yonggyo1125 동영상 강의 URL 추가 98ee819 · 2 years ago

Name	Name	Last commit date
..		
images	스프링 MVC : 폼 태그 강의 ...	2 years ago
예제 소스	스프링 MVC 폼 태그 강의 ...	2 years ago
README.md	동영상 강의 URL 추가	2 years ago

README.md

강의 동영상 링크

동영상 링크

스프링 MVC : 요청 매핑, 커맨드 객체, 리다이렉트, 폼 태그, 모델

- 스프링 MVC를 사용해서 웹 어플리케이션을 개발한다는 것은 결국 컨트롤러와 뷰 코드를 구현한다는 것을 뜻한다.
- 대부분의 설정은 개발 초기에 완성된다. 개발이 완료될 때까지 개발자가 만들어야 하는 코드는 컨트롤러와 뷰 코드이다.
- 어떤 컨트롤러를 이용해서 어떤 요청 경로를 처리할지 결정하고 웹 브라우저가 전송한 요청에서 필요한 값을 구하고, 처리 결과를 JSP를 이용해서 보여주면 된다.

프로젝트 준비

- 프로젝트 생성

```
mvn archetype:generate
```

- groupId, artifactId는 적절하게 입력해 줍니다.
- 프로젝트를 위한 폴더가 생성되었다면 그 폴더 하위에 다음 폴더를 생성한다.
    - src/main/java
    - src/main/webapp
    - src/main/webapp/WEB-INF
    - src/main/webapp/WEB-INF/view
- pom.xml : 자바 실습 버전을 최신버전(17)로 변경합니다
- pom.xml : javax.servlet-api, javax.servlet.jsp-api, jstl, spring-webmvc, spring-jdbc, tomcat-jdbc, mysql-connector-java 의존성을 mvnrepository 에서 검색하여 다음과 같이 추가합니다.

```
... 생략
<properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
</properties>
... 생략
<dependencies>
        <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>javax.servlet-api</artifactId>
                <version>4.0.1</version>
                <scope>provided</scope>
        </dependency>
        <dependency>
                <groupId>javax.servlet.jsp</groupId>
                <artifactId>javax.servlet.jsp-api</artifactId>
                <version>2.3.3</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>5.3.21</version>
        </dependency>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-jdbc</artifactId>
                <version>5.3.21</version>
        </dependency>
        <dependency>
                <groupId>org.apache.tomcat</groupId>
                <artifactId>tomcat-jdbc</artifactId>
                <version>9.0.64</version>
        </dependency>
        <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
                <version>8.0.29</version>
        </dependency>
        <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-api</artifactId>
                <version>1.7.36</version>
        </dependency>
        <dependency>
                <groupId>ch.qos.logback</groupId>
                <artifactId>logback-classic</artifactId>
                <version>1.2.11</version>
        </dependency>
... 생략
</dependencies>
```

- 이클립스에서 Import - Existing Maven Projects를 클릭하여 생성된 폴더를 선택하여 생성해 줍니다.
- 프로젝트가 추가되면 생성된 프로젝트에서 마우스 오른쪽 키 -> Properties -> Project Facets -> Dynamic Web Module을 체크한다.
- 다음의 코드를 복사한다 : 학습용 소스 > spring
    - spring 패키지를 생성한 뒤 복사하면 된다.
        - ChangePasswordService.java
        - DuplicateMemberException.java
        - Member.java
        - MemberDao.java
        - MemberNotFoundException.java
        - MemberRegisterService.java
        - RegisterRequest.java

        - WoringIdPasswordException.java

- JdbcTemplate, 트랜잭션, 마이바티스 에서 생성한 데이터베이스를 그대로 사용한다. 생성하지 않았다면 해당 페이지로 이동하여 데이터베이스와 테이블을 생성한다.

> 톰캣에서 Maven Dependency를 인식하지 못하여 실행이 안되는 경우
> 프로젝트 폴더 -> 마우스 오른쪽 버튼 -> Properties -> Deplolyment Assembly -> Add 버튼 -> Maven Dependency 선택 -> Apply 또는 Apply and Close 버튼을 클릭하여 적용한다.

- 두 개의 스프링 설정 파일과 web.xml파일을 작성할 것이다. 먼저 서비스 클래스와 DAO 클래스를 위한 스프링 설정 클래스를 다음과 같이 작성한다. DataSource와 트랜잭션 관련 설정도 포함되어 있다.

- 로그 출력 설정

src/main/resources/logback.xml

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

<configuration>
    <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d %5p %c{2} - %m%n</pattern>
        </encoder>
    </appender>
    <root level="INFO">
        <appender-ref ref="stdout" />
    </root>

    <logger name="org.springframework.jdbc" level="DEBUG" />
</configuration>
```

src/main/java/config/MemberConfig.java

```
package config;

import org.apache.tomcat.jdbc.pool.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManageme

import spring.ChangePasswordService;
import spring.MemberDao;
import spring.MemberRegisterService;

@Configuration
```

```
@EnableTransactionManagement
public class MemberConfig {

	@Bean(destroyMethod = "close")
	public DataSource dataSource() {
		DataSource ds = new DataSource();
		ds.setDriverClassName("com.mysql.cj.jdbc.Driver");
		ds.setUrl("jdbc:mysql://localhost/spring5fs?characterEncodi
		ds.setUsername("spring5");
		ds.setPassword("spring5");
		ds.setInitialSize(2);
		ds.setMaxActive(10);
		ds.setTestWhileIdle(true);
		ds.setMinEvictableIdleTimeMillis(60000 * 3);
		ds.setTimeBetweenEvictionRunsMillis(10 * 1000);
		return ds;
	}

	@Bean
	public PlatformTransactionManager transactionManager() {
		DataSourceTransactionManager tm = new DataSourceTransaction
		tm.setDataSource(dataSource());
		return tm;
	}

	@Bean
	public MemberDao memberDao() {
		return new MemberDao(dataSource());
	}

	@Bean
	public MemberRegisterService memberRegSvc() {
		return new MemberRegisterService(memberDao());
	}

	@Bean
	public ChangePasswordService changePwdSvc() {
		ChangePasswordService pwdSvc = new ChangePasswordService();
		pwdSvc.setMemberDao(memberDao());
		return pwdSvc;
	}
}
```

src/main/java/config/MvcConfig.java

```
package config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.DefaultServletHand
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
```

```
import org.springframework.web.servlet.config.annotation.ResourceHandlerReg
import org.springframework.web.servlet.config.annotation.ViewResolverRegist
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
@EnableWebMvc
public class MvcConfig implements WebMvcConfigurer {

        @Override
        public void configureDefaultServletHandling(DefaultServletHandlerCo
                configurer.enable();
        }

        @Override
        public void addResourceHandlers(ResourceHandlerRegistry registry) {
                registry.addResourceHandler("/**")
                        .addResourceLocations("classpath:/static/");
        }

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

src/main/webapp/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="4.0" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi

    <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.AnnotationConfigWeb
            </param-value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>
                config.MemberConfig
                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>

    <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>

</web-app>
```

요청 매핑 애노테이션을 이용한 경로 매핑

- 웹 어플리케이션을 개발하는 것은 다음 코드를 작성하는 것이다.
    - 특정 요청 URL을 처리할 코드
    - 처리 결과를 HTML과 같은 형식으로 응답하는 코드
- 이 중 첫 번째는 @Controller 애노테이션을 사용한 컨트롤러 클래스를 이용해서 구현한다.
- 컨트롤러 클래스는 요청 매핑 애노테이션을 사용해서 메서드가 처리할 요청 경로를 지정한다. 요청 매핑 애노테이션에는 @RequestMapping, @GetMapping, @PostMapping등이 있다.
- 앞서 HelloController 클래스는 다음과 같이 @GetMapping 애노테이션을 사용해서 "/hello" 요청 경로를 hello() 메서드가 처리하도록 설정했다.

```
import org.springframework.sterotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class HelloController {
```

```
        public String hello(Model model, @RequestParam(value = "name", requ
                model.addAttribute("greeting", "안녕하세요, " + name);
                return "hello";
        }
}
```

- 요청 매핑 애노테이션을 적용한 메서드를 두 개 이상 정의할 수 도 있다.
- 예를 들어 회원가입 과정은 "약관동의 -> 회원 정보 입력 -> 가입완료"인데 각 과정을 위한 URL을 다음과 같이 정할수 있을 것이다.
  - 약관 동의 화면 요청 : /register/step1
  - 회원 정보 입력 화면 : /register/step2
  - 가입 처리 결과 화면 : /register/step3

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

        @RequestMapping("/register/step2")
        public String handleStep2() {
                ...
        }

        @RequestMapping("/register/step3")
        public String handleStep3() {
                ...
        }
}
```

- 이 코드를 보면 각 요청 애노테이션의 경로가 "/register"로 시작한다. 이 경우 다음 코드 처럼 공통되는 부분의 경로를 담은 @RequestMapping 애노테이션을 클래스에 적용하고 각 메서드는 나머지 경로를 값으로 갖는 요청 맺핑 애노테이션을 적용할 수 있다.

```
@Controller
@RequestMapping("/register")  // 각 메서드에 공통되는 경로
public class RegistController {

        @RequestMapping("/step1") // 공통 경로를 제외한 나머지 경로
        public String handleStep1() {
                return "register/step1";
        }

        @RequestMapping("/step2")
```

```
        public String handleStep2() {
                ...
        }
}
```

- 스프링 MVC는 클래스에 적용한 요청 매핑 애노테이션의 경로와 메서드에 적용한 요청 매핑 애노테이션의 경로를 합쳐서 경로를 찾기 때문에 위 코드에서 handleStep1() 메서드가 처리하는 경로는 "/step1"이 아닌 "/register/step1"이 된다.

src/main/java/controller/RegisterController.java

```
package controller;

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

@Controller
public class RegisterController {

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

src/main/webapp/WEB-INF/view/register/step1.jsp

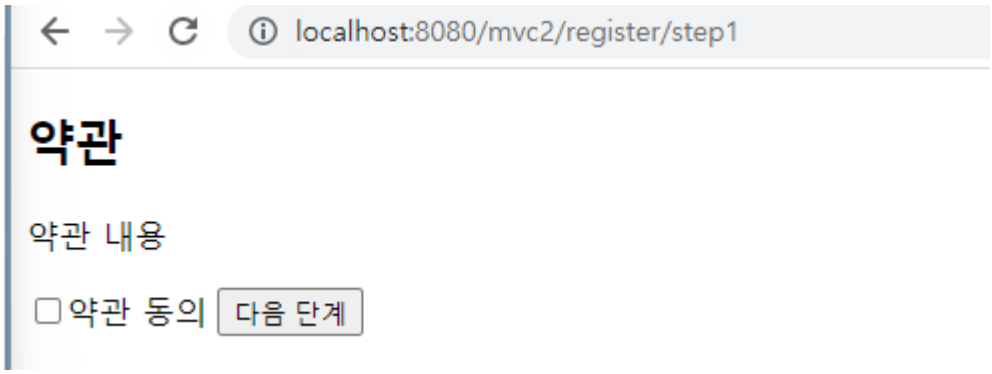
```
<%@ page contentType="text/html; charset=utf-8" %>
<!DOCTYPE html>
<html>
    <head>
        <title>회원가입</title>
    </head>
    <body>
        <h2>약관</h2>
        <p>약관 내용</p>
        <form action="step2" method="post">
            <label>
                <input type="checkbox" name="agree" value="true">약관 동의
            </label>
            <input type="submit" value="다음 단계" />
        </form>
    </body>
</html>
```

- 남은 작업은 Controller.java 파일을 작성하고 이 파일에 RegisterController 클래스를 빈으로 등록하는 것이다. 설정 파일을 같이 작성한다.

src/main/java/config/ControllerConfig.java

```
package config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import controller.RegisterController;

@Configuration
public class ControllerConfig {

	@Bean
	public RegisterController registerController() {
		return new RegisterController();
	}
}
```

- 실행 결과



GET과 POST 구분 : @GetMapping, @PostMapping

- HTML 폼 코드를 보면 전송 방식을 POST로 지정했다. 주로 폼을 전송할 때 POST 방식을 사용하는데, 스프링 MVC는 별도 설정이 없으면 GET과 POST 방식에 상관없이 @RequestMapping에 지정한 경로와 일치하는 요청을 처리한다.
- 만약 POST 방식 요청만 처리하고 싶다면 다음과 같이 @PostMapping 애노테이션을 사용해서 제한할 수 있다.

```
import org.springframework.web.bind.annotation.PostMapping;

@Controller
public class RegisterController {
	@PostMapping("/register/step2")
	public String handleStep2() {
		return "register/step2";
	}
}
```

- 위와 같이 설정하면 handleStep2() 메서드는 POST 방식의 "/register/step2" 요청 경로만 처리하며 GET 방식의 "/register/step2" 요청 경로는 처리하지 않는다.
- 동일하게 @GetMapping 애노테이션을 사용하면 GET 방식만 처리하도록 제한할 수 있다.이 두 애노테이션을 사용하면 다음 코드 처럼 같은 경로에 대해 GET과 POST 방식을 각각 다른 메서드가 처리하도록 설정할 수 있다.

```
@Controller
public class LoginController {
        @GetMapping("/member/login")
        public String form() {
                ...
        }

        @PostMapping("/member/login")
        public String login() {
                ...
        }
}
```

- @GetMapping 애노테이션과 @PostMapping 애노테이션은 스프링 4.3 버전에 추가된 것으로 이전 버전까지는 다음 코드 처럼 @RequestMapping 애노테이션의 method 속성을 사용해서 HTTP 방식을 제한했다.

```
@Controller
public class LoginController {
        @RequestMapping(value = "/member/login", method = RequestMethod.GET
        public String form() {
                ...
        }

        @RequestMapping(value = "/member/login", method = RequestMethod.POS
        public String login() {
                ...
        }
}
```

> @GetMapping 애노테이션, @PostMapping 애노테이션뿐만 아니라 @PutMapping 애노테이션, @DeleteMapping 애노테이션, @PatchMapping 애노테이션을 제공하므로 HTTP의 GET, POST, PUT, DELETE, PATCH에 대한 매핑을 제한할 수 있다.

요청 파라미터 접근

- 약관 동의 화면을 생성하는 step1.jsp 코드를 보면 다음처럼 약관에 동의할 경우 값이 true인 'agree' 요청 파라미터의 값을 POST 방식으로 전송한다.

- 따라서 폼에서 지정한 agree 요청 파라미터의 값을 이용해서 약관 동의 여부를 확인할 수 있다.

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

- 컨트롤러 메서드에서 요청 파라미터를 사용하는 첫 번째 방법은 HttpServletRequest를 직접 이용하는 것이다.
- 예를 들면 다음과 같이 컨트롤러 처리 메서드의 파라미터로 HttpServletRequest 타입을 사용하고 HttpServletRequest와 getParameter() 메서드를 이용해서 파라미터의 값을 구하면 된다.

```
import javax.servlet.http.HttpServletRequest;

@Controller
public class RegisterController {

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

        @PostMapping("/register/step2")
        public String handleStep2(HttpServletRequest request) {
                String agreeParam = request.getParameter("agree");
                if (agreeParam == null || !agreeParam.equals("true")) {
                        return "register/step1";
                }
                return "register/step2";
        }
}
```

- 요청 파라미터에 접근하는 또 다른 방법은 @RequestParam 애노테이션을 사용하는 것이다.
- 요청 파라미터 개수가 몇 개 안되면 이 애노테이션을 사용해서 간단하게 요청 파라미터의 값을 구할 수 있다.

```
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class RegisterController {
```

```
        @PostMapping("/register/step2")
        public String handleStep2(@RequestParam(value = "agree", defaultVal
                if (!agree) {
                        return "register/step1";
                }
                return "register/step2";
        }
}
```

- @RequestParam 애노테이션은 다음의 속성을 제공한다.
- 다음 표에 따르면 위 코드는 agree 요청 파라미터의 값을 읽어와 agreeVal 파라미터에 할당한다. 요청 파라미터의 값이 없으면 "false" 문자열을 값으로 사용한다.

@RequestParam 애노테이션의 속성

속성	타입	설명
value	String	HTTP 요청 파라미터의 이름을 지정한다.
required	boolean	필수 여부를 지정한다. 이 값이 true이면서 해당 요청 파라미터에 값이 없으면 익셉션이 발생한다. 기본값은 true이다.
defaultValue	String	요청 파라미터가 값이 없을 때 사용할 문자열 값을 지정한다. 기본값은 없다.

- @RequestParam 애노테이션을 사용한 코드를 보면 다음과 같이 agreeVal 파라미터의 타입이 Boolean이다.

```
@ReqeustParam(value="agree", defaultValue="false") Boolean agreeVal
```

- 스프링 MVC는 파라미터 타입에 맞게 String 값을 변환해 준다. 위 코드는 agree 요청 파라미터의 값을 읽어와 Boolean 타입으로 변환해서 agreeVal 파라미터에 전달한다.
- Boolean 타입 외에 int, long, Integer, Long 등 기본 데이터 타입과 래퍼 타입에 대한 변환을 지원한다.

-/register/step2 경로를 처리하기 위한 코드 추가

src/main/java/controller/RegisterController.java

```
package controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
```

```
@Controller
public class RegisterController {

        ... 생략

        @PostMapping("/register/step2")
        public String handleStep2(@RequestParam(value = "agree", defaultVal
                        if (!agree) {
                                return "register/step1";
                        }

                        return "register/step2";
        }
}
```

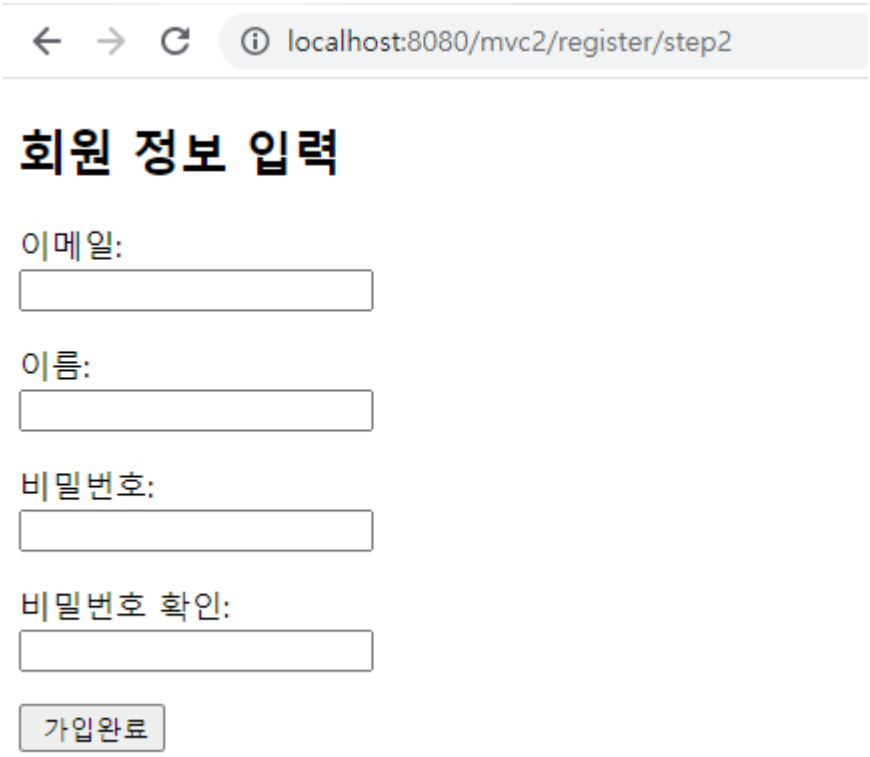
- handleStep2() 메서드는 agree 요청 파라미터의 값이 true가 아니면 다시 약관 동의 폼을 보여주기 위해 "register/step1" 뷰 이름을 리턴한다.
- 약관에 동의했다면 입력 폼을 보여주기 위해 "register/step2"를 뷰 이름으로 리턴한다.

src/main/webapp/WEB-INF/view/register/step2.jsp

```
<%@ page contentType="text/html; charset=utf-8" %>
<!DOCTYPE html>
<html>
    <head>
        <title>회원가입</title>
    </head>
    <body>
        <h2>회원 정보 입력</h2>
        <form action="step3" method="post">
            <p>
                <label>
                    이메일:<br>
                <input type="text" name="email" id="email">
                </label>
            </p>
            <p>
                <label>
                    이름:<br>
                    <input type="text" name="name" id="name">
                </label>
            </p>
            <p>
                <label>
                    비밀번호:<br>
                    <input type="password" name="password" id="password">
                </label>
```

```
            </p>
            <p>
                <label>
                    비밀번호 확인:<br>
                    <input type="password" name="confirmPassword" id="confi
                </label>
            </p>
            <input type="submit" value=" 가입완료">
        </form>
    </body>
</html>
```

- 실행 화면



리다이렉트 처리

- 컨트롤러에서 특정 페이지로 리다이렉트 시키는 방법은 "redirect:경로"를 뷰 이름으로 리턴하면 된다.
- /register/step2 경로를 GET 방식으로 접근할 때 약관 동의 화면인 /register/step1 경로를 리다이렉트 시키고 싶다면 다음과 같이 handleStep2Get() 메서드를 추가하면 된다.

src/main/java/controller/RegisterController.java

```
package controller;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
.. 생략


@Controller
public class RegisterController {

        ... 생략

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

- @RequestMapping, @GetMapping 등 요청 매핑 관련 애노테이션을 적용할 메서드가 "redirect:"로 시작하는 경로를 리턴하면 나머지 경로를 이용해서 리다이렉트할 경로를 구한다.
- "redirect:" 뒤의 문자열이 "/"로 시작하면 웹 어플리케이션을 기준으로 이동 경로를 생성한다. 예를 들어 뷰 값으로 "redirect:/register/step1"을 사용했는데 이동 경로가 "/"로 시작하므로 실제 리다이렉트할 경로는 웹 어플리케이션 경로인(현재 컨텍스트 경로가 /mvc2로 간주하면) "/mvc2"와 "/register/step1"을 연결한 "/mvc2/register/step1"이 된다.
- "/"로 시작하지 않으면 현재 경로를 기준으로 상대 경로를 사용한다. 예를 들어 "redirect:step1"을 리턴했으면 현재 요청 경로인 "http://localhost:8080/mvc2/register/step2"를 기준으로 상대 경로인 "http://localhost:8080/mvc2/register/step1"을 리다이렉트 경로로 사용한다.
- "redirect:http://localhost:8080/mvc2/register/step1"과 같이 완전한 URL을 사용하면 해당 경로로 리다이렉션한다.

커맨드 객체를 이용해서 요청 파라미터 사용하기

- step2.jsp가 생성하는 폼은 다음 파라미터를 이용해서 정보를 서버에 전송한다.
    - email
    - name
    - password
    - confirmPassword
- 폼 전송 요청을 처리하는 컨트롤러 코드는 각 파라미터의 값을 구하기 위해 다음과 같은 코드를 사용할 수 있다.

```
@PostMapping("/register/step3")
public String handleStep3(HttpServletRequest request) {
        String email = request.getParameter("email");
        String name = request.getParameter("name");
        String password = request.getParameter("password");
        String confirmPassword = request.getParameter("confirmPassword");

        RegisterRequest regReq = new RegisterRequest();
        regReq.setEmail(email);
        regReq.setName(name);
        ...
}
```

- 위 코드가 올바르게 동작하지만, 요청 파라미터 개수가 증가할 때마다 handleStep3() 메서드의 코드 길이도 함께 길어지는 단점이 있다.
- 파라미터가 20개가 넘는 복잡한 폼은 파라미터의 값을 읽어와 설정하는 코드만 40줄 이상 작성해야 한다.
- 스프링은 이런 불편함을 줄이기 위해 요청 파라미터의 값을 커맨드(command)객체에 담아주는 기능을 제공한다.
- 예를 들어 이름이 name인 요청 파라미터의 값을 커맨드 객체의 setName() 메서드를 사용해서 커맨드 객체에 전달하는 기능을 제공한다.
- 커맨드 객체라고 해서 특별한 코드를 작성해야 하는 것은 아니다. 요청 파라미터의 값을 전달받을 수 있는 세터 메서드를 포함하는 객체를 커맨드 객체로 사용하면 된다.
- 커맨드 객체는 다음과 같이 요청 매핑 애노테이션이 적용된 메서드의 파라미터에 위치한다.

```
@PostMapping("/register/step3")
public String handleStep3(RegisterRequest regReq) {
        ...
}
```

- RegisterRequest 클래스에는 setEmail(), setName(), setPassword(), setConfirmPassword() 메서드가 있다.
- 스프링은 이들 메서드를 사용해서 email, name, password, confirmPassword 요청 파라미터의 값을 커맨드 객체에 복사한 뒤 regReq 파라미터로 전달한다.
- 즉, 스프링 MVC가 handleStep3() 메서드에 전달할 RegisterRequest 객체를 생성하고 그 객체의 세터 메서드를 이용해서 일치하는 요청 파라미터의 값을 잔달한다.

- 폼에 입력한 값을 커맨드 객체로 전달받아 회원 가입을 처리하는 코드

src/main/java/controller/RegisterController.java

```
... 생략
import spring.DuplicateMemberException;
import spring.MemberRegisterService;
import spring.RegisterRequest;

@Controller
public class RegisterController {

	private MemberRegisterService memberRegisterService;

	public void setMemberRegisterService(MemberRegisterService memberRe
		this.memberRegisterService = memberRegisterService;
	}

	... 생략
	@PostMapping("/register/step3")
	public String handleStep3(RegisterRequest regReq) {

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

- handleStep3() 메서드는 MemberRegisterService를 이용해서 회원 가입을 처리한다.
- 회원 가입에 성공하면 뷰 이름으로 "register/step3"을 리턴하고, 이미 동일한 이메일 주소를 가진 회원 데이터가 존재하면 뷰 이름으로 "register/step2"를 리턴해서 다시 폼을 보여준다.
- RegisterController 클래스는 MemberRegisterService타입의 빈을 의존하므로 ControllerConfig.java 파일에 다음과 같이 의존 주입을 설정한다.
- memberRegSvc 필드에 주입받는 MemberRegisterService 타입 빈은 MemberConfig 설정 클래스에 정의되어 있다.

src/main/java/config/ControllerConfig.java

```
package config;

import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import controller.RegisterController;
import spring.MemberRegisterService;

@Configuration
public class ControllerConfig {

        @Autowired
        private MemberRegisterService memberRegSvc;

        @Bean
        public RegisterController registerController() {
                RegisterController controller =  new RegisterController();
                controller.setMemberRegisterService(memberRegSvc);
                return controller;
        }
}
```

- 회원 가입에 성공 했을 때 결과를 보여줄 step3.jsp는 다음과 같이 작성한다.

src/main/webapp/WEB-INF/view/register/step3.jsp

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
    <title>회원가입</title>
</head>
<body>
    <p>회원가입을 완료 했습니다.</p>
    <p><a href="<c:url value='/main' />">[첫 화면 이동]</a></p>
</body>
</html>
```

뷰 JSP 코드에서 커맨드 객체 사용하기

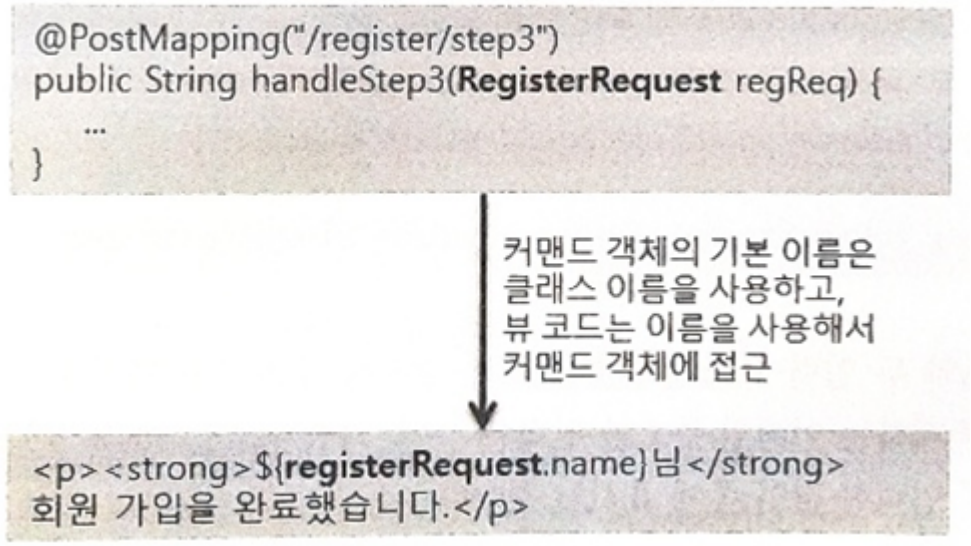
- 가입할 때 사용한 이메일 주소와 이름을 회원 가입 완료 화면에서 보여주면 사용자에게 조금 더 친절하게 보일 것이다.
- HTTP 요청 파라미터를 이용해서 회원 정보를 전달했으므로 JSP의 표현식 등을 이용해서 정보를 표시해도 되지만, **커맨드 객체를 사용해서 정보를 표시할 수도 있다**.

src/main/webapp/WEB-INF/view/register/step3.jsp

```jsp
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
    <title>회원가입</title>
</head>
<body>
    <p>
        <strong>${registerRequest.name}님</strong>
        회원가입을 완료 했습니다.
    </p>
    <p><a href="<c:url value='/main' />">[첫 화면 이동]</a></p>
</body>
</html>
```

- **${registerRequest.name}** : registerRequest가 커맨드 객체에 접근할 때 사용한 속성 이름이다.
- 스프링 MVC는 커맨드 객체의 (첫 글자를 소문자로 바꾼) 클래스의 이름과 동일한 속성 이름을 사용해서 커맨드 객체를 뷰에 전달한다.
- 커맨드 객체의 클래스 이름이 RegisterRequest인 경우 JSP 코드는 다음처럼 registerRequest라는 이름을 사용해서 커맨드 객체에 접근할 수 있다.

커맨드 객체와 뷰 모델 속성의 관계



@ModelAttribute 애노테이션으로 커맨드 객체 속성 이름 변경

- 커맨드 객체에 접근할 때 사용할 속성 이름을 변경하고 싶다면 커맨드 객체로 사용할 파라미터에 @ModelAttribute 애노테이션을 적용하면 된다.

```
import org.springframework.web.bind.annotation.ModelAttribute;

@PostMaping("/register/step3")
public String handleStep3(@ModelAttribute("formData") RegisterRequest regRe
        ...
}
```

- @ModelAttribute 애노테이션은 모델에서 사용할 속성 이름을 값으로 설정한다. 위 설정을 사용하면 뷰 코드에서 "formData"라는 이름으로 커맨드 객체에 접근할 수 있다.

커맨드 객체와 스프링 폼 연동

- 회원 정보 입력 폼에서 중복된 이메일 주소를 입력하면 텅 빈 폼을 보여준다. 폼이 비어 있으므로 입력한 값을 다시 입력해야 하는 불편함이 따른다.
- 다시 폼을 보여줄 때 커맨드 객체의 값을 폼에 채워주면 이런 불편함을 해소할 수 있다.

```
<input type="text" name="email" id="email" value="${registerRequest.email}"
<input type="text" name="name" id="name" value="${registerRequest.name}">
```

- 실제로 step2.jsp의 두 입력 요소에 굵게 표시한 코드를 추가하고 회원 가입 과정을 진행해보자. 이미 존재하는 이메일 주소를 입력한 뒤에 [가입완료] 버튼을 클릭하면 기존에 입력한 값이 폼에 표시될 것이다.
- 스프링 MVC가 제공하는 커스텀 태그를 사용하면 좀 더 간단하게 커맨드 객체의 값을 출력할 수 있다.
- 스프링은 <form:form> 태그와 <form:input>태그를 제공하고 있다. 이 두 태그를 사용하면 다음과 같이 커맨드 객체의 값을 폼으로 출력할 수 있다.

src/main/webapp/WEB-INF/view/register/step3.jsp

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<!DOCTYPE html>
<html>
    <head>
        <title>회원가입</title>
    </head>
    <body>
        <h2>회원 정보 입력</h2>
        <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>
        </body>
</html>
```

> modelAttribute 속성을 사용했다. 스프링 4.3 버전까지는 commandName 속성을 사용했는데, 스프링 5 버전부터 속성 이름이 modelAttribute로 바뀌었다. 스프링 4버전을 사용하는 환경이라면 modelAttribute 속성 대신 commandName 속성을 사용하면 된다.

- 스프링이 제공하는 폼 태그를 사용하기 위해 taglib 디렉티브를 설정했다.
- <form:form>태그는 HTML의 <form> 태그를 생성한다. <form:form>태그의 속성은 다음과 같다.
    - action: <form> 태그의 action 속성과 동일한 값을 사용한다.
    - modelAttribute : 커맨드 객체의 속성 이름을 지정한다. 설정하지 않는 경우 "command"를 기본값으로 사용한다. 예제에서 커맨드 객체의 속성 이름은 "registerRequest" 이므로 이 이름을 modelAttribute 속성값으로 설정했다.
- <form:input> 태그는 <input> 태그를 생성한다. path로 지정한 커맨드 객체의 프로퍼티를 <input> 태그의 value 속성값으로 사용한다.

- 예를 들어 <form:input path="name" />는 커맨드 객체의 name 프로퍼티 값을 value 속성으로 사용한다.
- 만약 커맨드 객체의 name 프로퍼티 값이 "스프링"이었다면 다음과 같은 input 태그를 생성한다.

```
<input id="name" name="name" type="text" value="스프링" />
```

- <form:password> 태그도 <form:input> 태그와 유사하다. password 타입의 <input>태그를 생성하므로 value 속성의 값을 빈 문자열로 설정한다.
- <form:form> 태그를 사용하려면 커맨드 객체가 존재해야 한다. step2.jsp에서 <form:form> 태그를 사용하기 때문에 step1에서 step2로 넘어오는 단계에서 이름이 "**registerRequest**"의 객체를 모델에 넣어야 <form:form> 태그가 정상 동작한다. 이를 위해 RegisterController 클래스의 handleStep2() 메서드에 다음과 같이 코드를 추가했다.

src/main/java/controller/RegisterController.java

```
package controller;

import org.springframework.ui.Model;
... 생략

@Controller
public class RegisterController {
        ... 생략

        @PostMapping("/register/step2")
        public String handleStep2(@RequestParam(value = "agree", defaultVal
                        if (!agree) {
                                return "register/step1";
                        }

                        model.addAttribute("registerRequest", new RegisterR
                        return "register/step2";
        }

        ... 생략
}
```

컨트롤러 구현 없는 경로 매핑

- step3.jsp 코드를 보면 다음 코드를 볼 수 있다.

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

- step3.jsp는 회원 가입 완료 후 첫 화면으로 이동할 수 있는 링크를 보여준다. 이 첫 화면은 단ㅎ순히 환영 문구와 회원 가입으로 이동할 후 있는 링크만 제공한다고 하자 이를 위한 컨트롤러 클래스는 틀별히 처리할 것이 없기 때문에 다음처럼 단순히 뷰 이름만 리턴하도록 구현한 것이다.

```
@Override
public void MainController {
        @RequestMapping("/main")
        public String main() {
                return "main";
        }
}
```

- 이 컨트롤러 코드는 요청 경로와 뷰 이름을 연결해주는 것에 불과하다. 단순 연결을 위해 특별한 로직이 없는 컨트롤러 클래스를 만드는 것은 성가신 일이다.
- WebMvcConfigurer 인터페이스의 addViewControllers() 메서드를 사용하면 이런 성가심을 없앨 수 있다. 이 메서드를 재정의하면 컨트롤러 구현없이 다음의 간단한 코드로 요청 경로와 뷰 이름을 연결할 수 있다.

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

src/main/java/config/MvcConfig.java

```
package config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.DefaultServletHand
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ViewControllerRegi
... 생략
@Configuration
@EnableWebMvc
public class MvcConfig implements WebMvcConfigurer {

... 생략

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

- 뷰 이름으로 main 을 사용하므로 이에 해당하는 main.jsp 파일을 다음과같이 작성해 보자.

src/main/webapp/WEB-INF/view/main.jsp

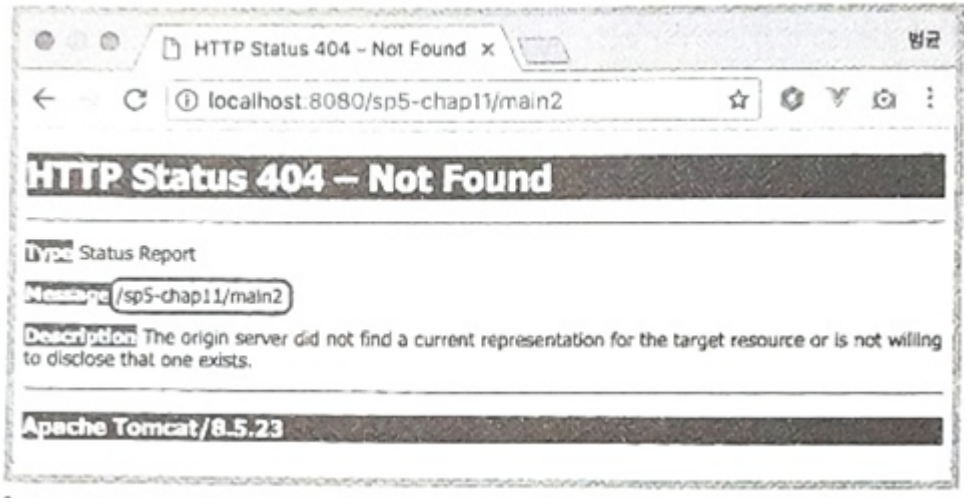
```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
    <head>
        <title>메인</title>
    </head>
    <body>
        <p>환영합니다.</p>
        <p><a href="<c:url value="/register/step1" />">[회원 가입하기]</a></
    </body>
</html>
```

주요 에러 발생 상황

- 처음 스프링 MVC를 이용해서 웹 개발을 하다보면 사소한 설정 오류나 오타로 고생한다.이 절에서는 입문 과정에서 겪게 되는 에러 사례를 정리해 보았다.

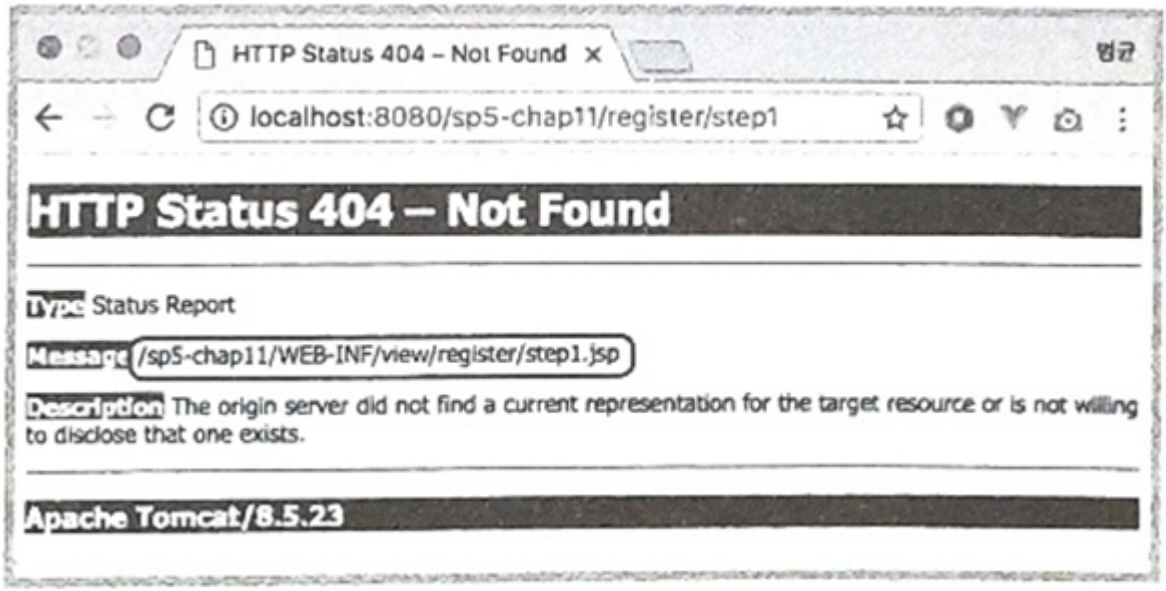
요청 매핑 애노테이션과 관련된 주요 익셉션

- 흔한 에러는 404 에러이다.
- 요청 경로를 처리할 컨트롤러가 존재하지 않거나 WebMvcConfigurer를 이용한 설정이 없다면 다음과 같이 404 에러가 발생한다.

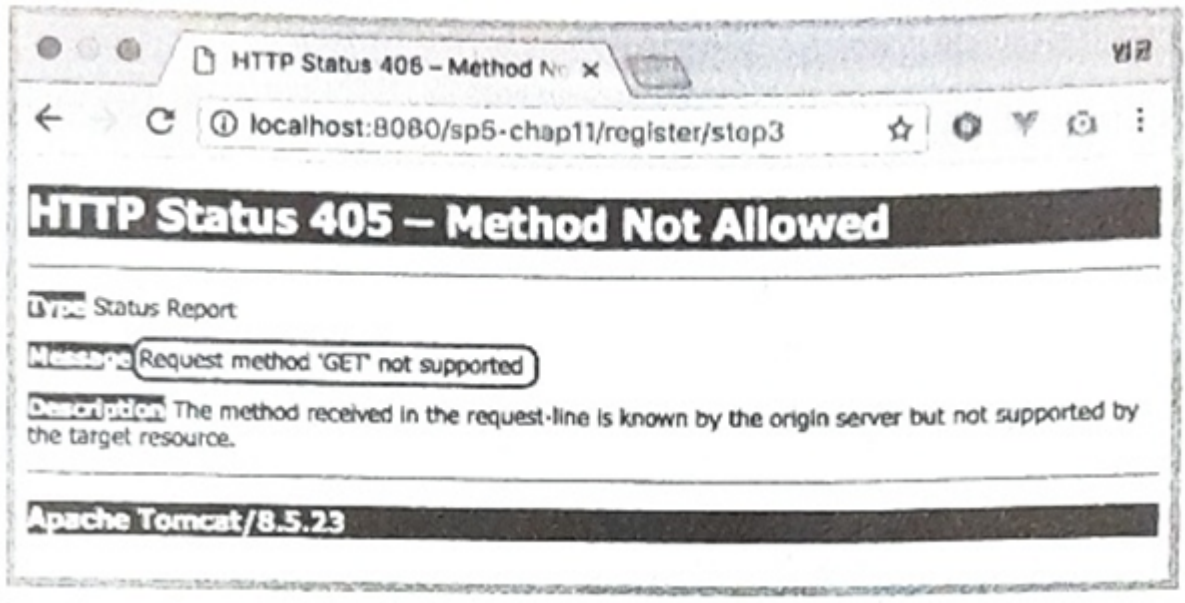


- 404 에러가 발생하면 다음 사항을 확인해야 한다.
    - 요청 경로가 올바른지
    - 컨트롤러에 설정한 경로가 올바른지

    - 컨트롤러 클래스를 빈으로 등록했는지
    - 컨트롤러 클래스에 @Controller 애노테이션을 적용했는지
- 뷰 이름에 해당하는 JSP 파일이 존재하지 않아도 404 에러가 발생한다.
- 차이점이 있다면 다음처럼 존재하지 않는 JSP 파일의 경로가 출력된다는 점이다.



- 다음과 같은 에러가 발생한다면 컨트롤러에서 리턴하는 뷰 이름에 해당하는JSP 파일이 존재하는지 확인해야 한다.
- 지원하지 않는 전송 방식(method)을 사용한 경우 405 에러가 발생한다. 예를 들어 POST 방식만 처리하는 요청 경로를 GET 방식으로 연결하면 다음과와 같이 405에러가 발생한다.

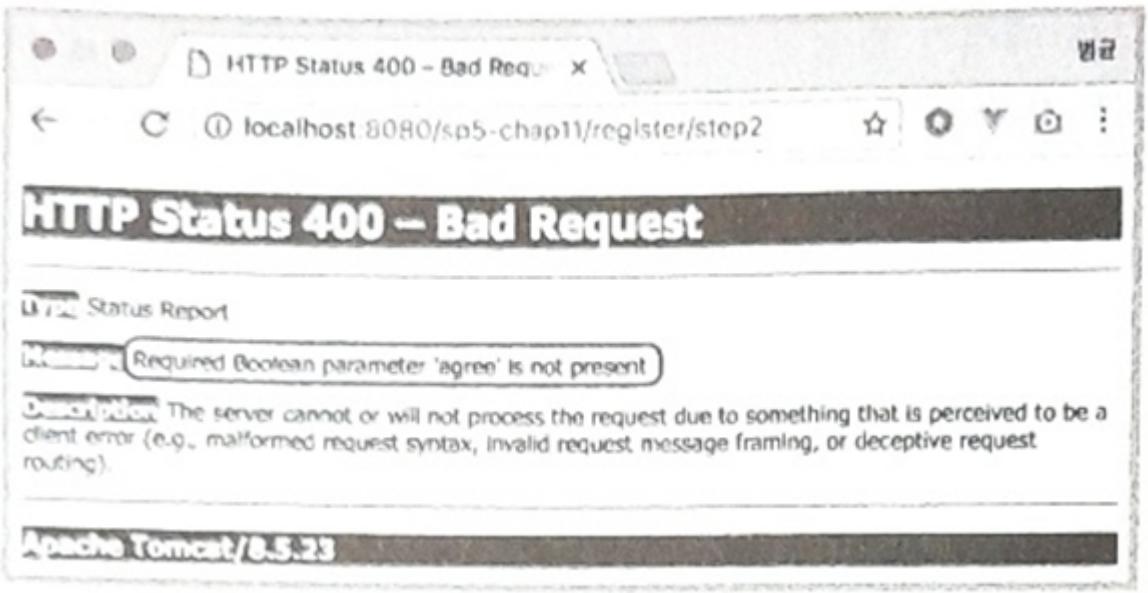


@RequestParam이나 커맨드 객체와 관련된 주요 익셉션

- RegisterController 클래스의 handleStep2() 메서드에서 다음과 같이 @RequestParam애노테이션을 필수로 설정하고 기본값을 지정하지 않았다고 하자.

```
@PostMapping("/register/step2")
public String handleStep2(
        // 필수로 존재해야 하고 기본값이 없음
        @RequestParam("agree") Boolean agree, Model model) {
        ...
}
```

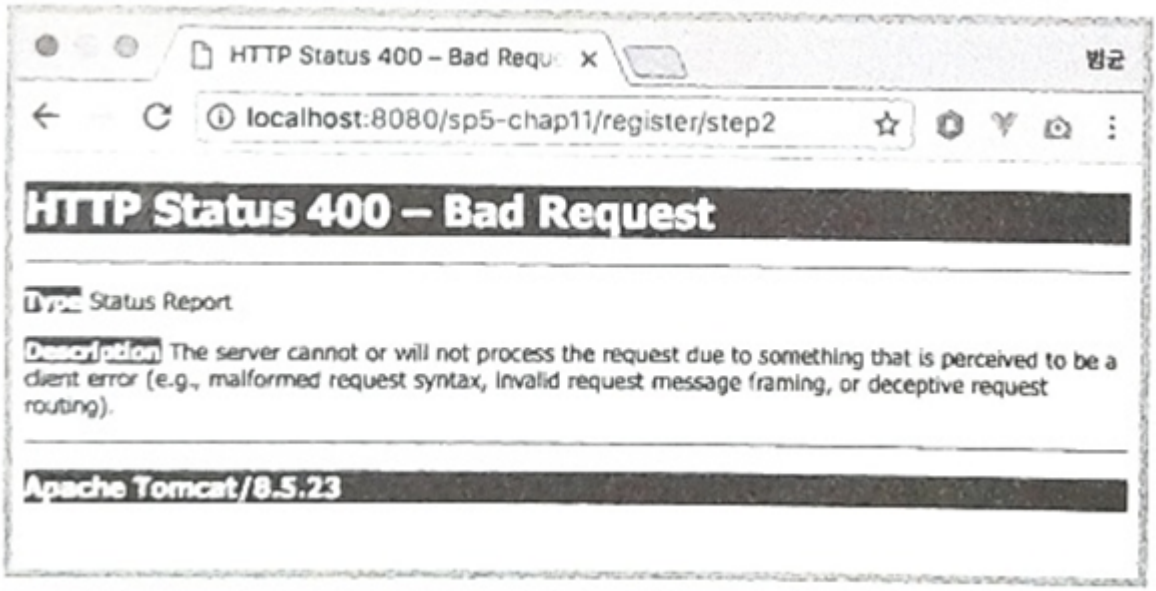
- 이렇게 수정한 뒤 약관 동의 화면에서 '약관 동의'를 선택하지 않고 [다음 단계] 버튼을클릭해보자.
- checkbox 타입의 <input> 요소는 선택되지 않으면 파라미터로 아무 값도전송하지 않는다.
- 즉 agree 파라미터를 전송하지 않기 때문에 @RequestParam 애노테이션을 처리하는 과정에서 필수인 "agree" 파라미터가 존재하지 않는다는 익셉션이 발생하게 된다.
- 스프링 MVC는 이 익셉션이 발생하면 다음과 같이 400 에러를 응답으로 전송한다.
- 에러 메시지는 필수인 'agree' 파라미터가 없다는 내용이다.



- 요청 파라미터의 값을 @RequestParam이 적용된 파라미터의 타입으로 변환할 수 없는 경우에도 에러가 발생한다.
- 예를 들어 step1.jsp에서 <input> 태그의 value 속성을 다음과 같이 "true"에서 "true1"로 변경해보자

```
<input type="checkbox" name="agree" value="true1"> 약관 동의
```

- 이렇게 변경한 약관 동의 과정을 진행하면 다음과 같이 400 에러가 발생한다. 400 에러가 발생하는 이유는 "truel" 값을 Boolean 타입으로 변환할 수 없기 때문이다.



- 400 에러는 요청 파라미터의 값을 커맨드 객체에 복사하는 과정에서도 동일하게 발생한다.
- 커맨드 객체의 프로퍼티가 int 타입인데 요청 파라미터의 값이 "abc"라면, "abc"를 int 타입으로 변환할 수 없기 때문에 400 에러가 발생한다.
- 브라우저에 표시된 400 에러만 보면 어떤 문제로 이 에러가 발생했는지 찾기가 쉽지 않다.
- 이때는 콘솔에 출력된 로그 메시지를 참고하면 도움이 된다. 400 에러가 발생할 때 콘솔에 출력되는 로그를 보면 다음 메시지를 확인할 수 있다(Logback 등의 로깅 프레임워크를 설정했다면 다른 형식으로 메시지가 나올 수 있지만 기본 내용은 동일하다).

Logback으로 자세한 에러 로그 출력하기

- 로그 레벨을 낮추면 더 자세한 로그를 얻을 수 있다. Logback을 예로 들어보자. 먼저 pom.xml에 다음의 Logback 관련 의존을 추가한다.

```
<dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.36</version>
</dependency>
<dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.11</version>
</dependency>
```

- src/main/resources 폴더를 생성하고 그 폴더에 다음 logback.xml 파일을 생성한다.

- src/main/resources 폴더를 새로 생성했다면 메이븐 프로젝트를 업데이트해야(프로젝트에서 우클릭 - Maven -> Update Project 메뉴) src/main/resources 폴더가 소스 폴더로 잡힌다.

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

<configuration>
        <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender"
        <encoder>
            <pattern>%d %5p %c{2} - %m%n</pattern>
        </encoder>
    </appender>
    <root level="INFO">
        <appender-ref ref="stdout" />
    </root>

    <logger name="org.springframework.jdbc" level="DEBUG" />
</configuration>
```

- 위 설정은 org.springframework.web.servlet과 그 하위 패키지의 클래스에서 출력한 로그를 상세한 수준(DEBUG' 레벨)으로 남긴다. 0 이렇게 로그 설정을 한 뒤에 서버를 재구동하고 400 에러가 발생하는 상황이 되면 콘솔에서 보다 상세한 로그를 볼 수 있다.
- 상세한 로그를 보면 문제 원인을 찾는데 도움이 된다.

커맨드 객체 : 중첩 · 콜렉션 프로퍼티

- 세 개의 설문 항복과 응답자의 지역과 나이를 입력받는 설문 조사 정보를 담기 위해 다음과 같이 클래스를 작성해보자.

src/main/survey/Respondent.java

```
package survey;

public class Respondent {

        private int age;
        private String location;

        public int getAge() {
                return age;
        }

        public void setAge(int age) {
                this.age = age;
        }
```

```
        public String getLocation() {
                return location;
        }

        public void setLocation(String location) {
                this.location = location;
        }
}
```

src/main/java/survey/AnsweredData.java

```
package survey;

import java.util.List;

public class AnsweredData {

        private List<String> responses;
        private Respondent res;

        public List<String> getResponses() {
                return responses;
        }

        public void setResponses(List<String> responses) {
                this.responses = responses;
        }

        public Respondent getRes() {
                return res;
        }

        public void setRes(Respondent res) {
                this.res = res;
        }
}
```

- Respondent 클래스는 응답자 정보를 담는다.
- AnsweredData 클래스는 설문 항목에 대한 답변과 응답자 정보를 함께 담는다.
- AnsweredData 클래스는 답변 목록을 제장하기 위해 List 타입의 responses 프로퍼티를 사용했고, 응답자 정보를 담기 위해 Respondent 타입의 res 프로퍼티를 사용했다
- AnsweredData 클래스는 앞서 커맨드 객체로 사용한 클래스와 비교하면 다음 차이가 있다.

    - 리스트 타입의 프로퍼티가 존재한다. responses 프로퍼티는 String 타입의 값을 갖는 List 콜렉션이다.
    - 중첩 프로퍼티를 갖는다. res 프로퍼티는 Respondent 타입이며 res 프로퍼티는 다시 age와 location 프로퍼티를 갖는다. 이를 중첩된 형식으로 표시하면 res, age 프로퍼티나 res Jocation 프로퍼티로 표현할 수 있다.

- 스프링 MVC는 커맨드 객체가 리스트 타입의 프로퍼티를 가졌거나 중첩 프로퍼티를 가진 경우에도 요청 파라미터의 값을 알맞게 커맨드 객체에 설정해주는 기능을 제공하고 있다. 규칙은 다음과 같다.

    - HTTP 요청 파라미터 이름이 "프로퍼티이름[인덱스]' 형식이면 List 타입 프로퍼티의 값 목록으로 처리한다.
    - HTTP 요청 파라미터 이름이 "프로퍼티이름, 프로퍼티이름'과 같은 형식이면 중첩 프로퍼티 값을 처리한다.

- 예를 들어 이름이 responses이고 List 타입인 프로퍼티를 위한 요청 파라미터의 이름으로 “responses[0]", "responses[1]"을 사용하면 각각 0번 인덱스와 1번 인덱스의 값으로 사용된다.

- 중첩 프로퍼티의 경우 파라미터 이름을 "res.name"로 지정하면 다음과 유사한 방식으로 커맨드 객체에 파라미터의 값을 설정한다.

```
commandObj.getRes().setName(request.getParameter("res.name"));
```

- AnsweredData 클래스를 커맨드 객체로 사용하는 예제를 작성해보자. 먼저 간단한 컨트롤러 클래스를 다음과 같이 작성한다.

src/main/java/survey/SurveyController.java

```
package survey;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/survey")
public class SurveyController {

	@GetMapping
	public String form() {
		return "survey/surveyForm";
	}

	@PostMapping
```

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

- form() 메서드와 submit() 메서드의 요청 매핑 애노테이션은 전송 방식만을 설정하고 클래스의 @RequestMapping에만 경로를 지정했다.
- 이 경우 form() 메서드와 submit()메서드가 처리하는 경로는 "/survey"가 된다.
- 즉 form() 메서드는 GET 방식의"/survey" 요청을 처리하고 submit() 메서드는 POST 방식의 "/survey" 요청을 처리한다.
- submit() 메서드는 커맨드 객체로 AnsweredData 객체를 사용한다.
- 컨트롤러 클래스를 새로 작성했으니 다음과 같이 ControllerConfig 파일에 컨트롤러 클래스를 빈으로 추가하자.

src/main/java/config/ControllerConfig.java

```
package config;
... 생략

@Configuration
public class ControllerConfig {

        @Autowired
        private MemberRegisterService memberRegSvc;

        ... 생략

        @Bean
        public SurveyController surveyController() {
                return new SurveyController();
        }
}
```

- SurveyController 클래스의 form() 메서드와 submit() 메서드는 각각 뷰 이름으로"survey/surveyForm" "survey/submitted". 사용한다.
- 두 뷰를 위한 JSP 파일을 만들자 먼저 surveyForm.jsp 파일을 다음과 같이 작성한다.

src/main/webapp/WEB-INF/view/survey/surveyForm.jsp

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

```
    <title>설문조사</title>
</head>
<body>
    <h2>설문조사</h2>
    <form method="post">
        <p>
            1. 당신의 역할은?<br>
            <label>
                <input type="radio" name="responses[0]" value="서버">
                서버개발자
            </label>
            <label>
                <input type="radio" name="responses[0]" value="프론트">
                프론트개발자
            </label>
            <label>
                <input type="radio" name="responses[0]" value="풀스택">
                풀스택가발자
            </label>
        </p>
        <p>
            2. 가장 많이 사용하는 개발도구는?<br>
            <label>
                <input type="radio" name="responses[1]" value="Eclipse"
                Eclipse
            </label>
            <label>
                <input type="radio" name="responses[1]" value="Intellij
                Intellij
            </label>
            <label>
                <input type="radio" name="responses[1]" value="Sublime"
                Sublime
            </label>
        </p>
        <p>
            3. 하고싶은 말<br />
            <input type="text" name="responses[2]">
        </p>
        <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>
    </form>
```

```
    </body>
</html>
```

- 각 <input> 태그의 name 속성은 다음과 같이 커맨드 객체의 프로퍼티에 매핑된다.
  - responses[0] → responses 프로퍼티(List 타입)의 첫 번째 값
  - responses[1] → responses 프로퍼티(List 타입)의 두 번째 값
  - responses[2] → responses 프로퍼티(List 타입)의 세 번째 값
  - res.location → res 프로퍼티(Respondent 타입)의 프로퍼티
- 폼 전송 후 결과를 보여주기 위한 submitted.jsp는 다음과 같이 작성한다.
- 커맨드 객체(ansData)의 List 타입 프로퍼티인 responses에 담겨 있는 각 값을 출력한다.
- 중첩 프로퍼티인 res. location과 res.age의 값을 출력한다.

src/main/webapp/WEB-INF/view/survey/submitted.jsp

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
    <head>
        <title>응답 내용</title>
    </head>
    <body>
        <p>응답 내용:</p>
        <ul>
            <c:forEach var="response" items="${ansData.responses}" varStatu
                <li>${status.index + 1}번 문항 : ${response}</li>
            </c:forEach>
        </ul>
        <p>응답자 위치: ${ansData.res.location}</p>
        <p>응답자 나이: ${ansData.res.age}</p>
    </body>
</html>
```

- 필요한 코드를 모두 작성했으니 서버를 재시작하고 웹 브라우저에 http://localhost:8080/... /survey 주소를 입력해보자.
- 다음과 같이 설문 조사 폼이 출력된다.

설문조사

1. 당신의 역할은?
○ 서버개발자 ○ 프론트개발자 ○ 풀스택가발자

2. 가장 많이 사용하는 개발도구는?
○ Eclipse ○ Intellij ○ Sublime

3. 하고싶은 말

응답자 위치:

응답자 나이:

전송

- 폼에 알맞게 값을 입력한 다음 [전송] 버튼을 누르자.
- 응답자 나이에 해당하는 "res.age"프로퍼티의 타입은 int 타입이기 때문에 나이에는 정수를 입력해야 한다는 점에 주의하자
- [전송] 버튼을 누르면 다음과 같이 커맨드 객체의 값이 출력된다. 결과를 보면 폼에서 전송한 데이터가 커맨드 객체에 알맞게 저장된 것을 확인할 수 있다.

응답 내용:

- 1번 문항 : 서버
- 2번 문항 : Intellij
- 3번 문항 : 1

응답자 위치: 2

응답자 나이: 3

Model을 통해 컨트롤러에서 뷰에 데이터 전달하기

- 컨트롤러는 뷰가 응답 화면을 구성하는데 필요한 데이터를 생성해서 전달해야 한다. 이때 사용하는 것이 Model이다.
- 앞서 HelloController 클래스를 작성할 때 다음과 같이Model을 사용했다.

```
import org.springframework.ui.Model;

@Controller
public class HelloController {
        @RequestMapping("/hello")
        public String hello(Model model, @RequestParam(value = "name", requ
                model.addAttribute("greeting", "안녕하세요, " + name);
                return "hello";
        }
}
```

- 뷰에 데이터를 전달해는 컨트롤러는 hello() 메서드처럼 다음 두 가지를 하면 된다.
    - 요청 매핑 애노테이션이 적용된 메서드의 파라미터로 Model을 추가
    - Model 파라미터의 addAttribute() 메서드로 뷰에서 사용할 데이터 전달
- addAttribute() 메서드의 첫 번째 파라미터는 속성 이름이다.
- 뷰 코드는 이 이름을 사용해서 데이터에 접근한다. JSP는 다음과 같이 표현식을 사용해서 속성값에 접근한다

```
${greeting}
```

- 앞서 작성한 SurveyController 예제는 surveyForm.jsp에 설문 항목을 하드 코딩했다.
- 설문 항목을 컨트롤러에서 생성해서 뷰에 전달하는 방식으로 변경해보자. 먼저 개별 설문 항목 데이터를 담기 위한 클래스를 다음과 같이 작성한다.
- Question 클래스의 title options는 각각 질문 제목과 답변 옵션을 보관한다. 주관식이면 생성자를 사용해서 답변 옵션이 없는 Question 객체를 생성한다.

src/main/java/survey/Question.java

```
package survey;

import java.util.Collections;
import java.util.List;

public class Question {

        private String title;
        private List<String> options;

        public Question(String title, List<String> options) {
                this.title = title;
                this.options = options;
        }
```

```
        public Question(String title) {
                this(title, Collections.<String>emptyList());
        }

        public String getTitle() {
                return title;
        }

        public List<String> getOptions() {
                return options;
        }

        public boolean isChoice() {
                return options !=null && !options.isEmpty();
        }
}
```

- 다음 작업은 SurveyController가 Question 객체 목록을 생성해서 뷰에 전달하도록 구현하는 것이다.
- 실제로는 DB와 같은 곳에서 정보를 읽어와 Question 목록을 생성하겠지만 이 예제는 컨트롤러에서 직접 생성하도록 구현했다.
- 앞서 작성한 SurveyController 클래스의 코드를 다음과 같이 변경하자.

src/main/java/survey/SurveyController.java

```
package survey;

import java.util.Arrays;
import java.util.List;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
... 생략

@Controller
@RequestMapping("/survey")
public class SurveyController {

        @GetMapping
        public String form(Model model) {
                List<Question> questions = createQuestions();
                model.addAttribute("questions", questions);
                return "survey/surveyForm";
        }

        private List<Question> createQuestions() {
                Question q1 = new Question("당신의 역할은 무엇입니까?", Array
                Question q2 = new Question("많이 사용하는 개발도구는 무엇입니
                Question q3 = new Question("하고 싶은 말을 적어주세요.");
                return Arrays.asList(q1, q2, q3);
```

```
        }

        ... 생략
}
```

- form() 메서드에 Model 타입의 파라미터를 추가했고 생성한 Question 리스트를 "questions"라는 이름으로 모델에 추가했다.
- 컨트롤러에서 전달한 Question 리스트를 사용해서 폼 화면을 생성하도록 JSP 코드를 다음과 같이 수정하자.

src/main/webapp/WEB-INF/view/survey/surveyForm.jsp

```
<%@ page contentType="text/html; charset=utf-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
        <head>
        <title>설문조사</title>
    </head>
    <body>
        <h2>설문조사</h2>
        <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.in
                            ${option}
                        </label>
                    </c:forEach>
                </c:if>
                <c:if test="${! q.choice}">
                    <input type="text" name="responses[${status.index}]">
                </c:if>
            </p>
        </c:forEach>

        <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>
        </body>
</html>
```

- 코드를 수정했으니 다시 실행해보자. 다음처럼 SurveyController에서 Model을통해 전달한 Question 리스트를 이용해서 설문 폼이 생성된 것을 확인할 수 있다.

> **설문조사**
>
> 1. 당신의 역할은 무엇입니까?
> ○ 서버 ○ 프론트 ○ 풀스택
>
> 2. 많이 사용하는 개발도구는 무엇입니까?
> ○ 이클립스 ○ 인텔리J ○ 서브라임
>
> 3. 하고 싶은 말을 적어주세요.
>
> 응답자 위치:
>
> 응답자 나이:
>
> 전송

ModelAndView를 통한 뷰 선택과 모델 전달

- 지금까지 구현한 컨트롤러는 두 가지 특징이 있다.
    - Model을 이용해서 뷰에 전달할 데이터 설정
    - 결과를 보여줄 뷰 이름을 리턴
- ModelAndView를 사용하면 이 두 가지를 한 번에 처리할 수 있다.
- 요청 매핑 애노테이션을 적용한 메서드는 String 타입 대신 ModelAndView를 리턴할 수 있다.
- ModelAndView는 모델과 뷰 이름을 함께 제공한다. 다음과 같이 ModelAndView 클래스를 이용해서 SurveyController 클래스의 form() 메서드를 구현할 수 있다.

```
import org.springframework.web.servlet.ModelAndView;
```

```
@Controller
@RequestMapping("/survey")
public class SurveyController {
        @GetMapping
        public ModelAndView form() {
                List<Question> questions = createQuestions();

                ModelAndView mav = new ModelAndView();
                mav.addObject("questions", questions);
                mav.setViewName("survey/surveyForm");
                return mav;
        }
}
```

- 뷰에 전달할 모델 데이터는 addObject() 메서드로 추가한다.
- 뷰 이름은 setViewName()메서드를 이용해서 지정한다.

GET 방식과 POST 방식에 동일 이름 커맨드 객체 사용하기

- \〈form:form\〉태그를 사용하려면 커맨드 객체가 반드시 존재해야 한다.
- 최초에 폼을 보여주는 요청에 대해 <form form> 태그를 사용하려면 폼 표시 요청이 왔을 때에도 커맨드 객체를 생성해서 모델에 저장해야 한다.
- 이를 위해 RegisterController 클래스의 handleStep2() 메서드는 다음과 같이 Model에 직접 객체를 추가했다.

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

- 커맨드 객체를 파라미터로 추가하면 좀 더 간단해진다.

```
@PostMapping("/register/step2")
public String handleStep2(@RequestParam(value = "agree", defaultValue = "fa
        if (!agree) {
                return "agree/step1";
        }
        return "register/step2";
}
```

- 이름을 명시적으로 지정하려면 ModelAttribute 애노테이션을 사용한다.

- 예를 들어"/login" 요청 경로일 때 GET 방식이면 로그인 폼을 보여주고 POST 방식이면 로그인을 처리하도록 구현한 컨트롤러를 만들어야 한다고 하자.
- 입력 폼과 폼 전송 처리에서 사용할 커맨드 객체의 속성 이름이 클래스 이름과 다르다면 다음과 같이 GET 요청과 POST 요청을 처리하는 메서드에 @ModelAttribute 애노테이션을 붙인 커맨드 객체를 파라미터로 추가하면 된다

```
@Controller
@RequestMapping("/login")
public class LoginController {
        @GetMapping
        public String form(@ModelAttribute("login") LoginCommand loginComma
                return "login/loginForm";
        }

        @PostMapping
        public String form(@ModelAttribute("login") LoginCommand loginComma
                ...
        }
}
```

주요 폼 태그 설명

- 스프링 MVC는 <form:form>, <form:input> 등 HTML 폼과 커맨드 객체를 연동하기 위한 JSP 태그 라이브러리를 제공한다.
- 이 두 태그 외에도 <select>를 위한 태그와 체크박스나 라디오 버튼을 위한 커스텀 태그도 제공한다.

<form> 태그를 위한 커스텀 태그: <form:form>

- \〈form:form\〉커스텀 태그는 <form> 태그를 생성할 때 사용된다.
- <form:form> 커스텀 태그를 사용하는 가장 간단한 방법은 다음과 같다.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
...
<form:form>
...
<input type="submit" value="가입 완료">
</form:form>
```

- \〈form:form\〉태그의 method 속성과 action 속성을 지정하지 않으면 method 속성값은 "post"로 설정되고 action 속성값은 현재 요정 URL로 설정된다.
- 예를 들어 요청 URI가 "/mvc2/register/step2"라면 위 <form:form> 태그는 다음의 <form> 태그를 생성한다.

```
<form id="command" action="/mvc2/register/step2" method="post">
</form>
```

- 생성된 <form> 태그의 id 속성값으로 입력 폼의 값을 저장하는 커맨드 객체의 이름을 사용한다.
- 커맨드 객체 이름이 기본값인 "command"가 아니면 다음과 같이 modelAttribute 속성값으로 커맨드 객체의 이름을 설정해야 한다.

```
<form:form modelAttribute="loginCommand">
        ...
</form:form>
```

- <form:form> 커스텀 태그는 <form> 태그와 관련하여 다음 속성을 추가로 제공한다.
    - action : 폼 데이터를 전송할 URL을 입력 (HTML <form> 태그 속성)
    - enctype : 전송될 데이터의 인코딩 타입. HTML <form> 태그 속성과 동일
    - method : 전송 방식. HTML <form> 태그 속성과 동일
- \〈form:form\〉태그의 몸체에는 <input> 태그나 <select> 태그와 같이 입력 폼을 출력하는 데 필요한 HTML 태그를 입력할 수 있다.
- 이때 입력한 값이 잘못되어 다시 값을 입력해야 하는 경우 다음과 같이 커맨드 객체의 값을 사용해서 이전에 입력한 값을 출력할 수 있을 것이다.

```
<form:form modelAttribute="loginCommand">
        ...
        <input type="text" name="id" value="${loginCommand.id}" />
        ...
</form:form>
```

- <input> 태그를 직접 사용하기보다는 뒤에서 설명할 <form:input> 등의 태그를 사용해서 폼에 커맨드 객체의 값을 표시하면 편리하다.

<input> 관련 커스텀 태그 : <form:input>, <form:password>, <form:hidden>

- 스프링은 <input> 태그를 위해 다음과 같은 커스텀 태그를 제공한다

<input>태그와 관련된 기본 커스텀 태그

커스텀 태그	설명
<form:input>	text 타입의 <input>태그

커스텀 태그	설명
<form:password>	password 타입의 <input>태그
<form:hidden>	hidden 타입의 <input>태그

- <form:input> 커스텀 태그는 다음과 같이 path 속성을 사용해서 연결할 커맨드 객체의 프로퍼티를 지정한다.

```
<form:form modelAttribute="registerRequest" action="step3">
<p>
        <label>
                이메일:<br>
                <form:input path="email" />
        </label>
</p>
... 생략
</form:form>
```

- 코드가 생성하는 HTML<input〉 태그는 아래와 같다.

```
<form id="registerRequest" action="step3" method="post">
<p>
        <label>
                이메일:<br>
                <input id="email" name="email" type="text" value="" />
        </label>
</p>
</form>
```

- id 속성과 name 속성값은 프로퍼티의 이름으로 설정하고, value 속성에는 <form:input> 커스텀 태그의 path 속성으로 지정한 커맨드 객체의 프로퍼티 값이 출력된다.
- <form:password> 커스텀 태그는 password 타입의 <input> 태그를 생성하고, <form:hidden> 커스텀 태그는 hidden 타입의 <input> 태그를 생성한다.
- 두 태그 모두 path 속성을 사용하여 연결할 커맨드 객체의 프로퍼티를 지정한다.

```
<form:form modelAttribute="loginCommand">
        <form:hidden path="defaultSecurityLevel" />
        ...
        <form:password path="password" />
</form:form>
```

<select> 관련 커스텀 태그 : <form:select>, <form:options>, <form:option>

- <select> 태그와 관련된 커스텀 태그는 다음과 같이 세 가지가 존재한다.

커스텀 태그	설명
<form:select>	<select>태그를 생성한다. <option> 태그를 생성할 때 필요한 콜렉션을 전달받을 수도 있다.
<form:options>	지정한 콜렉션 객체를 이용하여 <option> 태그를 생성한다.
<form:option>	<option> 태그 한 개를 생성한다.

- <select> 태그는 선택 옵션을 제공할 때 주로 사용한다.
- 예를 들어 <selec> 태그를 이용해서 직업 선택을 위한 옵션을 제공한다고 하자. 이런 옵션 정보는 컨트롤러에서 생성해서 뷰에 전달하는 경우가 많다.
- <select> 태그에서 사용할 옵션 목록을 Model을 통해 전달한다.

```
@GetMapping("/login")
public String form(Model model) {
        List<String> loginTypes = new ArrayList<>();
        loginTypes.add("일반회원");
        loginTypes.add("기업회원");
        loginTypes.add("헤드헌터회원");
        model.addAttribute("loginTypes", loginTypes);
        return "login/form";
}
```

- <form:select>커스텀 태그를 사용하면 뷰에 전달한 모델 객체를 갖고 간단하게 <select>와 <option> 태그를 생성할 수 있다.
- 다음은 <form:select> 커스텀 태그를 이용해서 <select> 태그를 생성하는 코드 예이다.

```
<form:form modelAttribute="login">
<p>
        <label for="loginType">로그인 타입</label>
        <form:select path="loginType" items="${loginTypes}" />
</p>
</form:form>
```

- path 속성은 커맨드 객체의 프로퍼티 이름을 입력하며, items 속성에는 <option> 태그를 생성할 때 사용할 콜렉션 객체를 지정한다.
- 이 코드의 <form:select> 커스텀 태그는 다음 HTML 태그를 생성한다.

```
<select id="loginType" name="loginType">
        <option value="일반회원">일반회원</option>
        <option value="기업회원">기업회원</option>
```

```
        <option value="헤드헌터회원">헤드헌터회원</option>
</select>
```

- 생성한 코드를 보면 콜렉션 객체의 값을 이용해서 <option> 태그의 value 속성과 텍스트를 설정한 것을 알 수 있다.
- <form:options> 태그를 사용해도 된다. <form:select> 커스텀 태그에 <form:options>커스텀 태그를 중첩해서 사용한다. <form:options> 커스텀 태그의 items 속성에 값 목록으로 사용할 모델 이름을 설정한다.

```
<form:select path="loginType">
        <option value="">---- 선택하세요 ---</option>
        <form:options items="${loginTypes}" />
</form:select>
```

-<form:options>커스텀 태그는 주로 콜렉션에 없는 값을 <option> 태그로 추가할 때 사용한다.

- <form:option>커스텀 태그는 <option> 태그를 직접 지정할 때 사용된다. 다음 코드는 <form:option> 커스텀 태그의 사용 예이다.

```
<form:select path="loginType">
        <form:option value="일반회원" />
        <form:option value="기업회원">기업</form:option>
        <form:option value="헤드헌터회원" label="헤드헌터" />
</form>
```

- <form:option> 커스텀 태그의 value 속성은 <option> 태그의 value 속성값을 지정한다.
- <form:option> 커스텀 태그의 몸체 내용을 입력하지 않으면 value 속성에 지정한 값을 텍스트로 사용한다.
- 몸체 내용을 입력하면 몸체 내용을 텍스트로 사용한다.
- label 속성을 사용하면 그 값을 텍스트로 사용한다.
- 다음은 위 코드가 생성한 HTML 결과이다.

```
<select id="loginType" name="loginType">
        <option value="일반회원">일반회원</option>
        <option value="기업회원">기업</option>
        <option value="헤드헌터회원">헤드헌터</option>
</select>
```

- <option> 태그를 생성하는데 사용할 콜렉션 객체가 String이 아닐 수도 있다. 예를 들어 다음 Code 클래스를 보자.

```
public class Code {
        private String code;
        private String label;

        public Code(String code, String label) {
                this.code = code;
                this.label = label;
        }

        public String getCode() {
                return code;
        }

        public String getLabel() {
                return label;
        }
}
```

- 컨트롤러는 코드 목록 표시를 위해 Code 객체 목록을 생성해서 뷰에 전달할 수 있다.
- 뷰는 Code 객체의 code 프로퍼티와 label 프로퍼티를 각각 <option> 태그의 value 속성과 텍스트로 사용해야 한다.
- 이렇게 콜렉션에 저장된 객체의 특정 프로퍼티를 사용해야 하는 경우 itemValue 속성과 itemLabel 속성을 사용한다. 이 두 속성은 <option>태그를 생성하는 데 사용할 객체의 프로퍼티를 지정한다.

```
<form:select path="jobCode">
        <option value="">--- 선택하세요 ---</option>
        <form:options items="${jobCodes}" itemLabel="label" itemValue="code
</form:select>
```

- 위 코드는 jobCodes 콜렉션에 저장된 객체를 이용해서 <option> 태그를 생성한다.
- 이때 객체의 code 프로퍼티 값을 <option> 태그의 value 속성값으로 사용하고, 객체의 label 프로퍼티 값을 <option> 태그의 텍스트로 사용한다.
- <form:select> 커스텀 태그도<form:options>커스텀 태그와 마찬가지로 itemLabel 속성과 itemValue 속성을 사용할 수 있다.
- 스프링이 제공하는 <form:select>, <form:options>, <form:option> 커스텀 태그의 장점은 커맨드 객체의 프로퍼티 값과 일치하는 값을 갖는 <option>을 자동으로 선택해 준다는 점이다.

- 예를 들어 커맨드 객체의 loginType 프로퍼티 값이 "기업회원"이면 다음과 같이 일치하는 <option> 태그에 selected 속성이 추가된다.

체크박스 관련 커스텀 태그 : <form:checkboxes>, <form:checkbox>

- 한 개 이상의 값을 커맨드 객체의 특정 프로퍼티에 저장하고 싶다면 배열이나 List와 같은 타입을 사용해서 값을 저장한다.

```
public class MemberRegisterRequest {
        private String[] favoriteOs;

        public String[] getFavoriteOs() {
                return favoriteOs;
        }

        public void setFavoriteOs(String[] favoriteOs) {
                this.favoriteOs = favoriteOs;
        }

        ...
}
```

- HTML 입력 폼에서는 checkbox 타입의 <input> 태그를 이용해서 한 개 이상의 값을 선택할 수 있도록 한다.

```
<input type="checkbox" name="favoriteOs" value="윈도우8">윈도우8</input>
<input type="checkbox" name="favoriteOs" value="윈도우10">윈도우10</input>
```

- 스프링은 checkbox 타입의 <input> 태그와 관련하여 다음과 같은 커스텀 태그를 제공한다.

checkbox 타입의 <input> 태그와 관련된 커스텀 태그

커스텀 태그	설명
<form:checkboxes>	커맨드 객체의 특정 프로퍼티와 관련된 checkbox 타입의 <input> 태그 목록을 생성한다.
<form:checkbox>	커맨드 객체의 특정 프로퍼티와 관련된 한 개의 checkbox 타입 <input> 태그를 생성한다.

- <form:checkboxes> 커스텀 태그는 items 속성을 이용하여 값으로 사용할 콜렉션을 지정한다.
- path 속성으로 커맨드 객체의 프로퍼티를 지정한다. 아래 코드는 <form:checkboxes> 커스텀 태그의 사용 예이다.

```
<p>
        <label>선호 OS</label>
        <form:checkboxes items="${favoriteOsNames}" path="favoriteOs" />
</p>
```

- favoriteOsNames 모델의 값이 {"윈도우8", "윈도우10") 일 경우 위 코드의 <form:checkboxes> 커스텀 태그는 다음과 같은 HTML 코드를 생성한다.

```
<span>
        <input id="favoriteOs1" name="favoriteOs" type="checkbox" value="윈
        <label for="favoriteOs1">윈도우8</label>
</span>
<span>
        <input id="favoriteOs2" name="favoriteOs" type="checkbox" value="윈
        <label for="favoriteOs2">윈도우10</label>
</span>
<input type="hidden" name="_favoriteOs" value="on" />
```

- <input>태그의 value 속성에 사용한 값이 체크박스를 위한 텍스트로 사용되고 있다.
- <option> 태그와 마찬가지로 콜렉션에 저장된 객체가 String이 아니면 item Value 속성과 itemLabel 속성을 이용해서 값과 텍스트로 사용할 객체의 프로퍼티를 지정한다.

```
<p>
        <label>선호 OS</label>
        <form:checkboxes items="${favoriteOsCodes}" path="favoriteOs" i
                        temValue="code" itemsLabel="label" />
</p>
```

- <form:checkbox> 커스텀 태그는 한 개의 checkbox 타입의 <input> 태그를 한 개 생성할 때 사용된다.
- <form:checkbox> 커스텀 태그는 value 속성과 label 속성을 사용해서 값과 텍스트를 설정한다.

```
<form:checkbox path="favoriteOs" value="WIN8" label="윈도우8" />
<form:checkbox path="favoriteOs" value="WIN10" label="윈도우10" />
```

- <form:checkbox> 커스텀 태그는 연결되는 값 타입에 따라 처리 방식이 달라진다.
- 다음 코드를 보자. 이 코드는 boolean 타입의 프로퍼티를 포함한다.

```
public class MemberRegisterRequest {

        private boolean allowNoti;
```

```
        public boolean isAllowNoti() {
                return allowNoti;
        }

        public void setAllowNoti(boolean allowNoti) {
                this.allowNoti = allowNoti;
        }
        ...
}
```

- <form:checkbox>는 연결되는 프로퍼티 값이 true이면 "checked" 속성을 설정한다. false이면 "checked" 속성을 설정하지 않는다.
- 또한 생성되는 <input> 태그의 value 속성값은 "true가 된다. 아래 코드는 사용 예다.

```
<form:checkbox path="allowNoti" label="이메일을 수신합니다." />
```

- allowNoti의 값이 false와 true인 경우 각각 생성되는 HTML 코드는 다음과 같다.

```
<!-- allowNoti false인 경우 -->
<input id="allowNoti1" name="allowNoti" type="checkbox" value="true"/>
<label for="allowNoti1">이메일을 수신합니다.</label>
<input type="hidden" name="_allowNoti" value="on"/>

<!-- allowNoti true인 경우 -->
<input id="allowNoti`" name="allowNoti" type="checkbox" value="true" checke
<label for="allowNoti1">이메일을 수신합니다.</label>
<input type="hidden" name="_allowNoti" value="on"/>
```

- <form:checkbox>태그는 프로퍼티가 배열이나 Collection일 경우 해당 콜렉션에 값이 포함되어 있다면 "checked" 속성을 설정한다.
- 예를 들어 아래와 같은 배열 타입의 프로퍼티가 있다고 해보자.

```
public class MemberRegisterRequest {

        private String[] favoriteOs;

        public String[] getFavoriteOs() {
                return favoriteOs;
        }

        public void setFavoriteOs(String[] favoriteOs) {
                this.favoriteOs = favoriteOs;
        }
```

```
        ...
}
```

- <form:checkbox> 커스텀 태그를 사용하면 다음과 같이 favoriteOs 프로퍼티에 대한 폼을 처리할 수 있다.

```
<form:checkbox path="favoriteOs" value="윈도우8" label="윈도우8" />
<form:checkbox path="favoriteOs" value="윈도우10" label="윈도우10" />
```

라디오버튼 관련 커스텀 태그: <form:radiobuttons>, <form:radiobutton>

- 여러 가지 옵션 중에서 한 가지를 선택해야 하는 경우 radio 타입의 <input> 태그를 사용한다.
- 스프링은 radio 타입의 <input> 태그와 관련하여 다음과 같은 커스텀 태그를 제공하고 있다.

커스텀 태그	설명
<form:radiobuttons>	커맨드 객체의 특정 프로퍼티와 관련된 radio 타입의 <input> 태그 목록을 생성한다.
<form:radiobutton>	커맨드 객체의 특정 프로퍼티와 관련한 한 개의 radio 타입 <input> 태그를 생성한다.

- <form:radiobuttons> 커스텀 태그는 다음과 같이 items 속성에 값으로 사용할 콜렉션을 전달받고 path 속성에 커맨드 객체의 프로퍼티를 지정한다.

```
<p>
        <label>주로 사용하는 개발툴</label>
        <form:radiobuttons items="${tools}" path="tool" />
</p>
```

- <form.radiobuttons> 커스텀 태그는 다음과 같은 HTML 태그를 생성한다.

```
<span>
        <input id="tool1" name="tool" type="radio" value="Eclipse" />
        <label for="tool1">Eclipse</label>
</span>
<span>
        <input id="tool2" name="tool" type="radio" value="IntelliJ" />
        <label for="tool2">IntelliJ</label>
</span>
<span>
        <input id="tool3" name="tool" type="radio" value="NetBeans" />
```

```
        <label for="tool3">NetBeans</label>
</span>
```

- <form:radiobutton> 커스텀 태그는 1개의 radio 타입 <input> 태그를 생성할 때 사용되며 value 속성과 label 속성을 이용하여 값과 텍스트를 설정한다.
- 사용 방법은 <form:checkbox> 태그와 동일하다.

<textarea> 태그를 위한 커스텀 태그 : <form:textarea>

- 게시글 내용과 같이 여러 줄을 입력받아야 하는 경우 <textarea> 태그를 사용한다.
- 스프링은 <form:textarea> 커스텀 태그를 제공하고 있다.
- 이 태그를 이용하면 커맨드 객체와 관련된 <textarea> 태그를 생성할 수 있다. 다음 코드는 사용 예이다.

```
<p>
        <label for="etc">기타</label>
        <form:textarea path="etc" cols="20" rows="3" />
</p>
```

- <form:textarea> 커스텀 태그가 생성하는 HTML 태그는 다음과 같다.

```
<p>
        <label for="etc">기타</label>
        <textarea id="etc" name="etc" rows="3" cols="20"></textarea>
</p>
```

CSS 및 HTML 태그와 관련된 공통 속성

- <form:input>, <form:select> 등 입력 폼과 관련해서 제공하는 스프링 커스텀 태그는 HTML의 CSS 및 이벤트 관련 속성을 제공하고 있다. CSS와 관련된 속성은 다음과 같다.
    - cssClass: HTML의 class 속성값
    - cssErrorClass : 폼 검증 에러가 발생했을 때 사용할 HTML의 class 속성값
    - cssStyle: HTML의 style 속성값
- 스프링은 폼 검증 기능을 제공하는데 cssErrorClass 속성은 이와 관련된 것이다.
- HTML 태그가 사용하는 다음 속성도 사용 가능하다.
    - id, title, dir
    - disabled, tabindex
    - fonfocus, onblur, onchange onclick, ondblclick
    - onkeydown, onkeypress, onkeyup

    - onmousedown, onmousemove, onmouseup
    - onmouseout, onmouseover
- 또한 각 커스텀 태그는 htmlEscape 속성을 사용해서 커맨드 객체의 값에 포함된 HTML 특수 문자를 엔티티 레퍼런스로 변환할지를 결정할 수 있다.