context: defines this servlet's request-processing infrastructure -->
<!-- Enables the Spring MVC @Controller programming model -->
<annotation-driven />

<!-- Handles HTTP GET requests for /resources/** by efficiently serving up static resources (without using a controller to handle each request) -->
<resources mapping="/resources/**" location="/resources/" />

<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->
<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>

<context:component-scan base-package="chap09" />
```

Static path 설정

ViewResolver 설정

Quickstart (설정)

- MvcConfig.java (code based spring web mvc configuration)

Spring Web MVC 설정 활성화 → 내부적으로 다양한 Bean 활성화

```
@Configuration
@EnableWebMvc
public class MvcConfig implements WebMvcConfigurer {

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurator) {
        configurator.enable();
    }

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.jsp("/WEB-INF/view/", ".jsp");
    }

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("/resources/")
            .setCachePeriod(3600)
            .resourceChain(true);
    }
}
```

리소스 처리 서블릿 설정

ViewResolver 설정

Static path 설정

Quickstart (설정)

- web.xml

- 한글 지원을 위해 Character Encoding Filter 적용

```
<filter>
  <filter-name>characterEncodingFilter</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>
  <init-param>
    <param-name>forceEncoding</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>characterEncodingFilter</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

Quickstart (Controller + View 구현)

▪ HelloController.java

```
@Controller
public class HelloController {

    @GetMapping("/hello")
    public String hello(Model model,
        @RequestParam(value = "name", required = false) String name) {
        model.addAttribute("greeting", "안녕하세요, " + name);
        return "hello";
    }
}
```

▪ hello.jsp

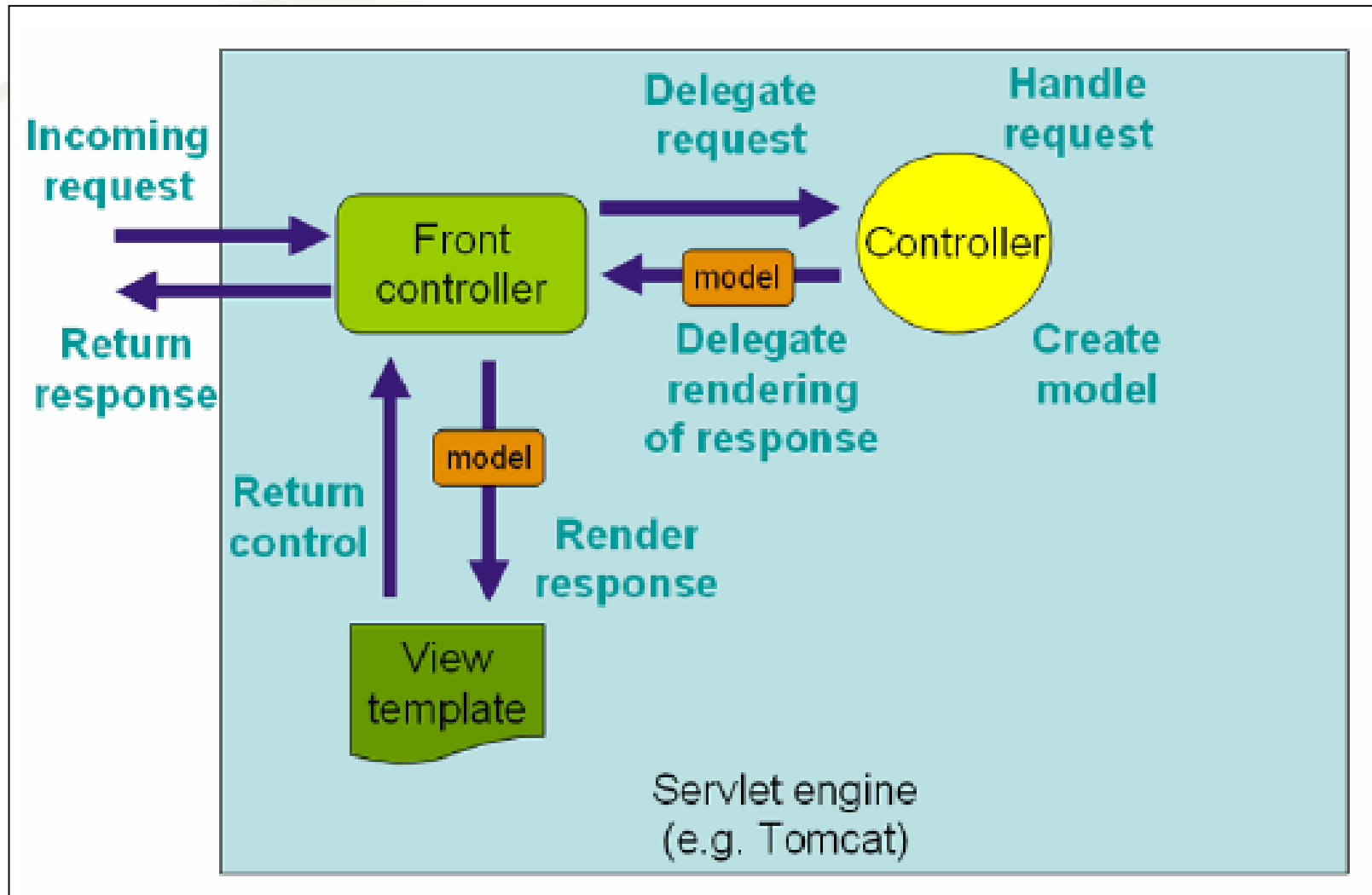
```
<%@ page contentType="text/html; charset=utf-8" %>

<!DOCTYPE html>
<html>
    <head>
        <title>Hello</title>
    </head>
    <body>
        인사말: ${greeting}
    </body>
</html>
```

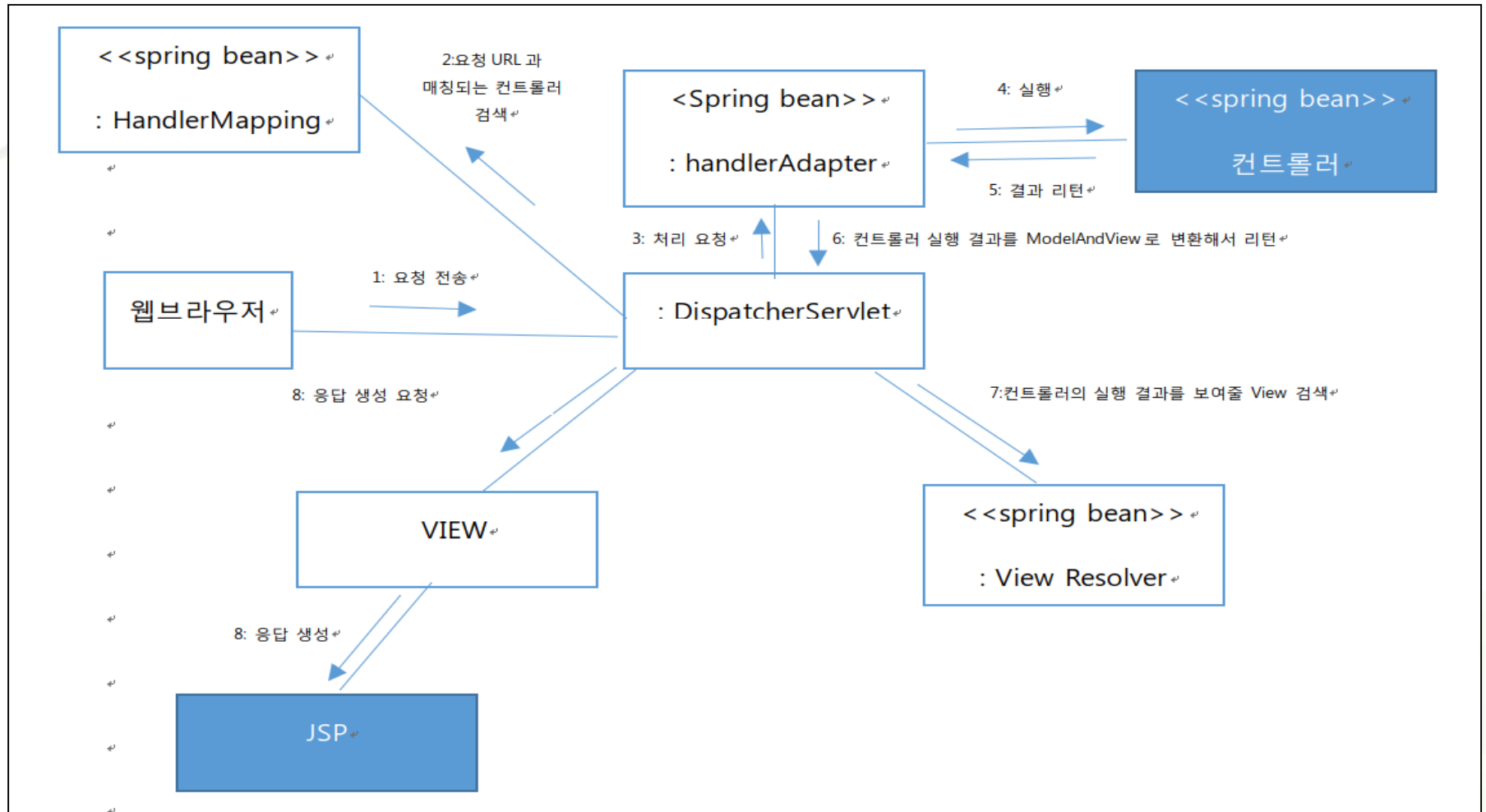
Quickstart (실행)



FrontController Based MVC Pattern



Spring MVC



WebApplicationContext

스프링 MVC

- Front Controller Pattern 기반 구현
- DispatcherServlet을 FrontController 구현체로 제공
- DispatcherServlet 생성과 함께 WebApplicationContext 객체 생성
- web.xml 및 스프링 bean 설정 파일의 내용에 따라 동작

요청 처리 과정 요약

- DispatcherServlet 요청 수신

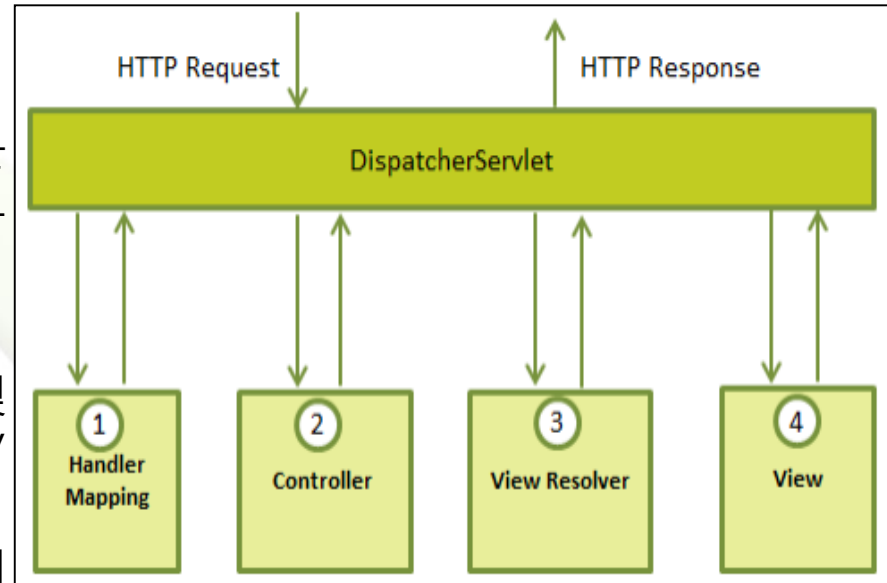
- DispatcherServlet은 HandlerMapping을 통해 적절한 Controller를 선택하고 Controller에 요청 위임

- Controller는 Model 영역의 객체 생성 및 호출 후 Model 객체 생성 / 데이터 할당 / 뷰이름 지정한 후 FrontController에 반환
 - 일반적으로 ModelAndView 형식의 객체 사용해서 결과 반환

- DispatcherServlet은 Controller의 반환 값에 따라 ViewResolver를 이용해서 View를 결정하고 호출

- 호출된 View는 전달된 Model 객체를 이용해서 화면을 구성하고 반환

- DispatcherServlet은 View의 반환 결과를 요청 영역에 응답



스프링 컨트롤러

- DispatcherServlet에서 전달된 개별 요청을 처리하는 객체
- 스프링 IoC 컨테이너에서 관리
- 스프링 3.0 버전부터 Annotation 기반 컨트롤러 클래스 구현 권장

컨트롤러 구현

■ 요청 매핑

@Controller 어노테이션으로 빈 설정 등록

@Controller

```
public class RegisterController {  
    @RequestMapping("/register/step1")  
    public String handleStep1() {  
        return "register/step1";  
    }  
}
```

@RequestMapping 어노테이션으로 요청과 컨트롤러 매핑

@Controller

@RequestMapping("/register")

```
public class RegisterController {  
    @RequestMapping("/step1")  
    public String handleStep1() {  
        return "register/step1";  
    }  
}
```

/WEB-INF/views/register.jsp 페이지를 뷰로 사용
(/WEB-INF/views + **register/step1** + .jsp)

컨트롤러 구현

■ 전송 방식 제어

- Get, Post 등 전송 방식을 구분해서 요청 컨트롤러 매핑 설정 가능

POST/GET 방식 요청에 매핑

POST 방식 요청에 매핑

```
@RequestMapping("/register/step1")
```

```
public String handleStep1() {  
    return "register/step1";  
}
```

```
// @RequestMapping("/register/step2", method = RequestMethod.POST)
```

```
@PostMapping("/register/step2")
```

```
public String handleStep2(  
    @RequestParam(value = "agree", defaultValue = "false") Boolean agree,  
    Model model) {  
    model.addAttribute("registerRequest", new RegisterRequest());  
    return "register/step2";  
}
```

```
// @RequestMapping("/register/step2", method = RequestMethod.GET)
```

```
@GetMapping("/register/step2")
```

```
public String handleStep2Get() {  
    return "redirect:/register/step1";  
}
```

GET 방식 요청에 매핑

컨트롤러 구현

▪ redirect 처리

```
@GetMapping("/register/step2")  
public String handleStep2Get() {  
    return "redirect:/register/step1";  
}
```

요청 매핑 구성 요소

- web.xml 파일에 등록된 DispatcherServlet의 서블릿 매핑 url

```
<servlet-mapping>  
  <servlet-name>customer</servlet-name>  
  <url-pattern>/customer/*</url-pattern>  
</servlet-mapping>
```

- @Controller 어노테이션이 지정된 컨트롤러 클래스의 @RequestMapping에 지정된 경로

- 생략되면 @RequestMapping(value="/")

```
@Controller  
@RequestMapping(value="/")  
public class CustomerController {
```

- 컨트롤러 클래스에 포함된 메서드의 @RequestMapping에 지정된 경로

```
@RequestMapping(value="edit.do", method=RequestMethod.GET)  
public String edit(Model model) {
```

- 최종 경로

http://.../customer/edit.do

컨트롤러 구현

■ 요청 데이터 읽기

- 클라이언트가 전달하는 요청 데이터, 헤더 등의 정보를 수신할 수 있도록 다양한 전달인자 형식을 통해 데이터 전달

■ 요청 데이터 매핑 전달인자 종류

종류	설명
모델	<ul style="list-style-type: none">Model, ModelMap, Map컨트롤러에서 데이터를 저장하고 뷰로 전달되는 용도의 전달인자 (클라이언트가 전달하는 데이터 수신 기능은 없음)
@ModelAttribute	<ul style="list-style-type: none">사용자 정의 객체 모델 지정 (DTO를 이용한 데이터 수신)어노테이션 생략 가능
@PathVariable	<ul style="list-style-type: none">@RequestMapping에 지정된 URL 중 {}에 명시된 경로 변수<code>@RequestMapping("/customer/{name} ")</code> <code>handler(@RequestParam("name") String name) { ...</code>

컨트롤러 구현

■ 요청 데이터 매핑 전달인자 종류 (계속)

종류	설명
@RequestParam	<ul style="list-style-type: none">• 개별 Http 요청 패러미터 저장하는 전달인자 지정• 어노테이션 생략 가능• 모든 요청을 일괄 수신하기 위해 Map<String, String> 사용
@RequestBody	<ul style="list-style-type: none">• Http 요청의 본문을 저장하는 전달인자 지정
HttpServletRequest, HttpServletResponse	<ul style="list-style-type: none">• 일반 서블릿에 전달되는 요청, 응답 객체
HttpSession	<ul style="list-style-type: none">• 일반 서블릿에 전달되는 세션 객체
Locale	<ul style="list-style-type: none">• Locale Resolver가 결정한 Locale 정보
스트림	<ul style="list-style-type: none">• InputStream, Reader, OutputStream, Writer• 요청 및 응답에 대응하는 저수준 스트림 객체
@RequestHeader	<ul style="list-style-type: none">• Http 헤더 정보를 전달인자에 매핑
@Cookievalue	<ul style="list-style-type: none">• Http 쿠키 값을 전달인자에 매핑

컨트롤러 구현

■ 요청 데이터 읽기

```
<body>
  <h2>약관</h2>
  <p>약관 내용</p>
  <form action="step2" method="post">
    <label>
      <input type="checkbox" name="agree" value="true"> 약관 동의
    </label>
    <input type="submit" value="다음 단계" />
  </form>
</body>
```

```
// @RequestMapping("/register/step2", method = RequestMethod.POST)
@PostMapping("/register/step2")
public String handleStep2(
    @RequestParam(value = "agree", defaultValue = "false") Boolean agree,
    Model model) {
    if (!agree) {
        return "register/step1";
        // return "redirect:register/step1";
    }
    model.addAttribute("registerRequest", new RegisterRequest());
    return "register/step2";
}
```

속성	타입	설명
value	String	요청 파라미터 이름
required	boolean	필수 여부. true인 경우 값이 전달되지 않으면 예외 발생
default	String	값이 전달되지 않으면 사용할 값

컨트롤러 구현

- 커맨드 객체 사용해서 요청 데이터 읽기 → 스칼라 변수에 매핑

```
<form action="step3">
<div>
<div>
<div>이메일:<br>
<input type="text" name="email" />
</div>
</div>
<div>이름:<br>
<input type="text" name="name" />
</div>
<div>비밀번호:<br>
<input type="password" name="password" />
</div>
<div>비밀번호 확인:<br>
<input type="password" name="confirmPassword" />
</div>
<div>
<input type="submit" value="가입 완료">
</div>
</div>
</form>
```

```
public class RegisterRequest {
    private String email;
    private String password;
    private String confirmPassword;
    private String name;
}
```

```
@PostMapping("/register/step3")
public String handleStep3(String email, String password,
    String confirmPassword, String name) {
    RegisterRequest req = new RegisterRequest();
    req.setEmail(email);
    req.setPassword(password);
    req.setConfirmPassword(confirmPassword);
    req.setName(name);
}
```


컨트롤러 구현

- 커맨드 객체 사용해서 요청 데이터 읽기 → 커맨드 객체에 매핑

```
<form action="step3">
<p>
  <label>이메일:<br>
  <input type="text" name="email" />
</label>
</p>
<p>
  <label>이름:<br>
  <input type="text" name="name" />
</label>
</p>
<p>
  <label>비밀번호:<br>
  <input type="password" name="password" />
</label>
</p>
<p>
  <label>비밀번호 확인:<br>
  <input type="password" name="confirmPassword" />
</label>
</p>
<input type="submit" value="가입 완료">
</form>
```

```
public class RegisterRequest {

    private String email;
    private String password;
    private String confirmPassword;
    private String name;
}
```

```
@PostMapping("/register/step3")
public String handleStep3(RegisterRequest regReq) {
    try {
        memberRegisterService.register(regReq);
        return "register/step3";
    } catch (DuplicateMemberException ex) {
        return "register/step2";
        //return "redirect:/register/step2";
    }
}
```

컨트롤러 구현

■ 중첩 커맨드 객체 매핑

```
<form method="post">

<!-- question and response tags will go here -->

<p>
    <label>응답자 위치:<br>
    <input type="text" name="res.location">
</label>
</p>
<p>
    <label>응답자 나이:<br>
    <input type="text" name="res.age">
</label>
</p>
<input type="submit" value="전송">
</form>
```

```
public class Respondent {

    private int age;
    private String location;
```

```
public class AnsweredData {

    private List<String> responses;
    private Respondent res;
```

```
@PostMapping
public String submit(@ModelAttribute("ansData") AnsweredData data) {
    return "survey/submitted";
}
```

컨트롤러 구현

■ 컬렉션 커맨드 객체 매핑

```
<form method="post">
<c:forEach var="q" items="${questions}" varStatus="status">
<p>
    ${status.index + 1}. ${q.title}<br/>
    <c:if test="${q.choice}">
        <c:forEach var="option" items="${q.options}">
            <label><input type="radio"
                name="responses[${status.index}]" value="${option}">
                ${option}</label>
        </c:forEach>
    </c:if>
    <c:if test="${! q.choice }">
        <input type="text" name="responses[${status.index}]">
    </c:if>
</p>
</c:forEach>

<!-- nested property tags will go here -->
```

```
public class Respondent {

    private int age;
    private String location;
```

```
public class AnsweredData {

    private List<String> responses;
    private Respondent res;
```

```
@PostMapping
public String submit(@ModelAttribute("ansData") AnsweredData data) {
    return "survey/submitted";
}
```

컨트롤러 구현

▪ View에서 커맨드 객체 사용

```
@PostMapping("/register/step2")
public String handleStep2(
    @RequestParam(value = "agree", defaultValue = "false") Boolean agree,
    Model model) {
    if (!agree) {
        return "register/step1";
        // return "redirect:register/step1";
    }
    model.addAttribute("registerRequest", new RegisterRequest());
    return "register/step2";
}
```

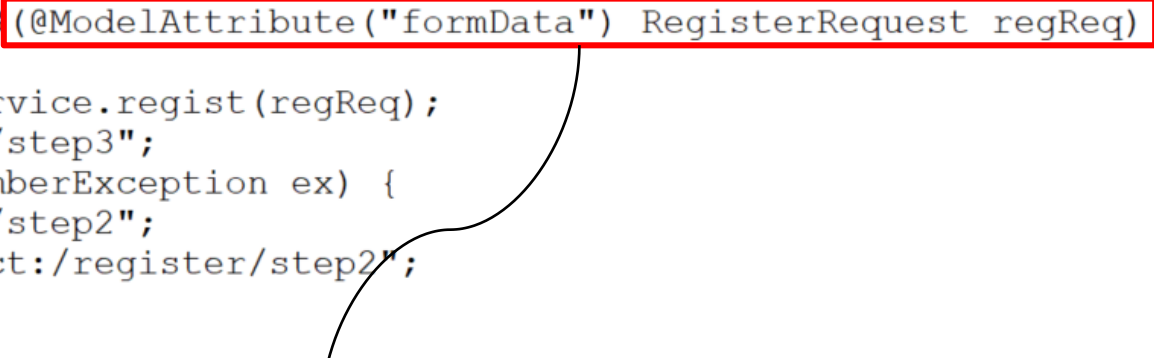
`<p>`
 `${registerRequest.name}님`
 회원 가입을 완료했습니다.
`</p>`

`<p><a href="<c:url value='/main'/>">[첫 화면 이동]</p>`

컨트롤러 구현

■ 커맨드 객체의 이름 변경

```
@PostMapping("/register/step3")
public String handleStep3(@ModelAttribute("formData") RegisterRequest regReq) {
    try {
        memberRegisterService.regist(regReq);
        return "register/step3";
    } catch (DuplicateMemberException ex) {
        return "register/step2";
        //return "redirect:/register/step2";
    }
}
```



@ModelAttribute를 사용해서 이름 변경할 경우
→ request.getParameter("formData")로 커맨드 객체 접근

@ModelAttribute를 사용해서 이름을 변경하지 않는 경우
→ request.getParameter("regReq")로 커맨드 객체 접근

컨트롤러 구현

■ 커맨드 객체와 스프링 폼 연동

- 스프링 폼 태그 라이브러리를 사용해서 커맨드 객체 바인딩

```
@PostMapping("/register/step2")  
public String handleStep2(Boolean agree, Model model) {  
    model.addAttribute("registerRequest", new RegisterRequest());  
    return "register/step2";  
}
```

```
<form:form action="step3" modelAttribute="registerRequest">
```

```
<p>
```

```
<label>이메일:<br>
```

```
<form:input path="email" />
```

```
</label>
```

```
</p>
```

```
<p>
```

```
<label>이름:<br>
```

```
<form:input path="name" />
```

```
</label>
```

```
</p>
```

```
<p>
```

```
<label>비밀번호:<br>
```

```
<form:password path="password" />
```

```
</label>
```

```
</p>
```

```
<p>
```

```
<label>비밀번호 확인:<br>
```

```
<form:password path="confirmPassword" />
```

```
</label>
```

```
</p>
```

```
<input type="submit" value="가입 완료">
```

```
</form:form>
```

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

컨트롤러 구현

- 컨트롤러 구현 없는 경로 매핑

- 컨트롤러에서 특별한 처리 작업이 없는 단순한 JSP 요청의 경우 컨트롤러를 만들지 않고 JSP를 연결할 수 있음

code based configuration (MvcConfig.java)

```
@Configuration
@EnableWebMvc
public class MvcConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/main").setViewName("main");
    }
}
```

xml based configuration (servlet-context.xml)

```
<view-controller path="/main" view-name="main" />
```

컨트롤러 구현

■ 요청 처리기 메서드 반환 타입

- 요청 처리기 메서드는 논리적 뷰와 모델을 DispatcherServlet에게 반환
- 반환 형식 종류

종류	설명
자동 추가되는 객체 모델	<ul style="list-style-type: none">• Model, ModelMap, Map 타입 전달인자• @ModelAttribute 어노테이션이 명시적/암시적 지정된 전달인자
@ModelAttribute	<ul style="list-style-type: none">• 메서드에 @ModelAttribute 어노테이션이 지정된 반환 타입 (이 때 뷰 이름은 메서드 이름에 일치)
ModelAndView	<ul style="list-style-type: none">• 뷰와 반환 데이터를 저장할 수 있는 전용 타입
String	<ul style="list-style-type: none">• 뷰 이름으로 사용될 문자열 (이 때 모델 데이터는 다른 방법으로 전달하도록 구현)
void	<ul style="list-style-type: none">• 뷰 이름은 메서드 이름에 일치• 모델 데이터는 다른 방법으로 전달하도록 구현
View	<ul style="list-style-type: none">• 사용자 정의 구현 내용을 포함하는 뷰 객체 반환
@ResponseBody	<ul style="list-style-type: none">• HTTP 응답 메시지 본문을 문자열로 반환

JSTL 뷰 구현 - 컨트롤러와 뷰 사이의 데이터 전달

- Controller에서 ModelAttribute로 설정한 데이터는 JSP에서 사용할 수 있도록 HttpServletRequest 내장 객체에 저장되어 JSP에 전달됨
- 전달된 데이터는 스크립트 코드 블록의 자바 코드, EL 등을 통해 사용됨

```
@RequestMapping(value="/edit.do", method=RequestMethod.POST)
public String add(
    @Valid @ModelAttribute("customer") CustomerModel model,
    BindingResult bindingResult) {
    if(bindingResult.hasErrors()) {
        return "edit";
    }
    try {
        customerService.saveCustomer(model.buildDomain());
    }
    catch(Exception e) {
        return "forward:/error.do";
    }
    return "result";
}
```

```
<h1>고객 등록 정보</h1>
이름 : ${customer.name} <br>
주소 : ${customer.address} <br>
이메일 : ${customer.email}
```

```
<h1>고객 등록 정보</h1>
<%
Customer customer =
    (Customer)request.getAttribute("customer");
%>
이름 : <%= customer.getName() %> <br>
주소 : <%= customer.getAddress() %> <br>
이메일 : <%= customer.getEmail() %>
```

JSTL 뷰 구현 - 스프링 지원 폼 태그 라이브러리

- Controller 에서 전달된 데이터를 JSP의 HTML 요소의 속성의 값에 자동 바인딩 처리

종류	설명
form	HTML 폼 태그를 렌더링하고 다른 폼 태그에 바인딩 경로 제공
input	text 타입의 HTML input 요소를 렌더링
password	password 타입의 HTML input 요소를 렌더링
hidden	hidden 타입의 HTML input 요소를 렌더링
select	HTML select 요소를 렌더링
option	HTML select 요소 안에서 option 요소를 렌더링
options	HTML select 요소 안에서 option 요소의 컬렉션을 렌더링
radiobutton	radiobutton 타입의 HTML input 요소를 렌더링
radiobuttons	radiobutton 타입의 HTML input 요소의 컬렉션을 렌더링
checkbox	checkbox 타입의 HTML input 요소를 렌더링
checkboxes	checkbox 타입의 HTML input 요소의 컬렉션을 렌더링
textarea	HTML textarea 요소 렌더링
errors	사용자에게 바인딩 및 검증 에러 표시
label	HTML label 요소를 렌더링 하고 input 요소와 연관
button	HTML button 요소를 렌더링

JSTL 뷰 구현 - 스프링 지원 폼 태그 라이브러리

■ 폼 태그 애트리뷰트

종류	설명
path	바인딩할 모델 객체의 속성 지정
readonly	폼 요소의 읽기 전용 여부 지정
disabled	폼 요소 활성화 여부 지정
cssClass	폼 요소에 적용할 css 클래스 지정
cssErrorClass	바인딩 또는 검증 에러 정보를 표시하는 데 사용할 css 클래스 지정
id	Javascript에서 식별자로 사용할 폼 요소의 id 지정

■ 컬렉션 타입 태그 애트리뷰트

종류	설명
items	렌더링할 모델 객체 컬렉션 속성 지정
itemValue	렌더링할 단일 요소의 값 지정
itemLabel	렌더링할 단일 요소의 제목 지정
multiple	다중 선택 허용 여부 지정

폼 태그 예제

```
@RequestMapping(value="/edit.do", method=RequestMethod.POST)
public String add(
    @Valid @ModelAttribute("customer") CustomerModel model,
    BindingResult bindingResult) {
    if(bindingResult.hasErrors()) {
        return "edit";
    }
    try {
        customerService.saveCustomer(model.buildDomain());
    }
    catch(Exception e) {
        return "forward:/error.do";
    }
    return "result";
}
```

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

```
<h3>고객 정보 입력</h3>
```

```
<fieldset>
```

```
<form:form method="post" modelAttribute="customer">
```

```
    <form:label path="name" cssErrorClass="error">이름 : </form:label>
```

```
    <form:input type="text" path="name" />
```

```
    <form:errors path="name" cssClass="error"/> <br>
```

```
    <form:label path="address" cssErrorClass="error">주소 : </form:label>
```

```
    <form:input type="text" path="address" size="60" />
```

```
    <form:errors path="address" cssClass="error"/> <br>
```

```
    <form:label path="email" cssErrorClass="error">이메일 : </form:label>
```

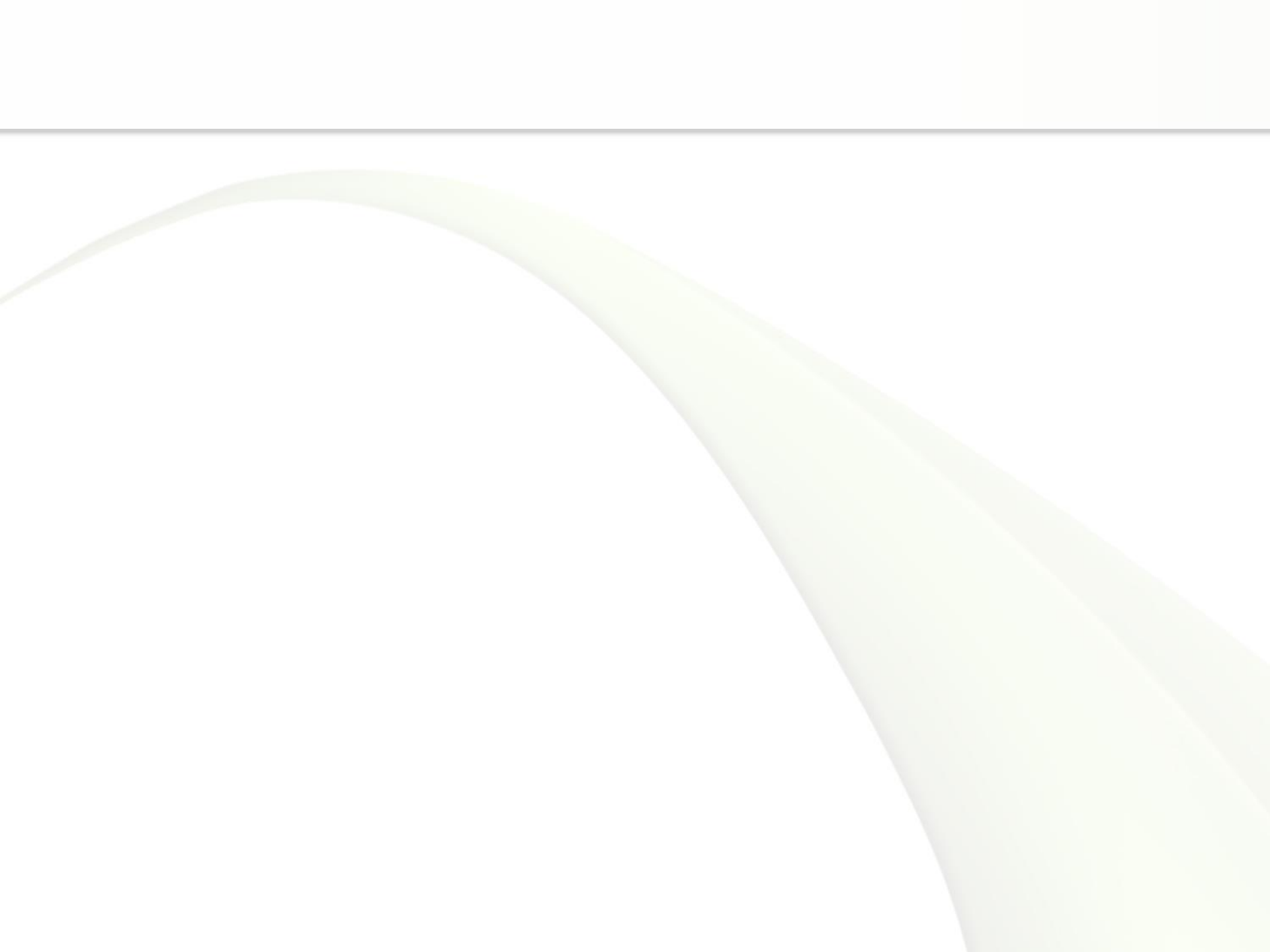
```
    <form:input type="text" path="email" />
```

```
    <form:errors path="email" cssClass="error"/> <br>
```

```
    <input type="submit" value="저장" />
```

```
</form:form>
```

```
</fieldset>
```



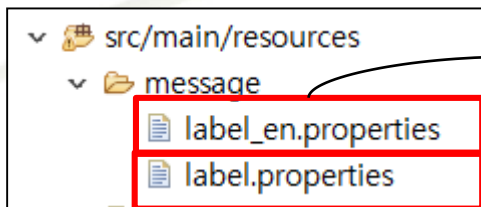
스프링 국제화

- 스프링은 표준 JSTL 태그 라이브러리와 동일한 방식으로 리소스 번들과 메시지 포맷을 사용하여 국제화 지원
- 리소스 번들을 쉽게 사용할 수 있도록 추상화 메시지 소스 제공

종류	설명
ResourceBundleMessageSource	리소스 파일을 반드시 /WEB-INF/classes 폴더 또는 resources 폴더에 저장하도록 제약
ReloadableResourceBundleMessageSource	리소스 파일의 위치를 자유롭게 설정

스프링 국제화 구현

▪ .properties 리소스 파일 작성



member.register=register

term=약관
term.agree=약관동의
next.btn=다음단계

member.info=회원정보
email=이메일
name=이름

password=비밀번호
password.confirm=비밀번호 확인
register.btn=가입 완료

register.done={0} ({1}), registration successfully completed

go.main=move to main

required=required
bad.email=invalid email
duplicate.email=duplicated email
nomatch.confirmPassword=password not mached

member.register=회원가입

term=약관
term.agree=약관동의
next.btn=다음단계

member.info=회원정보
email=이메일
name=이름

password=비밀번호
password.confirm=비밀번호 확인
register.btn=가입 완료

register.done={0}님 ({1}), 회원 가입을 완료했습니다.

go.main=메인으로 이동

required=필수항목입니다.
bad.email=이메일이 올바르지 않습니다.
duplicate.email=중복된 이메일입니다.
nomatch.confirmPassword=비밀번호와 확인이 일치하지 않습니다.

스프링 국제화 구현

- 메시지 소스 빈 등록 (빈 설정 파일)
 - code based configuration

```
@Bean
public MessageSource messageSource() {
    ResourceBundleMessageSource ms =
        new ResourceBundleMessageSource();
    ms.setBasenames("message.label");
    ms.setDefaultEncoding("UTF-8");
    return ms;
}
```

bean name은 반드시 messageSource

- xml based configuration

```
<beans:bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <beans:property name="basenames" value="message.label" />
    <beans:property name="defaultEncoding" value="UTF-8" />
</beans:bean>
```


스프링 국제화 구현

■ 스프링 태그 라이브러리를 이용한 메시지 표시

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

```
<h2><spring:message code="term" /></h2>
<p>약관 내용</p>
<form action="step2" method="post">
<label>
  <input type="checkbox" name="agree" value="true">
  <spring:message code="term.agree" />
</label>
<input type="submit" value="<spring:message code="next.btn" />" />
</form>
```

term=약관

term.agree=약관동의

next.btn=다음단계

member.info=회원정보

email=이메일

name=이름

password=비밀번호

password.confirm=비밀번호 확인

register.btn=가입 완료

register.done={0}님 ({1}), 회원 가입을 완료했습니다.

스프링 국제화 구현

■ 스프링 태그 라이브러리를 이용한 메시지 표시

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

```
<h2><spring:message code="member.info" /></h2>
<form:form action="step3" modelAttribute="registerRequest">
<p>
  <label><spring:message code="email" />:<br>
  <form:input path="email" />
</label>
</p>
<p>
  <label><spring:message code="name" />:<br>
  <form:input path="name" />
</label>
</p>
<p>
  <label><spring:message code="password" />:<br>
  <form:password path="password" />
</label>
</p>
<p>
  <label><spring:message code="password.confirm" />:<br>
  <form:password path="confirmPassword" />
</label>
</p>
<input type="submit" value="<spring:message code="register.btn" />" />
</form:form>
```

member.info=회원정보
email=이메일
name=이름
password=비밀번호
password.confirm=비밀번호 확인
register.btn=가입 완료

스프링 국제화 구현

■ 스프링 태그 라이브러리를 이용한 메시지 표시

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

```
<head>
  <title><spring:message code="member.register" /></title>
</head>
<body>
  <p>
    <spring:message code="register.done">
      <spring:argument value="${registerRequest.name}" />
      <spring:argument value="${registerRequest.email}" />
    </spring:message>
  </p>
  <p>
    <a href="<c:url value='/main' />">
      [<spring:message code="go.main" />]
    </a>
  </p>
</body>
```

register.btn=가입 완료

register.done={0}님 ({1}), 회원 가입을 완료했습니다.

go.main=메인으로 이동

데이터 검증

- Validator, Errors 인터페이스 사용해서 검증 객체 구현

```
public class RegisterRequestValidator implements Validator {  
    private static final String emailRegExp =  
        "^[_A-Za-z0-9-\\+](\\.[_A-Za-z0-9-]+)*@" +  
        "[A-Za-z0-9-]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})$";  
    private Pattern pattern;  
  
    public RegisterRequestValidator() {}  
  
    public boolean supports(Class<?> clazz) {}  
  
    @Override  
    public void validate(Object target, Errors errors) {  
        System.out.println("RegisterRequestValidator#validate(): " + this);  
        RegisterRequest regReq = (RegisterRequest) target;  
        if (regReq.getEmail() == null || regReq.getEmail().trim().isEmpty()) {  
            errors.rejectValue("email", "required");  
        } else {  
            Matcher matcher = pattern.matcher(regReq.getEmail());  
            if (!matcher.matches()) {  
                errors.rejectValue("email", "bad");  
            }  
        }  
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name", "required");  
        ValidationUtils.rejectIfEmpty(errors, "password", "required");  
        ValidationUtils.rejectIfEmpty(errors, "confirmPassword", "required");  
        if (!regReq.getPassword().isEmpty()) {  
            if (!regReq.isPasswordEqualToConfirmPassword()) {  
                errors.rejectValue("confirmPassword", "nomatch");  
            }  
        }  
    }  
}
```

요청 데이터 검증

■ 검증 객체 직접 사용

```
@PostMapping("/register/step3")
public String handleStep3(RegisterRequest regReq, Errors errors) {
    new RegisterRequestValidator().validate(regReq, errors);
    if (errors.hasErrors())
        return "register/step2";

    try {
        memberRegisterService.regist(regReq);
        return "register/step3";
    } catch (DuplicateMemberException ex) {
        errors.rejectValue("email", "duplicate");
        return "register/step2";
    }
}
```

요청 데이터 검증

■ 컨트롤러 수준에서 검증 자동화

```
@PostMapping("/register/step3")
public String handleStep3(@Valid RegisterRequest regReq, Errors errors) {
    if (errors.hasErrors())
        return "register/step2";

    try {
        memberRegisterService.regist(regReq);
        return "register/step3";
    } catch (DuplicateMemberException ex) {
        errors.rejectValue("email", "duplicate");
        return "register/step2";
    }
}
```

```
@InitBinder
protected void initBinder(WebDataBinder binder) {
    binder.addValidators(new RegisterRequestValidator());
}
```

요청 데이터 검증

■ 전역 수준에서 검증 자동화

```
@Configuration
@EnableWebMvc
public class MvcConfig implements WebMvcConfigurer {

    @Override
    public Validator getValidator() {
        return new RegisterRequestValidator();
    }

    public void configureDefaultServletHandling(
    public void configureViewResolvers(ViewResolverRegistry registry) {
    public void addViewControllers(ViewControllerRegistry registry) {
    public MessageSource messageSource() {
}
```

```
@PostMapping("/register/step3")
public String handleStep3(@Valid RegisterRequest regReq, Errors errors) {
    if (errors.hasErrors())
        return "register/step2";

    try {
        memberRegisterService.regist(regReq);
        return "register/step3";
    } catch (DuplicateMemberException ex) {
        errors.rejectValue("email", "duplicate");
        return "register/step2";
    }
}
```


요청 데이터 검증

■ Bean Validation을 활용한 검증

```
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
  <version>1.1.0.Final</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>5.4.2.Final</version>
</dependency>
```

```
public class RegisterRequest {
  @NotBlank
  @Email
  private String email;
  @Size(min = 6)
  private String password;
  @NotEmpty
  private String confirmPassword;
  @NotEmpty
  private String name;
}
```

```
@PostMapping("/register/step3")
public String handleStep3(@Valid RegisterRequest regReq, Errors errors) {
  if (errors.hasErrors())
    return "register/step2";

  try {
    memberRegisterService.regist(regReq);
    return "register/step3";
  } catch (DuplicateMemberException ex) {
    errors.rejectValue("email", "duplicate");
    return "register/step2";
  }
}
```


빈 검증 어노테이션

■ 주요 JSR-303 표준 어노테이션

종류	설명
@Null	필드 값이 null이 아니면 검증 실패
@NotNull	필드 값이 null이면 검증 실패
@NotBlank	문자열 필드의 값이 빈문자열이면 검증 실패
@AssertTrue @AssertFalse	boolean 타입의 필드의 true, false 여부 검증
@DecimalMax @DecimalMin	숫자 타입의 필드의 최소/최대 값 지정 double, float은 정밀도 문제로 지원하지 않음
@Digits(integer=, fraction=)	정수 및 소숫점 자릿수 제한
@Future @Past	날짜 타입의 필드가 미래 또는 과거인지 검증
@Max(value=) @Min(value=)	최대 또는 최소 값 지정
@Pattern(regexp=, flag=) @Patterns({@Pattern()})	필드 값이 정규 표현식과 일치하지 않으면 검증 실패
@Size(min=, max=)	배열, 컬렉션, 맵 타입 필드에 포함된 요소 갯수의 최대/최소 값 지정

빈 검증 어노테이션

- hibernate-validator 구현체에서 제공하는 주요 어노테이션

종류	설명
@Length(min=, max=)	문자열 타입 필드의 최소, 최대 길이 지정
@NotEmpty	문자열 타입의 필드가 null 또는 빈문자열이면 검증 실패
@Range(min=, max=)	숫자 또는 숫자 값을 표시하는 문자열의 값 범위 지정
@Valid	관련된 객체를 재귀적으로 검증 컬렉션은 요소들을 재귀적 검증 / 맵은 값 요소들을 재귀적으로 검증
@Email	문자열 타입의 필드가 이메일 주소 명세를 충족하는지 검증
@CreditCardNumber	문자열 타입의 필드가 신용카드 번호인지 검증

요청 데이터 검증

■ 에러 메시지 출력

```
<body>
  <h2><spring:message code="member.info" /></h2>
  <form:form action="step3" modelAttribute="registerRequest">
    <p>
      <label><spring:message code="email" />:<br>
      <form:input path="email" />
      <form:errors path="email"/>
    </label>
  </p>
  <p>
    <label><spring:message code="name" />:<br>
    <form:input path="name" />
    <form:errors path="name"/>
  </label>
</p>
<p>
  <label><spring:message code="password" />:<br>
  <form:password path="password" />
  <form:errors path="password"/>
</label>
</p>
<p>
  <label><spring:message code="password.confirm" />:<br>
  <form:password path="confirmPassword" />
  <form:errors path="confirmPassword"/>
</label>
</p>
  <input type="submit" value="<spring:message code="register.btn" />" />
</form:form>
</body>
```

label.properties file

required=필수항목입니다.
bad.email=이메일이 올바르지 않습니다.
duplicate.email=중복된 이메일입니다.
nomatch.confirmPassword=비밀번호와 확인이 일치하지 않습니다.

인터셉터

- 다수의 컨트롤러에 동일한 기능을 적용할 수 있는 도구
 - 컨트롤러의 요청 처리 전/후에 공통 기능 적용 가능
- 필터는 모든 요청을 대상으로 하지만 인터셉터는 컨트롤러의 요청 처리만을 대상으로 적용

인터셉터

■ 구현

```
public class AuthCheckInterceptor implements HandlerInterceptor {  
  
    @Override  
    public boolean preHandle(  
        HttpServletRequest request,  
        HttpServletResponse response,  
        Object handler) throws Exception {  
        HttpSession session = request.getSession(false);  
        if (session != null) {  
            Object authInfo = session.getAttribute("authInfo");  
            if (authInfo != null) {  
                return true;  
            }  
        }  
        response.sendRedirect(request.getContextPath() + "/login");  
        return false;  
    }  
}
```

code based configuration

xml based configuration

```
@Override  
public void addInterceptors(InterceptorRegistry registry) {  
    registry.addInterceptor(authCheckInterceptor())  
        .addPathPatterns("/edit/**")  
        .excludePathPatterns("/edit/help/**");  
}  
  
@Bean  
public AuthCheckInterceptor authCheckInterceptor() {  
    return new AuthCheckInterceptor();  
}
```

```
<interceptors>  
  <interceptor>  
    <mapping path="/edit/**" />  
    <exclude-mapping path="/edit/help/**" />  
    <beans:bean class="interceptor.AuthCheckInterceptor" />  
  </interceptor>  
</interceptors>
```

날짜 형식 변환

```
public class ListCommand {  
    @DateTimeFormat(pattern = "yyyyMMddHH")  
    private LocalDateTime from;  
    @DateTimeFormat(pattern = "yyyyMMddHH")  
    private LocalDateTime to;  
}
```

```
@RequestMapping("/members")  
public String list(  
    @ModelAttribute("cmd") ListCommand listCommand,  
    Errors errors, Model model) {  
    if (errors.hasErrors()) {  
        return "member/memberList";  
    }  
    if (listCommand.getFrom() != null && listCommand.getTo() != null) {  
        List<Member> members = memberDao.selectByRegdate(  
            listCommand.getFrom(), listCommand.getTo());  
        model.addAttribute("members", members);  
    }  
    return "member/memberList";  
}
```

컨트롤러 예외 처리

■ 개별 컨트롤러 수준 예외 처리 구현

```
@Controller
public class MemberDetailController {

    private MemberDao memberDao;

    public void setMemberDao(MemberDao memberDao) {}

    public String detail(@PathVariable("id") Long memId, Model model) {}

    @ExceptionHandler(TypeMismatchException.class)
    public String handleTypeMismatchException() {
        return "member/invalidId";
    }

    @ExceptionHandler(MemberNotFoundException.class)
    public String handleNotFoundException() {
        return "member/noMember";
    }
}
```

■ 전역 컨트롤러 수준 예외 처리 구현

```
@ControllerAdvice("spring")
public class CommonExceptionHandler {

    @ExceptionHandler(RuntimeException.class)
    public String handleRuntimeException() {
        return "error/commonException";
    }
}
```