

0. 스프링 MVC 전체 목차

#인강/5. 스프링 MVC 2/강의#

- 인프런 강의
- 저자: 김영한

전체 목차

- 1. 타임리프 - 기본 기능
- 2. 타임리프 - 스프링 통합과 폼
- 3. 메시지, 국제화
- 4. 검증1 - Validation
- 5. 검증2 - Bean Validation
- 6. 로그인 처리1 - 쿠키, 세션
- 7. 로그인 처리2 - 필터, 인터셉터
- 8. 예외 처리와 오류 페이지
- 9. API 예외 처리
- 10. 스프링 타입 컨버터
- 11. 파일 업로드

사용 예제

- 1. 타임리프 - 기본 기능 - thymeleaf-basic
- 2. 타임리프 - 스프링 통합과 폼 - form-start → form
- 3. 메시지, 국제화 - message-start → message
- 4. 검증1 - Validation - validation-start → validation
- 5. 검증2 - Bean Validation - validation
- 6. 로그인 처리1 - 쿠키, 세션 - login-start → login
- 7. 로그인 처리2 - 필터, 인터셉터 - login
- 8. 예외 처리와 오류 페이지 - exception
- 9. API 예외 처리 - exception
- 10. 스프링 타입 컨버터 - type-converter
- 11. 파일 업로드 - upload

버전 수정 이력

2021-07-18

- 오탏 수정(rere님 도움)
- 영상 발음 수정(김 재연님 도움)
- 오탏 수정(hanul_kr님 도움)
- 오탏 수정(opensesame님 도움)
- 오탏 수정(나현창님 도움)
- 오탏 수정(yellowsunn님 도움)

2021-06-30

- 오탏 수정 (SeJongDeveloper님 도움)
 - th:attrappend 관련 오탏

2021-06-24-2

- 오탏 수정 (Dokkabei97님 도움)

2021-06-24

- 릴리즈

1. 타임리프 - 기본 기능

#인강/5. 스프링 MVC 2/강의#

목차

- 1. 타임리프 - 기본 기능 - 프로젝트 생성
- 1. 타임리프 - 기본 기능 - 타임리프 소개
- 1. 타임리프 - 기본 기능 - 텍스트 - text, utext
- 1. 타임리프 - 기본 기능 - 변수 - SpringEL
- 1. 타임리프 - 기본 기능 - 기본 객체들
- 1. 타임리프 - 기본 기능 - 유ти리티 객체와 날짜
- 1. 타임리프 - 기본 기능 - URL 링크
- 1. 타임리프 - 기본 기능 - 리터럴
- 1. 타임리프 - 기본 기능 - 연산
- 1. 타임리프 - 기본 기능 - 속성 값 설정
- 1. 타임리프 - 기본 기능 - 반복
- 1. 타임리프 - 기본 기능 - 조건부 평가
- 1. 타임리프 - 기본 기능 - 주석
- 1. 타임리프 - 기본 기능 - 블록
- 1. 타임리프 - 기본 기능 - 자바스크립트 인라인
- 1. 타임리프 - 기본 기능 - 템플릿 조각
- 1. 타임리프 - 기본 기능 - 템플릿 레이아웃1
- 1. 타임리프 - 기본 기능 - 템플릿 레이아웃2
- 1. 타임리프 - 기본 기능 - 정리

프로젝트 생성

사전 준비물

- Java 11 설치
- IDE: IntelliJ 또는 Eclipse 설치

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

<https://start.spring.io>

- 프로젝트 선택

- Project: Gradle Project
- Language: Java

- Spring Boot: 2.5.x
- Project Metadata
 - Group: hello
 - Artifact: thymeleaf-basic
 - Name: thyme-leaf-basic
 - Package name: **hello.thymeleaf**
 - Packaging: **Jar**
 - Java: 11
- Dependencies: **Spring Web, Lombok , Thymeleaf**

build.gradle

```

plugins {
    id 'org.springframework.boot' version '2.5.0'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

```

```
}
```

```
test {
    useJUnitPlatform()
}
```

- 동작 확인

- 기본 메인 클래스 실행(ThymeleafBasicApplication.main())
- <http://localhost:8080> 호출해서 Whitelabel Error Page가 나오면 정상 동작

홈 화면

```
/resources/static/index.html
```

```
<html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<ul>
    <li>텍스트
        <ul>
            <li><a href="/basic/text-basic">텍스트 출력 기본</a></li>
            <li><a href="/basic/text-unescaped">텍스트 text, utext</a></li>
        </ul>
    </li>
    <li>표준 표현식 구문
        <ul>
            <li><a href="/basic/variable">변수 - SpringEL</a></li>
            <li><a href="/basic/basic-objects?paramData=HelloParam">기본 객체들</a></li>
            <li><a href="/basic/date">유ти리티 객체와 날짜</a></li>
            <li><a href="/basic/link">링크 URL</a></li>
            <li><a href="/basic/literal">리터럴</a></li>
            <li><a href="/basic/operation">연산</a></li>
        </ul>
    </li>
</ul>
```

```

<li>속성 값 설정
    <ul>
        <li><a href="/basic/attribute">속성 값 설정</a></li>
    </ul>
</li>
<li>반복
    <ul>
        <li><a href="/basic/each">반복</a></li>
    </ul>
</li>
<li>조건부 평가
    <ul>
        <li><a href="/basic/condition">조건부 평가</a></li>
    </ul>
</li>
<li>주석 및 블록
    <ul>
        <li><a href="/basic/comments">주석</a></li>
        <li><a href="/basic/block">블록</a></li>
    </ul>
</li>
<li>자바스크립트 인라인
    <ul>
        <li><a href="/basic/javascript">자바스크립트 인라인</a></li>
    </ul>
</li>
<li>템플릿 레이아웃
    <ul>
        <li><a href="/template/fragment">템플릿 조각</a></li>
        <li><a href="/template/layout">유연한 레이아웃</a></li>
        <li><a href="/template/layoutExtend">레이아웃 상속</a></li>
    </ul>
</li>
</ul>
</body>
</html>

```

실행

<http://localhost:8080>

IntelliJ Gradle 대신에 자바 직접 실행

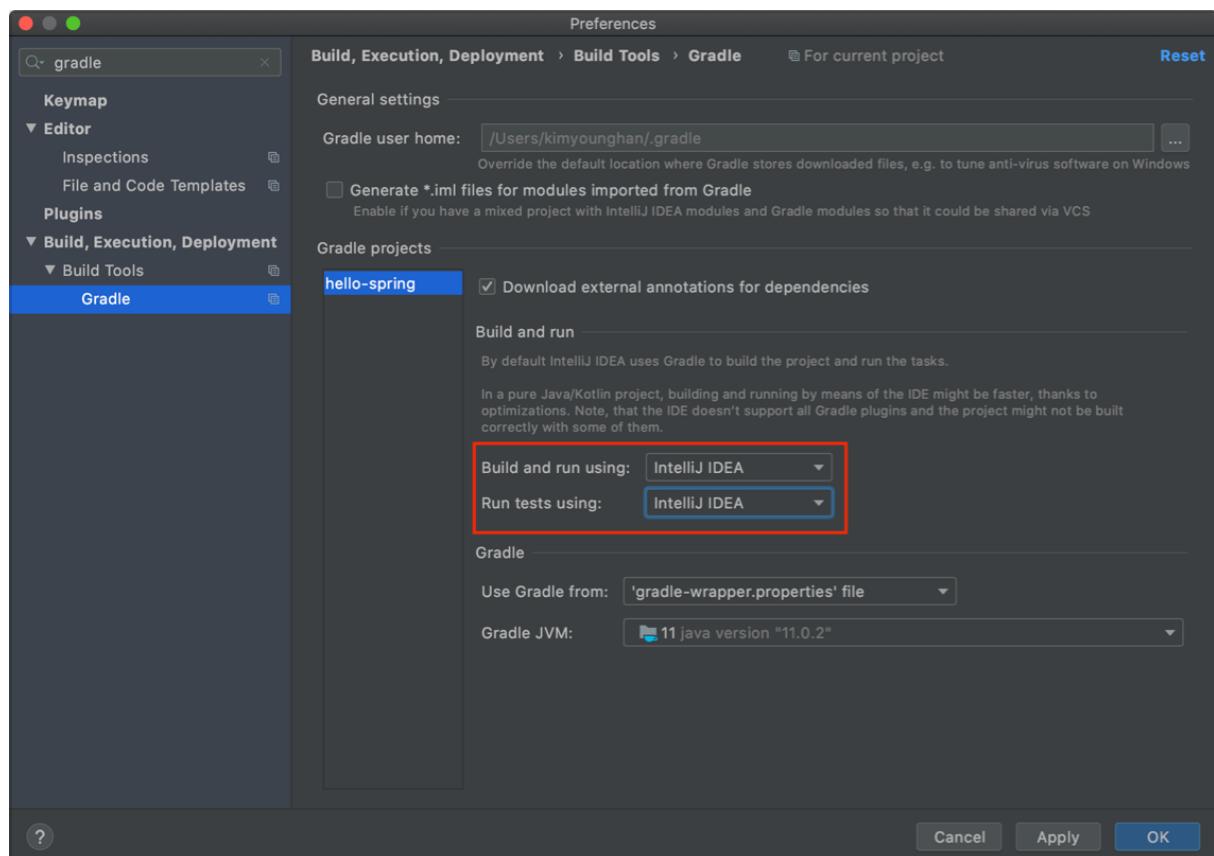
최근 IntelliJ 버전은 Gradle을 통해서 실행 하는 것이 기본 설정이다. 이렇게 하면 실행속도가 느리다.
다음과 같이 변경하면 자바로 바로 실행해서 실행속도가 더 빠르다

- Preferences → Build, Execution, Deployment → Build Tools → Gradle
 - Build and run using: Gradle → IntelliJ IDEA
 - Run tests using: Gradle → IntelliJ IDEA

원도우 사용자

File → Setting

설정 이미지



롬복 적용

1. Preferences → plugin → lombok 검색 실행 (재시작)
2. Preferences → Annotation Processors 검색 → Enable annotation processing 체크 (재시작)
3. 임의의 테스트 클래스를 만들고 @Getter, @Setter 확인

원도우 사용자

File → Setting

Postman을 설치하자

다음 사이트에서 Postman을 다운로드 받고 설치해두자

- <https://www.postman.com/downloads>

타임리프 소개

- 공식 사이트: <https://www.thymeleaf.org/>
- 공식 메뉴얼 - 기본 기능: <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html>
- 공식 메뉴얼 - 스프링 통합: <https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html>

이전 강의인 스프링 MVC 1편에서 타임리프를 간단히 사용해보고, 그 특징들도 알아보았다.

이번 시간에는 타임리프의 개념은 간단히 소개하고, 실제 동작하는 기본 기능 위주로 알아보겠다.

타임리프 특징

- 서버 사이드 HTML 렌더링 (SSR)
- 네츄럴 템플릿
- 스프링 통합 지원

서버 사이드 HTML 렌더링 (SSR)

타임리프는 백엔드 서버에서 HTML을 동적으로 렌더링 하는 용도로 사용된다.

네츄럴 템플릿

타임리프는 순수 HTML을 최대한 유지하는 특징이 있다.

타임리프로 작성한 파일은 HTML을 유지하기 때문에 웹 브라우저에서 파일을 직접 열어도 내용을 확인할 수 있고, 서버를 통해 뷰 템플릿을 거치면 동적으로 변경된 결과를 확인할 수 있다.

JSP를 포함한 다른 뷰 템플릿들은 해당 파일을 열면, 예를 들어서 JSP 파일 자체를 그대로 웹 브라우저에서 열어보면 JSP 소스코드와 HTML이 뒤죽박죽 섞여서 웹 브라우저에서 정상적인 HTML 결과를 확인할 수 없다. 오직 서버를 통해서 JSP가 렌더링 되고 HTML 응답 결과를 받아야 화면을 확인할 수 있다.

반면에 타임리프로 작성된 파일은 해당 파일을 그대로 웹 브라우저에서 열어도 정상적인 HTML 결과를 확인할 수 있다. 물론 이 경우 동적으로 결과가 렌더링 되지는 않는다. 하지만 HTML 마크업 결과가 어떻게 되는지 파일만 열어도 바로 확인할 수 있다.

이렇게 순수 HTML을 그대로 유지하면서 뷰 템플릿도 사용할 수 있는 타임리프의 특징을 **네츄럴 템플릿** (natural templates)이라 한다.

스프링 통합 지원

타임리프는 스프링과 자연스럽게 통합되고, 스프링의 다양한 기능을 편리하게 사용할 수 있게 지원한다. 이 부분은 스프링 통합과 품 장에서 자세히 알아보겠다.

타임리프 기본 기능

타임리프를 사용하려면 다음 선언을 하면 된다.

타임리프 사용 선언

```
<html xmlns:th="http://www.thymeleaf.org">
```

기본 표현식

타임리프는 다음과 같은 기본 표현식들을 제공한다. 지금부터 하나씩 알아보자.

- 간단한 표현:
 - 변수 표현식: \${...}
 - 선택 변수 표현식: *{...}
 - 메시지 표현식: #{...}
 - 링크 URL 표현식: @{...}
 - 조각 표현식: ~{...}
- 리터럴
 - 텍스트: 'one text', 'Another one!',...
 - 숫자: 0, 34, 3.0, 12.3,...
 - 불린: true, false
 - 널: null
 - 리터럴 토큰: one, sometext, main,...
- 문자 연산:
 - 문자 합치기: +
 - 리터럴 대체: |The name is \${name}|
- 산술 연산:
 - Binary operators: +, -, *, /, %
 - Minus sign (unary operator): -

- 불린 연산:
 - Binary operators: and, or
 - Boolean negation (unary operator): !, not
- 비교와 동등:
 - 비교: >, <, >=, <= (gt, lt, ge, le)
 - 동등 연산: ==, != (eq, ne)
- 조건 연산:
 - If-then: (if) ? (then)
 - If-then-else: (if) ? (then) : (else)
 - Default: (value) ?: (defaultvalue)
- 특별한 토큰:
 - No-Operation: _

참고: <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#standard-expression-syntax>

텍스트 - text, utext

타임리프의 가장 기본 기능인 텍스트를 출력하는 기능 먼저 알아보자.

타임리프는 기본적으로 HTML 테그의 속성에 기능을 정의해서 동작한다. HTML의 콘텐츠(content)에 데이터를 출력할 때는 다음과 같이 `th:text` 를 사용하면 된다.

```
<span th:text="${data}">
```

HTML 테그의 속성이 아니라 HTML 콘텐츠 영역안에서 직접 데이터를 출력하고 싶으면 다음과 같이

`[[...]]` 를 사용하면 된다.

콘텐츠 안에서 직접 출력하기 = `[$data]]`

BasicController

```
package hello.thymeleaf.basic;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
```

```

@Controller
@RequestMapping("/basic")
public class BasicController {

    @GetMapping("/text-basic")
    public String textBasic(Model model) {
        model.addAttribute("data", "Hello Spring!");
        return "basic/text-basic";
    }

}

```

/resources/templates/basic/text-basic.html

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<h1>컨텐츠에 데이터 출력하기</h1>
<ul>
    <li>th:text 사용 <span th:text="${data}"></span></li>
    <li>컨텐츠 안에서 직접 출력하기 = [[${data}]]</li>
</ul>

</body>
</html>

```

실행

- <http://localhost:8080/basic/text-basic>

Escape

HTML 문서는 <, > 같은 특수 문자를 기반으로 정의된다. 따라서 뷰 템플릿으로 HTML 화면을 생성할 때는 출력하는 데이터에 이러한 특수 문자가 있는 것을 주의해서 사용해야 한다.
앞에서 만든 예제의 데이터를 다음과 같이 변경해서 실행해보자.

변경 전

```
"Hello Spring!"
```

변경 후

```
"Hello <b>Spring!</b>"
```

 테그를 사용해서 **Spring!**이라는 단어가 진하게 나오도록 해보자.

웹 브라우저에서 실행결과를 보자.

- 웹 브라우저: Hello Spring!
- 소스보기: Hello Spring!

개발자가 의도한 것은 가 있으면 해당 부분을 강조하는 것이 목적이었다. 그런데 테그가 그대로 나온다.

소스보기를 하면 < 부분이 < 로 변경된 것을 확인할 수 있다.

HTML 엔티티

웹 브라우저는 < 를 HTML 테그의 시작으로 인식한다. 따라서 < 를 테그의 시작이 아니라 문자로 표현할 수 있는 방법이 필요한데, 이것을 HTML 엔티티라 한다. 그리고 이렇게 HTML에서 사용하는 특수 문자를 HTML 엔티티로 변경하는 것을 이스케이프(escape)라 한다. 그리고 타임리프가 제공하는 th:text, [...] 는 기본적으로 이스케이스(**escape**)를 제공한다.

- < → <
- > → >
- 기타 수 많은 HTML 엔티티가 있다.

참고

HTML 엔티티와 관련해서 더 자세한 내용은 HTML 엔티티로 검색해보자.

Unescape

이스케이프 기능을 사용하지 않으려면 어떻게 해야할까?

타임리프는 다음 두 기능을 제공한다.

- `th:text` → `th:utext`
- `[[...]]` → `[(....)]`

BasicController에 추가

```
@GetMapping("/text-unescaped")
public String textUnescaped(Model model) {
    model.addAttribute("data", "Hello <b>Spring!</b>");
    return "basic/text-unescaped";
}
```

/resources/templates/basic/text-unescape.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<h1>text vs utext</h1>
<ul>
    <li>th:text = <span th:text="${data}"></span></li>
    <li>th:utext = <span th:utext="${data}"></span></li>
</ul>

<h1><span th:inline="none">[...] vs [(....)]</span></h1>
<ul>
    <li><span th:inline="none">[...] = </span>[$data]]</li>
    <li><span th:inline="none">[(....)] = </span>[$data])</li>
</ul>

</body>
</html>
```

- `th:inline="none"` : 타임리프는 `[[...]]` 를 해석하기 때문에, 화면에 `[[...]]` 글자를 보여줄 수 없다. 이 테그 안에서는 타임리프가 해석하지 말라는 옵션이다.

실행

- <http://localhost:8080/basic/text-unescape>

실행해보면 다음과 같이 정상 수행되는 것을 확인할 수 있다.

- 웹 브라우저: Hello **Spring!**
- 소스보기: Hello Spring!

주의!

실제 서비스를 개발하다 보면 escape를 사용하지 않아서 HTML이 정상 렌더링 되지 않는 수 많은 문제가 발생한다. escape를 기본으로 하고, 꼭 필요한 때만 unescape를 사용하자.

변수 - SpringEL

타임리프에서 변수를 사용할 때는 변수 표현식을 사용한다.

변수 표현식 : `${...}`

그리고 이 변수 표현식에는 스프링 EL이라는 스프링이 제공하는 표현식을 사용할 수 있다.

BasicController 추가

```
@GetMapping("/variable")
public String variable(Model model) {

    User userA = new User("userA", 10);
    User userB = new User("userB", 20);

    List<User> list = new ArrayList<>();
    list.add(userA);
    list.add(userB);
```

```

Map<String, User> map = new HashMap<>();
map.put("userA", userA);
map.put("userB", userB);

model.addAttribute("user", userA);
model.addAttribute("users", list);
model.addAttribute("userMap", map);

return "basic/variable";
}

@Data
static class User {

    private String username;
    private int age;

    public User(String username, int age) {
        this.username = username;
        this.age = age;
    }
}

```

/resources/templates/basic/variable.html

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<h1>SpringEL 표현식</h1>
<ul>Object
    <li>${user.username} = <span th:text="${user.username}"></span></li>
    <li>${user['username']} = <span th:text="${user['username']}"></span></li>

```

```

<li>${user.getUsername()} = <span th:text="${user.getUsername()}"></span></li>
</ul>
<ul>List
    <li>${users[0].username} = <span th:text="${users[0].username}"></span></li>
    <li>${users[0]['username']} = <span th:text="${users[0]['username']}"></span></li>
    <li>${users[0].getUsername()} = <span th:text="${users[0].getUsername()}"></span></li>
</ul>
<ul>Map
    <li>${userMap['userA'].username} = <span th:text="${userMap['userA'].username}"></span></li>
    <li>${userMap['userA']['username']} = <span th:text="${userMap['userA']['username']}"></span></li>
    <li>${userMap['userA'].getUsername()} = <span th:text="${userMap['userA'].getUsername()}"></span></li>
</ul>
</body>
</html>

```

SpringEL 다양한 표현식 사용

Object

- user.username : user의 username을 프로퍼티 접근 → user.getUsername()
- user['username'] : 위와 같음 → user.getUsername()
- user.getUsername() : user의 getUsername()을 직접 호출

List

- users[0].username : List에서 첫 번째 회원을 찾고 username 프로퍼티 접근 → list.get(0).getUsername()
- users[0]['username'] : 위와 같음
- users[0].getUsername() : List에서 첫 번째 회원을 찾고 메서드 직접 호출

Map

- `userMap['userA'].username`: Map에서 userA를 찾고, username 프로퍼티 접근 →
`map.get("userA").getUsername()`
- `userMap['userA']['username']`: 위와 같음
- `userMap['userA'].getUsername()`: Map에서 userA를 찾고 메서드 직접 호출

실행

- <http://localhost:8080/basic/variable>

지역 변수 선언

`th:with` 를 사용하면 지역 변수를 선언해서 사용할 수 있다. 지역 변수는 선언한 테그 안에서만 사용할 수 있다.

/resources/templates/basic/variable.html 추가

```
<h1>지역 변수 – (th:with)</h1>
<div th:with="first=${users[0]}">
    <p>처음 사람의 이름은 <span th:text="${first.username}"></span></p>
</div>
```

기본 객체들

타임리프는 기본 객체들을 제공한다.

- `#{request}`
- `#{response}`
- `#{session}`
- `#{servletContext}`
- `#{locale}`

그런데 `#request` 는 `HttpServletRequest` 객체가 그대로 제공되기 때문에 데이터를 조회하려면
`request.getParameter("data")` 처럼 불편하게 접근해야 한다.

이런 점을 해결하기 위해 편의 객체도 제공한다.

- HTTP 요청 파라미터 접근: `param`
 - 예) `#{param.paramData}`

- HTTP 세션 접근: `session`
 - 예) `#{session.sessionData}`
- 스프링 빈 접근: `@`
 - 예) `#{@helloBean.hello('Spring!')}`

BasicController 추가

```

@GetMapping("/basic-objects")
public String basicObjects(HttpSession session) {
    session.setAttribute("sessionData", "Hello Session");
    return "basic/basic-objects";
}

@Component("helloBean")
static class HelloBean {
    public String hello(String data) {
        return "Hello " + data;
    }
}

```

/resources/templates/basic/basic-objects.html

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<h1>식 기본 객체 (Expression Basic Objects)</h1>
<ul>
    <li>request = <span th:text="#{request}"></span></li>
    <li>response = <span th:text="#{response}"></span></li>
    <li>session = <span th:text="#{session}"></span></li>
    <li>servletContext = <span th:text="#{servletContext}"></span></li>
    <li>locale = <span th:text="#{locale}"></span></li>

```

```

</ul>

<h1>편의 객체</h1>

<ul>
    <li>Request Parameter = <span th:text="${param.paramData}"></span></li>
    <li>session = <span th:text="${session.sessionData}"></span></li>
    <li>spring bean = <span th:text="${@helloBean.hello('Spring!')}"></span></li>
</ul>

</body>
</html>

```

실행

- <http://localhost:8080/basic/basic-objects?paramData=HelloParam>

유틸리티 객체와 날짜

타임리프는 문자, 숫자, 날짜, URI등을 편리하게 다루는 다양한 유틸리티 객체들을 제공한다.

타임리프 유틸리티 객체들

- #message : 메시지, 국제화 처리
- #uris : URI 이스케이프 지원
- #dates : java.util.Date 서식 지원
- #calendars : java.util.Calendar 서식 지원
- #temporals : 자바8 날짜 서식 지원
- #numbers : 숫자 서식 지원
- #strings : 문자 관련 편의 기능
- #objects : 객체 관련 기능 제공
- #bools : boolean 관련 기능 제공
- #arrays : 배열 관련 기능 제공
- #lists, #sets, #maps : 컬렉션 관련 기능 제공
- #ids : 아이디 처리 관련 기능 제공, 뒤에서 설명

타임리프 유틸리티 객체

- <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#expression-utility-objects>

유ти리티 객체 예시

- <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#appendix-b-expression-utility-objects>

참고

이런 유ти리티 객체들은 대략 이런 것이 있다 알아두고, 필요할 때 찾아서 사용하면 된다.

자바8 날짜

타임리프에서 자바8 날짜인 `LocalDate`, `LocalDateTime`, `Instant`를 사용하려면 추가 라이브러리가 필요하다. 스프링 부트 타임리프를 사용하면 해당 라이브러리가 자동으로 추가되고 통합된다.

타임리프 자바8 날짜 지원 라이브러리

`thymeleaf-extras-java8time`

자바8 날짜용 유ти리티 객체

`#temporals`

사용 예시

```
<span th:text="#{#temporals.format(localDateTime, 'yyyy-MM-dd HH:mm:ss')}"></span>
```

BasicController 추가

```
@GetMapping("/date")
public String date(Model model) {
    model.addAttribute("localDateTime", LocalDateTime.now());
    return "basic/date";
}
```

`/resources/templates/basic/date.html`

```
<!DOCTYPE html>
```

```

<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<h1>LocalDateTime</h1>
<ul>
    <li>default = <span th:text="${localDateTime}"></span></li>
    <li>yyyy-MM-dd HH:mm:ss = <span th:text="#{temporals.format(localDateTime,
'yyyy-MM-dd HH:mm:ss')}"></span></li>
</ul>

<h1>LocalDateTime - Utils</h1>
<ul>
    <li>${#temporals.day(localDateTime)} = <span th:text="$
{#temporals.day(localDateTime)}"></span></li>
    <li>${#temporals.month(localDateTime)} = <span th:text="$
{#temporals.month(localDateTime)}"></span></li>
    <li>${#temporals.monthName(localDateTime)} = <span th:text="$
{#temporals.monthName(localDateTime)}"></span></li>
    <li>${#temporals.monthNameShort(localDateTime)} = <span th:text="$
{#temporals.monthNameShort(localDateTime)}"></span></li>
    <li>${#temporals.year(localDateTime)} = <span th:text="$
{#temporals.year(localDateTime)}"></span></li>
    <li>${#temporals.dayOfWeek(localDateTime)} = <span th:text="$
{#temporals.dayOfWeek(localDateTime)}"></span></li>
    <li>${#temporals.dayOfWeekName(localDateTime)} = <span th:text="$
{#temporals.dayOfWeekName(localDateTime)}"></span></li>
    <li>${#temporals.dayOfWeekNameShort(localDateTime)} = <span th:text="$
{#temporals.dayOfWeekNameShort(localDateTime)}"></span></li>
    <li>${#temporals.hour(localDateTime)} = <span th:text="$
{#temporals.hour(localDateTime)}"></span></li>
    <li>${#temporals.minute(localDateTime)} = <span th:text="$
{#temporals.minute(localDateTime)}"></span></li>
    <li>${#temporals.second(localDateTime)} = <span th:text="$
{#temporals.second(localDateTime)}"></span></li>
    <li>${#temporals.nanosecond(localDateTime)} = <span th:text="$
{#temporals.nanosecond(localDateTime)}"></span></li>

```

```

{#temporals.nanosecond(localDateTime)}"></span></li>
</ul>

</body>
</html>

```

URL 링크

타임리프에서 URL을 생성할 때는 `@{...}` 문법을 사용하면 된다.

BasicController 추가

```

@GetMapping("/link")
public String link(Model model) {
    model.addAttribute("param1", "data1");
    model.addAttribute("param2", "data2");
    return "basic/link";
}

```

/resources/templates/basic/link.html

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1>URL 링크</h1>
<ul>
    <li><a th:href="@{/hello}">basic url</a></li>
    <li><a th:href="@{/hello(param1=${param1}, param2=${param2})}">hello query
param</a></li>
    <li><a th:href="@{/hello/{param1}/{param2}(param1=${param1}, param2=$
{param2})}">path variable</a></li>

```

```

<li><a th:href="@{/hello/{param1}(param1=${param1}, param2=${param2})}">path variable + query parameter</a></li>
</ul>
</body>
</html>

```

단순한 URL

- `@{/hello}` → `/hello`

쿼리 파라미터

- `@{/hello(param1=${param1}, param2=${param2})}`
 - → `/hello?param1=data1¶m2=data2`
 - ()에 있는 부분은 쿼리 파라미터로 처리된다.

경로 변수

- `@{/hello/{param1}/{param2}(param1=${param1}, param2=${param2})}`
 - → `/hello/data1/data2`
 - URL 경로상에 변수가 있으면 () 부분은 경로 변수로 처리된다.

경로 변수 + 쿼리 파라미터

- `@{/hello/{param1}(param1=${param1}, param2=${param2})}`
 - → `/hello/data1?param2=data2`
 - 경로 변수와 쿼리 파라미터를 함께 사용할 수 있다.

상대경로, 절대경로, 프로토콜 기준을 표현할 수 도 있다.

- `/hello`: 절대 경로
- `hello`: 상대 경로

참고: <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#link-urls>

리터럴

Literals

리터럴은 소스 코드상에 고정된 값을 말하는 용어이다.

예를 들어서 다음 코드에서 "Hello" 는 문자 리터럴, 10, 20 는 숫자 리터럴이다.

```
String a = "Hello"  
int a = 10 * 20
```

참고

이 내용이 쉬워 보이지만 처음 타임리프를 사용하면 많이 실수하니 잘 보아두자.

타임리프는 다음과 같은 리터럴이 있다.

- 문자: 'hello'
- 숫자: 10
- 불린: true, false
- null: null

타임리프에서 문자 리터럴은 항상 ' (작은 따옴표)로 감싸야 한다.

```
<span th:text="'hello'">
```

그런데 문자를 항상 ' 로 감싸는 것은 너무 귀찮은 일이다. 공백 없이 쭉 이어진다면 하나의 의미있는 토큰으로 인지해서 다음과 같이 작은 따옴표를 생략할 수 있다.

룰: A-Z, a-z, 0-9, [], ., -, _

```
<span th:text="hello">
```

오류

```
<span th:text="hello world!"></span>
```

문자 리터럴은 원칙상 ' 로 감싸야 한다. 중간에 공백이 있어서 하나의 의미있는 토큰으로도 인식되지 않는다.

수정

```
<span th:text="'hello world!'"></span>
```

이렇게 ' 로 감싸면 정상 동작한다.

BasicController 추가

```
@GetMapping("/literal")
```

```
public String literal(Model model) {  
    model.addAttribute("data", "Spring!");  
    return "basic/literal";  
}
```

/resources/templates/basic/literal.html

```
<!DOCTYPE html>  
<html xmlns:th="http://www.thymeleaf.org">  
<head>  
    <meta charset="UTF-8">  
    <title>Title</title>  
</head>  
<body>  
    <h1>리터럴</h1>  
    <ul>  
        <!--주의! 다음 주석을 풀면 예외가 발생함-->  
        <!--     <li>"hello world!" = <span th:text="hello world!"></span></li>-->  
        <li>'hello' + ' world!' = <span th:text="'hello' + ' world!'"></span></li>  
        <li>'hello world!' = <span th:text="'"hello world!'"></span></li>  
        <li>'hello ' + ${data} = <span th:text="'"hello ' + ${data}"></span></li>  
        <li>리터럴 대체 |hello ${data}| = <span th:text="|hello ${data}|"></span></li>  
    </ul>  
  
</body>  
</html>
```

리터럴 대체(Literal substitutions)

```
<span th:text="|hello ${data}|">
```

마지막의 리터럴 대체 문법을 사용하면 마치 템플릿을 사용하는 것처럼 편리하다.

연산

타임리프 연산은 자바와 크게 다르지 않다. HTML안에서 사용하기 때문에 HTML 엔티티를 사용하는 부분만 주의하자.

BasicController 추가

```
@GetMapping("/operation")
public String operation(Model model) {
    model.addAttribute("nullData", null);
    model.addAttribute("data", "Spring!");
    return "basic/operation";
}
```

/resources/templates/basic/operation.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<ul>
    <li>산술 연산
        <ul>
            <li>10 + 2 = <span th:text="10 + 2"></span></li>
            <li>10 % 2 == 0 = <span th:text="10 % 2 == 0"></span></li>
        </ul>
    </li>
    <li>비교 연산
        <ul>
            <li>1 > 10 = <span th:text="1 &gt; 10"></span></li>
            <li>1 gt 10 = <span th:text="1 gt 10"></span></li>
            <li>1 >= 10 = <span th:text="1 >= 10"></span></li>
            <li>1 ge 10 = <span th:text="1 ge 10"></span></li>
            <li>1 == 10 = <span th:text="1 == 10"></span></li>
            <li>1 != 10 = <span th:text="1 != 10"></span></li>
        </ul>
    </li>
</ul>
```

```

</li>
<li>조건식
    <ul>
        <li>(10 % 2 == 0)? '짝수': '홀수' = <span th:text="(10 % 2 == 0)? '짝수': '홀수'"></span></li>
    </ul>
</li>
<li>Elvis 연산자
    <ul>
        <li>${data}?: '데이터가 없습니다.' = <span th:text="${data}?: '데이터가 없습니다.'"></span></li>
        <li>${nullData}?: '데이터가 없습니다.' = <span th:text="${nullData}?: '데이터가 없습니다.'"></span></li>
    </ul>
</li>
<li>No-Operation
    <ul>
        <li>${data}?: _ = <span th:text="${data}?: _">데이터가 없습니다.</span></li>
        <li>${nullData}?: _ = <span th:text="${nullData}?: _">데이터가 없습니다.</span></li>
    </ul>
</li>
</ul>

</body>
</html>

```

- **비교연산**: HTML 엔티티를 사용해야 하는 부분을 주의하자,
 - > (gt), < (lt), >= (ge), <= (le), ! (not), == (eq), != (neq, ne)
- **조건식**: 자바의 조건식과 유사하다.
- **Elvis 연산자**: 조건식의 편의 버전
- **No-Operation**: _ 인 경우 마치 타임리프가 실행되지 않는 것처럼 동작한다. 이것을 잘 사용하면 HTML의 내용 그대로 활용할 수 있다. 마지막 예를 보면 데이터가 없습니다. 부분이 그대로 출력된다.

속성 값 설정

타임리프 태그 속성(Attribute)

타임리프는 주로 HTML 태그에 `th:*` 속성을 지정하는 방식으로 동작한다. `th:*`로 속성을 적용하면 기존 속성을 대체한다. 기존 속성이 없으면 새로 만든다.

BasicController 추가

```
@GetMapping("/attribute")
public String attribute() {
    return "basic/attribute";
}
```

/resources/templates/basic/attribute.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<h1>속성 설정</h1>
<input type="text" name="mock" th:name="userA" />

<h1>속성 추가</h1>
- th:attrappend = <input type="text" class="text" th:attrappend="class='large'" /><br/>
- th:attrprepend = <input type="text" class="text" th:attrprepend="class='large'" /><br/>
- th:classappend = <input type="text" class="text" th:classappend="large" /><br/>

<h1>checked 처리</h1>
- checked o <input type="checkbox" name="active" th:checked="true" /><br/>
```

```
- checked x <input type="checkbox" name="active" th:checked="false" /><br/>
- checked=false <input type="checkbox" name="active" checked="false" /><br/>

</body>
</html>
```

속성 설정

th:* 속성을 지정하면 타임리프는 기존 속성을 th:*로 지정한 속성으로 대체한다. 기존 속성이 없다면 새로 만든다.

```
<input type="text" name="mock" th:name="userA" />
→ 타임리프 렌더링 후 <input type="text" name="userA" />
```

속성 추가

th:attrappend : 속성 값의 뒤에 값을 추가한다.

th:attrprepend : 속성 값의 앞에 값을 추가한다.

th:classappend : class 속성에 자연스럽게 추가한다.

checked 처리

HTML에서는 <input type="checkbox" name="active" checked="false" /> → 0| 경우에도 checked 속성이 있기 때문에 checked 처리가 되어버린다.

HTML에서 checked 속성은 checked 속성의 값과 상관없이 checked라는 속성만 있어도 체크가 된다.
이런 부분이 true, false 값을 주로 사용하는 개발자 입장에서는 불편하다.

타임리프의 th:checked는 값이 false인 경우 checked 속성 자체를 제거한다.

```
<input type="checkbox" name="active" th:checked="false" />
→ 타임리프 렌더링 후: <input type="checkbox" name="active" />
```

반복

타임리프에서 반복은 th:each를 사용한다. 추가로 반복에서 사용할 수 있는 여러 상태 값을 지원한다.

BasicController 추가

```
@GetMapping("/each")
```

```

public String each(Model model) {
    addUsers(model);
    return "basic/each";
}

private void addUsers(Model model) {
    List<User> list = new ArrayList<>();
    list.add(new User("userA", 10));
    list.add(new User("userB", 20));
    list.add(new User("userC", 30));

    model.addAttribute("users", list);
}

```

/resources/templates/basic/each.html

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1>기본 테이블</h1>
<table border="1">
    <tr>
        <th>username</th>
        <th>age</th>
    </tr>
    <tr th:each="user : ${users}">
        <td th:text="${user.username}">username</td>
        <td th:text="${user.age}">0</td>
    </tr>
</table>

<h1>반복 상태 유지</h1>

```

```

<table border="1">
    <tr>
        <th>count</th>
        <th>username</th>
        <th>age</th>
        <th>etc</th>
    </tr>
    <tr th:each="user, userStat : ${users}">
        <td th:text="${userStat.count}">username</td>
        <td th:text="${user.username}">username</td>
        <td th:text="${user.age}">0</td>
        <td>
            index = <span th:text="${userStat.index}"></span>
            count = <span th:text="${userStat.count}"></span>
            size = <span th:text="${userStat.size}"></span>
            even? = <span th:text="${userStat.even}"></span>
            odd? = <span th:text="${userStat.odd}"></span>
            first? = <span th:text="${userStat.first}"></span>
            last? = <span th:text="${userStat.last}"></span>
            current = <span th:text="${userStat.current}"></span>
        </td>
    </tr>
</table>

</body>
</html>

```

반복 기능

- <tr th:each="user : \${users}">
- 반복시 오른쪽 컬렉션(\${users})의 값을 하나씩 꺼내서 왼쪽 변수(user)에 담아서 태그를 반복 실행합니다.
- th:each 는 List 뿐만 아니라 배열, java.util.Iterable, java.util.Enumeration 을 구현한 모든 객체를 반복에 사용할 수 있습니다. Map 도 사용할 수 있는데 이 경우 변수에 담기는 값은 Map.Entry 입니다.

반복 상태 유지

```
<tr th:each="user, userStat : ${users}">
```

반복의 두번째 파라미터를 설정해서 반복의 상태를 확인 할 수 있습니다.

두번째 파라미터는 생략 가능한데, 생략하면 지정한 변수명(`user`) + `Stat` 가 됩니다.

여기서는 `user` + `Stat` = `userStat` 이므로 생략 가능합니다.

반복 상태 유지 기능

- `index` : 0부터 시작하는 값
- `count` : 1부터 시작하는 값
- `size` : 전체 사이즈
- `even`, `odd` : 홀수, 짝수 여부(`boolean`)
- `first`, `last` : 처음, 마지막 여부(`boolean`)
- `current` : 현재 객체

조건부 평가

타임리프의 조건식

`if`, `unless` (`if` 의 반대)

BasicController 추가

```
@GetMapping("/condition")
public String condition(Model model) {
    addUsers(model);
    return "basic/condition";
}
```

/resources/templates/basic/condition.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <h1>if, unless</h1>
```

```

<table border="1">
  <tr>
    <th>count</th>
    <th>username</th>
    <th>age</th>
  </tr>
  <tr th:each="user, userStat : ${users}">
    <td th:text="${userStat.count}">1</td>
    <td th:text="${user.username}">username</td>
    <td>
      <span th:text="${user.age}">0</span>
      <span th:text="미성년자" th:if="${user.age lt 20}"></span>
      <span th:text="미성년자" th:unless="${user.age ge 20}"></span>
    </td>
  </tr>
</table>

<h1>switch</h1>
<table border="1">
  <tr>
    <th>count</th>
    <th>username</th>
    <th>age</th>
  </tr>
  <tr th:each="user, userStat : ${users}">
    <td th:text="${userStat.count}">1</td>
    <td th:text="${user.username}">username</td>
    <td th:switch="${user.age}">
      <span th:case="10">10살</span>
      <span th:case="20">20살</span>
      <span th:case="*>">기타</span>
    </td>
  </tr>
</table>

</body>
</html>

```

if, unless

타임리프는 해당 조건이 맞지 않으면 태그 자체를 렌더링하지 않는다.

만약 다음 조건이 `false`인 경우 `...` 부분 자체가 렌더링 되지 않고 사라진다.

```
<span th:text="'미성년자'" th:if="${user.age lt 20}"></span>
```

switch

* 은 만족하는 조건이 없을 때 사용하는 디폴트이다.

주석

BasicController 추가

```
@GetMapping("/comments")
public String comments(Model model) {
    model.addAttribute("data", "Spring!");
    return "basic/comments";
}
```

/resources/templates/basic/comments.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<h1>예시</h1>
<span th:text="${data}">html data</span>

<h1>1. 표준 HTML 주석</h1>
<!--
<span th:text="${data}">html data</span>
-->
```

```

<h1>2. 타임리프 파서 주석</h1>

<!--/* [[${data}]] */-->

<!--/*-->
<span th:text="${data}">html data</span>
<!--*/-->

<h1>3. 타임리프 프로토타입 주석</h1>

<!--/*
<span th:text="${data}">html data</span>
/*-->

</body>
</html>

```

결과

```

<h1>예시</h1>
<span>Spring!</span>

<h1>1. 표준 HTML 주석</h1>
<!--
<span th:text="${data}">html data</span>
-->

<h1>2. 타임리프 파서 주석</h1>

<h1>3. 타임리프 프로토타입 주석</h1>
<span>Spring!</span>

```

1. 표준 HTML 주석

자바스크립트의 표준 HTML 주석은 타임리프가 렌더링 하지 않고, 그대로 남겨둔다.

2. 타임리프 파서 주석

타임리프 파서 주석은 타임리프의 진짜 주석이다. 렌더링에서 주석 부분을 제거한다.

3. 타임리프 프로토타입 주석

타임리프 프로토타입은 약간 특이한데, HTML 주석에 약간의 구문을 더했다.

HTML 파일을 웹 브라우저에서 그대로 열어보면 HTML 주석이기 때문에 이 부분이 웹 브라우저가 렌더링하지 않는다.

타임리프 렌더링을 거치면 이 부분이 정상 렌더링 된다.

쉽게 이야기해서 HTML 파일을 그대로 열어보면 주석처리가 되지만, 타임리프를 렌더링 한 경우에만 보이는 기능이다.

블록

<th:block> 은 HTML 태그가 아닌 타임리프의 유일한 자체 태그다.

BasicController 추가

```
@GetMapping("/block")
public String block(Model model) {
    addUsers(model);
    return "basic/block";
}
```

/resources/templates/basic/block.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<th:block th:each="user : ${users}">
    <div>
        사용자 이름1 <span th:text="${user.username}"></span>
        사용자 나이1 <span th:text="${user.age}"></span>
    </div>
    <div>
        요약 <span th:text="${user.username} + ' / ' + ${user.age}"></span>
    </div>
</th:block>
```

```
</div>  
</th:block>  
  
</body>  
</html>
```

실행 결과

```
<div>  
사용자 이름1 <span>userA</span>  
사용자 나이1 <span>10</span>  
</div>  
<div>  
요약 <span>userA / 10</span>  
</div>  
  
<div>  
사용자 이름1 <span>userB</span>  
사용자 나이1 <span>20</span>  
</div>  
<div>  
요약 <span>userB / 20</span>  
</div>  
  
<div>  
사용자 이름1 <span>userC</span>  
사용자 나이1 <span>30</span>  
</div>  
<div>  
요약 <span>userC / 30</span>  
</div>
```

타임리프의 특성상 HTML 태그안에 속성으로 기능을 정의해서 사용하는데, 위 예처럼 이렇게 사용하기
애매한 경우에 사용하면 된다. `<th:block>`은 렌더링시 제거된다.

자바스크립트 인라인

타임리프는 자바스크립트에서 타임리프를 편리하게 사용할 수 있는 자바스크립트 인라인 기능을 제공한다.
자바스크립트 인라인 기능은 다음과 같이 적용하면 된다.

```
<script th:inline="javascript">
```

BasicController 추가

```
@GetMapping("/javascript")
public String javascript(Model model) {

    model.addAttribute("user", new User("userA", 10));
    addUsers(model);

    return "basic/javascript";
}
```

```
/resources/templates/basic/javascript.html
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<!-- 자바스크립트 인라인 사용 전 -->
<script>
    var username = [[${user.username}]];
    var age = [[${user.age}]];

    //자바스크립트 내추럴 템플릿
    var username2 = /*[[${user.username}]]*/ "test username";

    //객체
    var user = [[${user}]];
</script>
```

```

<!-- 자바스크립트 인라인 사용 후 -->
<script th:inline="javascript">
    var username = [[${user.username}]];
    var age = [[${user.age}]];

    //자바스크립트 내추럴 템플릿
    var username2 = /*[[${user.username}]]*/ "test username";

    //객체
    var user = [[${user}]];
</script>

</body>
</html>

```

자바스크립트 인라인 사용 전 - 결과

```

<script>
var username = userA;
var age = 10;

//자바스크립트 내추럴 템플릿
var username2 = /*userA*/ "test username";

//객체
var user = BasicController.User(username=userA, age=10);

</script>

```

자바스크립트 인라인 사용 후 - 결과

자바스크립트 인라인 사용 후

```

<script>
var username = "userA";
var age = 10;

```

```
//자바스크립트 내추럴 템플릿

var username2 = "userA";

//객체

var user = {"username":"userA", "age":10};

</script>
```

자바스크립트 인라인을 사용하지 않은 경우 어떤 문제들이 있는지 알아보고, 인라인을 사용하면 해당 문제들이 어떻게 해결되는지 확인해보자.

텍스트 렌더링

- var username = [[\${user.username}]];
 - 인라인 사용 전 → var username = userA;
 - 인라인 사용 후 → var username = "userA";
- 인라인 사용 전 렌더링 결과를 보면 userA라는 변수 이름이 그대로 남아있다. 타임리프 입장에서는 정확하게 렌더링 한 것이지만 아마 개발자가 기대한 것은 다음과 같은 "userA"라는 문자일 것이다. 결과적으로 userA가 변수명으로 사용되어서 자바스크립트 오류가 발생한다. 다음으로 나오는 숫자 age의 경우에는 " 가 필요 없기 때문에 정상 렌더링 된다.
- 인라인 사용 후 렌더링 결과를 보면 문자 타입인 경우 " 를 포함해준다. 추가로 자바스크립트에서 문제가 될 수 있는 문자가 포함되어 있으면 이스케이프 처리도 해준다. 예) " → \' "

자바스크립트 내추럴 템플릿

타임리프는 HTML 파일을 직접 열어도 동작하는 내추럴 템플릿 기능을 제공한다. 자바스크립트 인라인 기능을 사용하면 주석을 활용해서 이 기능을 사용할 수 있다.

- var username2 = /*[\${user.username}]*/ "test username";
 - 인라인 사용 전 → var username2 = /*userA*/ "test username";
 - 인라인 사용 후 → var username2 = "userA";
- 인라인 사용 전 결과를 보면 정말 순수하게 그대로 해석을 해버렸다. 따라서 내추럴 템플릿 기능이 동작하지 않고, 심지어 렌더링 내용이 주석처리 되어 버린다.
- 인라인 사용 후 결과를 보면 주석 부분이 제거되고, 기대한 "userA"가 정확하게 적용된다.

객체

타임리프의 자바스크립트 인라인 기능을 사용하면 객체를 JSON으로 자동으로 변환해준다.

- ```
var user = [[${user}]];
```

  - 인라인 사용 전 → 

```
var user = BasicController.User(username=userA, age=10);
```
  - 인라인 사용 후 → 

```
var user = {"username":"userA", "age":10};
```
- 인라인 사용 전은 객체의 `toString()`이 호출된 값이다.
- 인라인 사용 후는 객체를 JSON으로 변환해준다.

## 자바스크립트 인라인 each

자바스크립트 인라인은 `each`를 지원하는데, 다음과 같이 사용한다.

/resources/templates/basic/javascript.html에 추가

```
<!-- 자바스크립트 인라인 each -->
<script th:inline="javascript">

[# th:each="user, stat : ${users}"]
var user[[$(stat.count)]] = [[${user}]];
[/]

</script>
```

## 자바스크립트 인라인 each 결과

```
<script>
var user1 = {"username":"userA", "age":10};
var user2 = {"username":"userB", "age":20};
var user3 = {"username":"userC", "age":30};
</script>
```

## 템플릿 조각

웹 페이지를 개발할 때는 공통 영역이 많이 있다. 예를 들어서 상단 영역이나 하단 영역, 좌측 카테고리 등등 여러 페이지에서 함께 사용하는 영역들이 있다. 이런 부분을 코드를 복사해서 사용한다면 변경시 여러 페이지를 다 수정해야 하므로 상당히 비효율적이다. 타임리프는 이런 문제를 해결하기 위해 템플릿 조각과 레이아웃 기능을 지원한다.

**주의!** 템플릿 조각과 레이아웃 부분은 직접 실행해보아야 이해된다.

### 템플릿 조각

```
package hello.thymeleaf.basic;

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

@Controller
@RequestMapping("/template")
public class TemplateController {

 @GetMapping("/fragment")
 public String template() {
 return "template/fragment/fragmentMain";
 }

}
```

/resources/templates/template/fragment/footer.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<body>

<footer th:fragment="copy">
 푸터 자리입니다.

```

```

</footer>

<footer th:fragment="copyParam (param1, param2)">
 <p>파라미터 자리 입니다.</p>
 <p th:text="${param1}"></p>
 <p th:text="${param2}"></p>
</footer>

</body>

</html>

```

`th:fragment` 가 있는 태그는 다른곳에 포함되는 코드 조각으로 이해하면 된다.

/resources/templates/template/fragment/fragmentMain.html

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="UTF-8">
 <title>Title</title>
</head>
<body>
 <h1>부분 포함</h1>
 <h2>부분 포함 insert</h2>
 <div th:insert="~{template/fragment/footer :: copy}"></div>

 <h2>부분 포함 replace</h2>
 <div th:replace="~{template/fragment/footer :: copy}"></div>

 <h2>부분 포함 단순 표현식</h2>
 <div th:replace="template/fragment/footer :: copy"></div>

 <h1>파라미터 사용</h1>
 <div th:replace="~{template/fragment/footer :: copyParam ('데이터1', '데이터2')}"></div>
</body>
</html>

```

- template/fragment/footer :: copy : template/fragment/footer.html 템플릿에 있는 th:fragment="copy"라는 부분을 템플릿 조각으로 가져와서 사용한다는 의미이다.

footer.html의 copy 부분

```
<footer th:fragment="copy">
 푸터 자리입니다.
</footer>
```

### 부분 포함 insert

```
<div th:insert="~{template/fragment/footer :: copy}"></div>
```

```
<h2>부분 포함 insert</h2>
<div>
<footer>
 푸터 자리입니다.
</footer>
</div>
```

th:insert를 사용하면 현재 태그( div )내부에 추가한다.

### 부분 포함 replace

```
<div th:replace="~{template/fragment/footer :: copy}"></div>
```

```
<h2>부분 포함 replace</h2>
<footer>
 푸터 자리입니다.
</footer>
```

th:replace를 사용하면 현재 태그( div )를 대체한다.

## 부분 포함 단순 표현식

```
<div th:replace="template/fragment/footer :: copy"></div>
```

```
<h2>부분 포함 단순 표현식</h2>
<footer>
 푸터 자리입니다.
</footer>
```

~{...} 를 사용하는 것이 원칙이지만 템플릿 조각을 사용하는 코드가 단순하면 이 부분을 생략할 수 있다.

## 파라미터 사용

다음과 같이 파라미터를 전달해서 동적으로 조각을 렌더링 할 수도 있다.

```
<div th:replace="~{template/fragment/footer :: copyParam ('데이터1', '데이터2')}"></div>
```

```
<h1>파라미터 사용</h1>
<footer>
 <p>파라미터 자리입니다.</p>
 <p>데이터1</p>
 <p>데이터2</p>
</footer>
```

footer.html 의 copyParam 부분

```
<footer th:fragment="copyParam (param1, param2)">
 <p>파라미터 자리입니다.</p>
 <p th:text="${param1}"></p>
 <p th:text="${param2}"></p>
</footer>
```

## 템플릿 레이아웃1

### 템플릿 레이아웃

이전에는 일부 코드 조각을 가지고와서 사용했다면, 이번에는 개념을 더 확장해서 코드 조각을 레이아웃에 넘겨서 사용하는 방법에 대해서 알아보자.

예를 들어서 `<head>`에 공통으로 사용하는 `css`, `javascript` 같은 정보들이 있는데, 이러한 공통 정보들을 한 곳에 모아두고, 공통으로 사용하지만, 각 페이지마다 필요한 정보를 더 추가해서 사용하고 싶다면 다음과 같이 사용하면 된다.

```
@GetMapping("/layout")
public String layout() {
 return "template/layout/layoutMain";
}
```

/resources/templates/template/layout/base.html

```
<html xmlns:th="http://www.thymeleaf.org">
<head th:fragment="common_header(title,links)">

 <title th:replace="${title}">레이아웃 타이틀</title>

 <!-- 공통 -->
 <link rel="stylesheet" type="text/css" media="all" th:href="@{/css/
awesomeapp.css}">
 <link rel="shortcut icon" th:href="@{/images/favicon.ico}">
 <script type="text/javascript" th:src="@{/sh/scripts/codebase.js}"></
script>

 <!-- 추가 -->
 <th:block th:replace="${links}" />

</head>
```

```
/resources/templates/template/layout/layoutMain.html
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:replace="template/layout/base :: common_header(~{::title},~{::link})">
 <title>메인 타이틀</title>
 <link rel="stylesheet" th:href="@{/css/bootstrap.min.css}">
 <link rel="stylesheet" th:href="@{/themes/smoothness/jquery-ui.css}">
</head>
<body>
메인 컨텐츠
</body>
</html>
```

## 결과

```
<!DOCTYPE html>
<html>
<head>
<title>메인 타이틀</title>
<!-- 공통 -->

<link rel="stylesheet" type="text/css" media="all" href="/css/awesomeapp.css">
<link rel="shortcut icon" href="/images/favicon.ico">
<script type="text/javascript" src="/sh/scripts/codebase.js"></script>

<!-- 추가 -->
<link rel="stylesheet" href="/css/bootstrap.min.css">
<link rel="stylesheet" href="/themes/smoothness/jquery-ui.css">

</head>
<body>
메인 컨텐츠
</body>
</html>
```

- common\_header(~{::title},~{::link}) 이 부분이 핵심이다.
  - ::title 은 현재 페이지의 title 태그들을 전달한다.
  - ::link 는 현재 페이지의 link 태그들을 전달한다.

### 결과를 보자.

- 메인 타이틀이 전달한 부분으로 교체되었다.
- 공통 부분은 그대로 유지되고, 추가 부분에 전달한 <link> 들이 포함된 것을 확인할 수 있다.

이 방식은 사실 앞서 배운 코드 조각을 조금 더 적극적으로 사용하는 방식이다. 쉽게 이야기해서 레이아웃 개념을 두고, 그 레이아웃에 필요한 코드 조각을 전달해서 완성하는 것으로 이해하면 된다.

## 템플릿 레이아웃2

### 템플릿 레이아웃 확장

앞서 이야기한 개념을 <head> 정도에만 적용하는게 아니라 <html> 전체에 적용할 수도 있다.

```
@GetMapping("/layoutExtend")
public String layoutExtends() {
 return "template/layoutExtend/layoutExtendMain";
}
```

/resources/templates/template/layoutExtend/layoutFile.html

```
<!DOCTYPE html>
<html th:fragment="layout (title, content)" xmlns:th="http://
www.thymeleaf.org">
<head>
 <title th:replace="${title}">레이아웃 타이틀</title>
</head>
<body>
 <h1>레이아웃 H1</h1>
 <div th:replace="${content}">
 <p>레이아웃 컨텐츠</p>
 </div>
```

```
<footer>
 레이아웃 푸터
</footer>
</body>
</html>
```

/resources/templates/template/layoutExtend/layoutExtendMain.html

```
<!DOCTYPE html>
<html th:replace="~{template/layoutExtend/layoutFile :: layout(~{::title},
~{::section})}" xmlns:th="http://www.thymeleaf.org">
<head>
 <title>메인 페이지 타이틀</title>
</head>
<body>
<section>
 <p>메인 페이지 컨텐츠</p>
 <div>메인 페이지 포함 내용</div>
</section>
</body>
</html>
```

## 생성 결과

```
<!DOCTYPE html>
<html>
<head>
 <title>메인 페이지 타이틀</title>
</head>
<body>
 <h1>레이아웃 H1</h1>

 <section>
 <p>메인 페이지 컨텐츠</p>
 <div>메인 페이지 포함 내용</div>
 </section>
```

```
<footer>
```

레이아웃 푸터

```
</footer>
```

```
</body>
```

```
</html>
```

`layoutFile.html`을 보면 기본 레이아웃을 가지고 있는데, `<html>`에 `th:fragment` 속성이 정의되어 있다. 이 레이아웃 파일을 기본으로 하고 여기에 필요한 내용을 전달해서 부분부분 변경하는 것으로 이해하면 된다.

`layoutExtendMain.html`는 현재 페이지인데, `<html>` 자체를 `th:replace`를 사용해서 변경하는 것을 확인할 수 있다. 결국 `layoutFile.html`에 필요한 내용을 전달하면서 `<html>` 자체를 `layoutFile.html`로 변경한다.

## 정리

## 2. 타임리프 - 스프링 통합과 폼

#인강/5. 스프링 MVC 2/강의#

### 목차

- 2. 타임리프 - 스프링 통합과 폼 - 프로젝트 설정
- 2. 타임리프 - 스프링 통합과 폼 - 타임리프 스프링 통합
- 2. 타임리프 - 스프링 통합과 폼 - 입력 폼 처리
- 2. 타임리프 - 스프링 통합과 폼 - 요구사항 추가
- 2. 타임리프 - 스프링 통합과 폼 - 체크 박스 - 단일1
- 2. 타임리프 - 스프링 통합과 폼 - 체크 박스 - 단일2
- 2. 타임리프 - 스프링 통합과 폼 - 체크 박스 - 멀티
- 2. 타임리프 - 스프링 통합과 폼 - 라디오 버튼
- 2. 타임리프 - 스프링 통합과 폼 - 셀렉트 박스
- 2. 타임리프 - 스프링 통합과 폼 - 정리

### 프로젝트 설정

스프링 MVC 1편에서 마지막에 완성했던 상품 관리 프로젝트를 떠올려보자.

지금부터 이 프로젝트에 스프링이 지원하는 다양한 기능을 붙여가면서 스프링 MVC를 깊이있게 학습해보자.

MVC1 편에서 개발한 상품 관리 프로젝트를 약간 다듬어서 `form-start`라는 프로젝트에 넣어두었다.

#### 프로젝트 설정 순서

1. `form-start`의 폴더 이름을 `form`로 변경하자.
2. **프로젝트 임포트**  
File → Open → 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.
3. `ItemServiceApplication.main()`을 실행해서 프로젝트가 정상 수행되는지 확인하자.

#### 실행

- <http://localhost:8080>
- <http://localhost:8080/form/items>

## 타임리프 스프링 통합

타임리프는 크게 2가지 메뉴얼을 제공한다.

- 기본 메뉴얼: <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html>
- 스프링 통합 메뉴얼: <https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html>

타임리프는 스프링 없이도 동작하지만, 스프링과 통합을 위한 다양한 기능을 편리하게 제공한다. 그리고 이런 부분은 스프링으로 백엔드를 개발하는 개발자 입장에서 타임리프를 선택하는 하나의 이유가 된다.

### 스프링 통합으로 추가되는 기능들

- 스프링의 SpringEL 문법 통합
- \${@myBean.doSomething()} 처럼 스프링 빈 호출 지원
- 편리한 폼 관리를 위한 추가 속성
  - th:object (기능 강화, 폼 커맨드 객체 선택)
  - th:field, th:errors, th:errorclass
- 폼 컴포넌트 기능
  - checkbox, radio button, List 등을 편리하게 사용할 수 있는 기능 지원
- 스프링의 메시지, 국제화 기능의 편리한 통합
- 스프링의 검증, 오류 처리 통합
- 스프링의 변환 서비스 통합(ConversionService)

### 설정 방법

타임리프 템플릿 엔진을 스프링 빈에 등록하고, 타임리프용 뷰 리졸버를 스프링 빈으로 등록하는 방법

- <https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html#the-springstandard-dialect>
- <https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html#views-and-view-resolvers>

스프링 부트는 이런 부분을 모두 자동화 해준다. build.gradle에 다음 한줄을 넣어주면 Gradle은 타임리프와 관련된 라이브러리를 다운로드 받고, 스프링 부트는 앞서 설명한 타임리프와 관련된 설정용 스프링 빈을 자동으로 등록해준다.

### build.gradle

```
implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
```

타임리프 관련 설정을 변경하고 싶으면 다음을 참고해서 `application.properties`에 추가하면 된다.

### 스프링 부트가 제공하는 타임리프 설정, thymeleaf 검색 필요

<https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#common-application-properties-templating>

## 입력 폼 처리

지금부터 타임리프가 제공하는 입력 폼 기능을 적용해서 기존 프로젝트의 폼 코드를 타임리프가 지원하는 기능을 사용해서 효율적으로 개선해보자.

- `th:object` : 커맨드 객체를 지정한다.
- `*{...}` : 선택 변수 식이라고 한다. `th:object`에서 선택한 객체에 접근한다.
- `th:field`
  - HTML 태그의 `id`, `name`, `value` 속성을 자동으로 처리해준다.

### 렌더링 전

```
<input type="text" th:field="*{itemName}" />
```

### 렌더링 후

```
<input type="text" id="itemName" name="itemName" th:value="*{itemName}" />
```

## 등록 폼

`th:object`를 적용하려면 먼저 해당 오브젝트 정보를 넘겨주어야 한다. 등록 폼이기 때문에 데이터가 비어있는 빈 오브젝트를 만들어서 뷰에 전달하자.

### FormItemController 변경

```
@GetMapping("/add")
public String addForm(Model model) {
 model.addAttribute("item", new Item());
```

```
 return "form/addForm";
 }
```

이제 본격적으로 타임리프 등록 폼을 변경하자.

#### form/addForm.html 변경 코드 부분

```
<form action="item.html" th:action th:object="${item}" method="post">
 <div>
 <label for="itemName">상품명</label>
 <input type="text" id="itemName" th:field="*{itemName}" class="form-control" placeholder="이름을 입력하세요">
 </div>
 <div>
 <label for="price">가격</label>
 <input type="text" id="price" th:field="*{price}" class="form-control" placeholder="가격을 입력하세요">
 </div>
 <div>
 <label for="quantity">수량</label>
 <input type="text" id="quantity" th:field="*{quantity}" class="form-control" placeholder="수량을 입력하세요">
 </div>
```

- `th:object="${item}"` : `<form>`에서 사용할 객체를 지정한다. 선택 변수 식(`*{...}`)을 적용할 수 있다.
- `th:field="*{itemName}"`
  - `*{itemName}`는 선택 변수 식을 사용했는데,  `${item.itemName}` 과 같다. 앞서 `th:object`로 `item`을 선택했기 때문에 선택 변수 식을 적용할 수 있다.
  - `th:field`는 `id`, `name`, `value` 속성을 모두 자동으로 만들어준다.
    - `id` : `th:field`에서 지정한 변수 이름과 같다. `id="itemName"`
    - `name` : `th:field`에서 지정한 변수 이름과 같다. `name="itemName"`
    - `value` : `th:field`에서 지정한 변수의 값을 사용한다. `value=""`

참고로 해당 예제에서 `id` 속성을 제거해도 `th:field` 가 자동으로 만들어준다.

렌더링 전

```
<input type="text" id="itemName" th:field="*{itemName}" class="form-control"
placeholder="이름을 입력하세요">
```

렌더링 후

```
<input type="text" id="itemName" class="form-control" placeholder="이름을
입력하세요" name="itemName" value="">
```

수정 품

### FormItemController 유지

```
@GetMapping("/{itemId}/edit")
public String editForm(@PathVariable Long itemId, Model model) {
 Item item = itemRepository.findById(itemId);
 model.addAttribute("item", item);
 return "form/editForm";
}
```

form/editForm.html 변경 코드 부분

```
<form action="item.html" th:action th:object="${item}" method="post">
<div>
 <label for="id">상품 ID</label>
 <input type="text" id="id" th:field="*{id}" class="form-control"
readonly>
</div>
<div>
 <label for="itemName">상품명</label>
 <input type="text" id="itemName" th:field="*{itemName}" class="form-
control">
</div>
<div>
```

```
<label for="price">가격</label>
<input type="text" id="price" th:field="*{price}" class="form-control">
</div>

<div>
 <label for="quantity">수량</label>
 <input type="text" id="quantity" th:field="*{quantity}" class="form-
control">
</div>
```

수정 폼은 앞서 설명한 내용과 같다. 수정 폼의 경우 `id`, `name`, `value`를 모두 신경써야 했는데, 많은 부분이 `th:field` 덕분에 자동으로 처리되는 것을 확인할 수 있다.

### 렌더링 전

```
<input type="text" id="itemName" th:field="*{itemName}" class="form-control">
```

### 렌더링 후

```
<input type="text" id="itemName" class="form-control" name="itemName"
value="itemA">
```

### 정리

`th:object`, `th:field` 덕분에 폼을 개발할 때 약간의 편리함을 얻었다.

쉽고 단순해서 크게 어려움이 없었을 것이다.

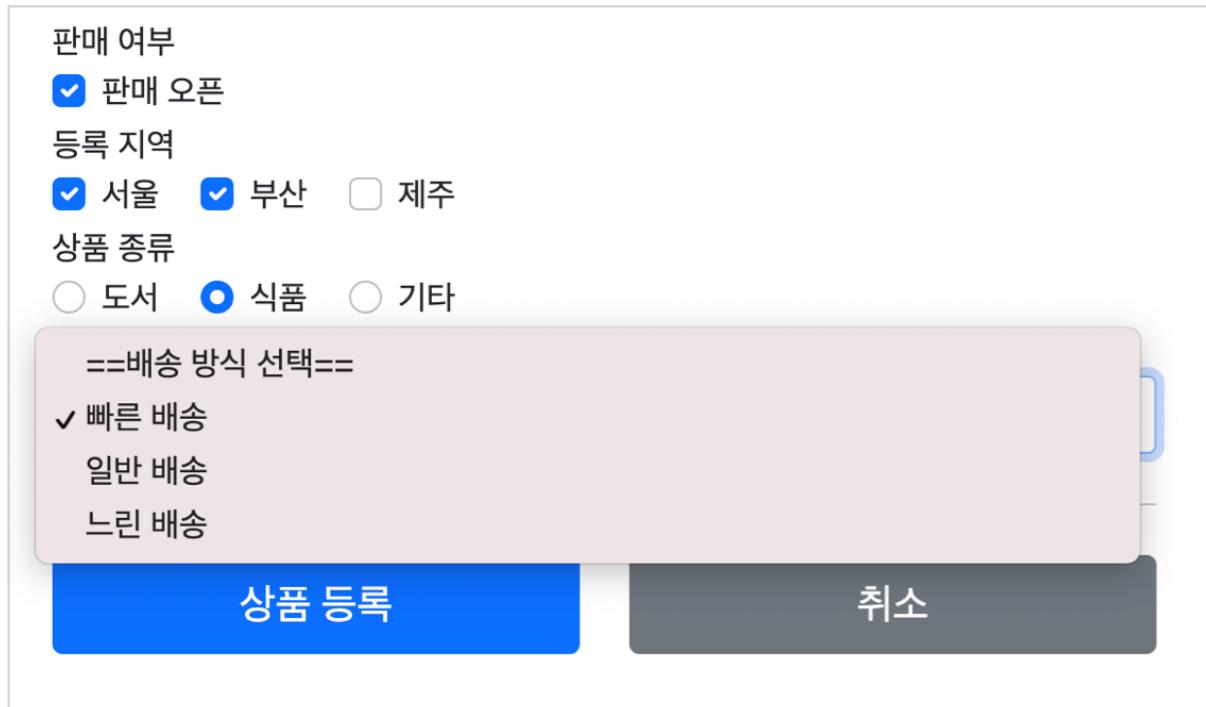
사실 이것의 진짜 위력은 뒤에 설명할 검증(Validation)에서 나타난다. 이후 검증 부분에서 폼 처리와 관련된 부분을 더 깊이있게 알아보자.

### 요구사항 추가

타임리프를 사용해서 폼에서 체크박스, 라디오 버튼, 셀렉트 박스를 편리하게 사용하는 방법을 학습해보자. 기존 상품 서비스에 다음 요구사항이 추가되었다.

- 판매 여부
  - 판매 오픈 여부
  - 체크 박스로 선택할 수 있다.
- 등록 지역
  - 서울, 부산, 제주
  - 체크 박스로 다중 선택할 수 있다.
- 상품 종류
  - 도서, 식품, 기타
  - 라디오 버튼으로 하나만 선택할 수 있다.
- 배송 방식
  - 빠른 배송
  - 일반 배송
  - 느린 배송
  - 셀렉트 박스로 하나만 선택할 수 있다.

### 예시 이미지



### ItemType - 상품 종류

```
package hello.itemservice.domain.item;

public enum ItemType {
```

```

BOOK("도서"), FOOD("식품"), ETC("기타");

private final String description;

ItemType(String description) {
 this.description = description;
}

public String getDescription() {
 return description;
}

```

상품 종류는 `ENUM`을 사용한다. 설명을 위해 `description` 필드를 추가했다.

### 배송 방식 - **DeliveryCode**

```

package hello.itemservice.domain.item;

import lombok.AllArgsConstructor;
import lombok.Data;

/**
 * FAST: 빠른 배송
 * NORMAL: 일반 배송
 * SLOW: 느린 배송
 */
@Data
@AllArgsConstructor
public class DeliveryCode {
 private String code;
 private String displayName;
}

```

배송 방식은 `DeliveryCode`라는 클래스를 사용한다. `code`는 `FAST` 같은 시스템에서 전달하는 값이고, `displayName`은 `빠른 배송` 같은 고객에게 보여주는 값이다.

## Item - 상품

```
package hello.itemservice.domain.item;

import lombok.Data;

import java.util.List;

@Data
public class Item {

 private Long id;
 private String itemName;
 private Integer price;
 private Integer quantity;

 private Boolean open; //판매 여부
 private List<String> regions; //등록 지역
 private ItemType itemType; //상품 종류
 private String deliveryCode; //배송 방식

 public Item() {
 }

 public Item(String itemName, Integer price, Integer quantity) {
 this.itemName = itemName;
 this.price = price;
 this.quantity = quantity;
 }
}
```

ENUM, 클래스, String 같은 다양한 상황을 준비했다. 각각의 상황에 어떻게 품의 데이터를 받을 수 있는지 하나씩 알아보자.

## 체크 박스 - 단일1

### 단순 HTML 체크 박스

resources/templates/form/addForm.html 추가

```
<hr class="my-4">

<!-- single checkbox -->
<div>판매 여부</div>

<div>
 <div class="form-check">
 <input type="checkbox" id="open" name="open" class="form-check-input">
 <label for="open" class="form-check-label">판매 오픈</label>
 </div>
</div>
```

상품이 등록되는 곳에 다음과 같이 로그를 남겨서 값이 잘 넘어오는지 확인해보자.

### FormItemController 추가

```
@PostMapping("/add")
public String addItem(Item item, RedirectAttributes redirectAttributes) {
 log.info("item.open={}, item.getOpen()", item.getOpen());
 ...
}
```

FormItemController에 @Slf4j 애노테이션 추가

### 실행 로그

```
FormItemController : item.open=true //체크 박스를 선택하는 경우
FormItemController : item.open=null //체크 박스를 선택하지 않는 경우
```

체크 박스를 체크하면 HTML Form에서 `open=on`이라는 값이 넘어간다. 스프링은 `on`이라는 문자를 `true` 타입으로 변환해준다. (스프링 타입 컨버터가 이 기능을 수행하는데, 뒤에서 설명한다.)

### 주의 - 체크 박스를 선택하지 않을 때

HTML에서 체크 박스를 선택하지 않고 폼을 전송하면 `open`이라는 필드 자체가 서버로 전송되지 않는다.

## HTTP 요청 메시지 로깅

HTTP 요청 메시지를 서버에서 보고 싶으면 다음 설정을 추가하면 된다.

`application.properties`

```
logging.level.org.apache.coyote.http11=debug
```

HTTP 메시지 바디를 보면 `open`의 이름도 전송이 되지 않는 것을 확인할 수 있다.

```
itemName=itemA&price=10000&quantity=10
```

서버에서 Boolean 타입을 찍어보면 결과가 `null`인 것을 확인할 수 있다.

```
log.info("item.open={}, item.getOpen()");
```

HTML checkbox는 선택이 안되면 클라이언트에서 서버로 값 자체를 보내지 않는다. 수정의 경우에는 상황에 따라서 이 방식이 문제가 될 수 있다. 사용자가 의도적으로 체크되어 있던 값을 체크를 해제해도 저장시 아무 값도 넘어가지 않기 때문에, 서버 구현에 따라서 값이 오지 않은 것으로 판단해서 값을 변경하지 않을 수도 있다.

이런 문제를 해결하기 위해서 스프링 MVC는 약간의 트릭을 사용하는데, 히든 필드를 하나 만들어서, `_open`처럼 기존 체크 박스 이름 앞에 언더스코어(`_`)를 붙여서 전송하면 체크를 해제했다고 인식할 수 있다. 히든 필드는 항상 전송된다. 따라서 체크를 해제한 경우 여기에서 `open`은 전송되지 않고, `_open`만 전송되는데, 이 경우 스프링 MVC는 체크를 해제했다고 판단한다.

## 체크 해제를 인식하기 위한 히든 필드

```
<input type="hidden" name="_open" value="on"/>
```

## 기존 코드에 히든 필드 추가

```
<!-- single checkbox -->
<div>판매 여부</div>

<div>
 <div class="form-check">
 <input type="checkbox" id="open" name="open" class="form-check-input">
 <input type="hidden" name="_open" value="on"/> <!-- 히든 필드 추가 -->
 <label for="open" class="form-check-label">판매 오픈</label>
 </div>
</div>
```

```
</div>
</div>
```

## 실행 로그

```
FormItemController : item.open=true //체크 박스를 선택하는 경우
FormItemController : item.open=false //체크 박스를 선택하지 않는 경우
```

### 체크 박스 체크

```
open=on&_open=on
```

체크 박스를 체크하면 스프링 MVC가 `open`에 값이 있는 것을 확인하고 사용한다. 이때 `_open`은 무시한다.

### 체크 박스 미체크

```
_open=on
```

체크 박스를 체크하지 않으면 스프링 MVC가 `_open`만 있는 것을 확인하고, `open`의 값이 체크되지 않았다고 인식한다.

이 경우 서버에서 `Boolean` 타입을 찍어보면 결과가 `null`이 아니라 `false`인 것을 확인할 수 있다.

```
log.info("item.open={}, item.getOpen());
```

## 체크 박스 - 단일2

### 타임리프

개발할 때마다 이렇게 히든 필드를 추가하는 것은 상당히 번거롭다. 타임리프가 제공하는 품 기능을 사용하면 이런 부분을 자동으로 처리할 수 있다.

### 타임리프 - 체크 박스 코드 추가

```
<!-- single checkbox -->
<div>판매 여부</div>
<div>
```

```

<div class="form-check">
 <input type="checkbox" id="open" th:field="*{open}" class="form-check-input">
 <label for="open" class="form-check-label">판매 오픈</label>
 </div>
</div>

```

체크 박스의 기존 코드를 제거하고 타임리프가 제공하는 체크 박스 코드로 변경하자.

## 타임리프 체크 박스 HTML 생성 결과

```

<!-- single checkbox -->
<div>판매 여부</div>

<div>
 <div class="form-check">
 <input type="checkbox" id="open" class="form-check-input" name="open"
value="true">
 <input type="hidden" name="_open" value="on"/>
 <label for="open" class="form-check-label">판매 오픈</label>
 </div>
</div>

```

- <input type="hidden" name="\_open" value="on"/>

타임리프를 사용하면 체크 박스의 히든 필드와 관련된 부분도 함께 해결해준다. HTML 생성 결과를 보면 히든 필드 부분이 자동으로 생성되어 있다.

## 실행 로그

FormItemController	: item.open=true //체크 박스를 선택하는 경우
FormItemController	: item.open=false //체크 박스를 선택하지 않는 경우

상품 상세에 적용하자.

## item.html

```

<hr class="my-4">

```

```

<!-- single checkbox -->
<div>판매 여부</div>

<div>
 <div class="form-check">
 <input type="checkbox" id="open" th:field="${item.open}" class="form-
check-input" disabled>
 <label for="open" class="form-check-label">판매 오픈</label>
 </div>
</div>

```

**주의:** item.html에는 th:object 를 사용하지 않았기 때문에 th:field 부분에 \${item.open} 으로 적어주어야 한다.

disabled 를 사용해서 상품 상세에서는 체크 박스가 선택되지 않도록 했다.

## HTML 생성 결과

```

<hr class="my-4">

<!-- single checkbox -->
<div class="form-check">
 <input type="checkbox" id="open" class="form-check-input" disabled
name="open" value="true"
 checked="checked">
 <label for="open" class="form-check-label">판매 오픈</label>
</div>

```

## 타임리프의 체크 확인

checked="checked"

체크 박스에서 판매 여부를 선택해서 저장하면, 조회시에 checked 속성이 추가된 것을 확인할 수 있다. 이런 부분을 개발자가 직접 처리하려면 상당히 번거롭다. 타임리프의 th:field 를 사용하면, 값이 true 인 경우 체크를 자동으로 처리해준다.

상품 수정에도 적용하자.

## editForm.html

```

<hr class="my-4">

<!-- single checkbox -->
<div>판매 여부</div>
<div>
 <div class="form-check">
 <input type="checkbox" id="open" th:field="*{open}" class="form-check-input">
 <label for="open" class="form-check-label">판매 오픈</label>
 </div>
</div>
...

```

상품 수정도 `th:object`, `th:field`를 모두 적용해야 한다.

실행해보면 체크 박스를 수정해도 반영되지 않는다. 실제 반영되도록 다음 코드를 수정하자.

### ItemRepository - update() 코드를 다음과 같이 수정하자

```

public void update(Long itemId, Item updateParam) {
 Item findItem = findById(itemId);
 findItem.setItemName(updateParam.getItemName());
 findItem.setPrice(updateParam.getPrice());
 findItem.setQuantity(updateParam.getQuantity());
 findItem.setOpen(updateParam.getOpen());
 findItem.setRegions(updateParam.getRegions());
 findItem.setItemType(updateParam.getItemType());
 findItem.setDeliveryCode(updateParam.getDeliveryCode());
}

```

`open` 이외에 나머지 필드도 업데이트 되도록 미리 넣어두자.

## 체크 박스 - 멀티

체크 박스를 멀티로 사용해서, 하나 이상을 체크할 수 있도록 해보자.

- 등록 지역
  - 서울, 부산, 제주
  - 체크 박스로 다중 선택할 수 있다.

## FormItemController - 추가

```
@ModelAttribute("regions")
public Map<String, String> regions() {
 Map<String, String> regions = new LinkedHashMap<>();
 regions.put("SEOUL", "서울");
 regions.put("BUSAN", "부산");
 regions.put("JEJU", "제주");
 return regions;
}
```

## @ModelAttribute의 특별한 사용법

등록 폼, 상세화면, 수정 폼에서 모두 서울, 부산, 제주라는 체크 박스를 반복해서 보여주어야 한다. 이렇게 하려면 각각의 컨트롤러에서 `model.addAttribute(...)`을 사용해서 체크 박스를 구성하는 데이터를 반복해서 넣어주어야 한다.

`@ModelAttribute`는 이렇게 컨트롤러에 있는 별도의 메서드에 적용할 수 있다.

이렇게하면 해당 컨트롤러를 요청할 때 `regions`에서 반환한 값이 자동으로 모델(`model`)에 담기게 된다. 물론 이렇게 사용하지 않고, 각각의 컨트롤러 메서드에서 모델에 직접 데이터를 담아서 처리해도 된다.

## addForm.html - 추가

```
<!-- multi checkbox -->
<div>
 <div>등록 지역</div>
 <div th:each="region : ${regions}" class="form-check form-check-inline">
 <input type="checkbox" th:field="*{regions}" th:value="${region.key}" class="form-check-input">
 <label th:for="${#ids.prev('regions')}" th:text="${region.value}" class="form-check-label">서울</label>
```

```
</div>
</div>
```

- `th:for="#ids.prev('regions'))"`

멀티 체크박스는 같은 이름의 여러 체크박스를 만들 수 있다. 그런데 문제는 이렇게 반복해서 HTML 태그를 생성할 때, 생성된 HTML 태그 속성에서 `name`은 같아도 되지만, `id`는 모두 달라야 한다. 따라서 타임리프는 체크박스를 `each` 루프 안에서 반복해서 만들 때 임의로 1, 2, 3 숫자를 뒤에 붙여준다.

### each로 체크박스가 반복 생성된 결과 - id 뒤에 숫자가 추가

```
<input type="checkbox" value="SEOUL" class="form-check-input" id="regions1"
name="regions">
<input type="checkbox" value="BUSAN" class="form-check-input" id="regions2"
name="regions">
<input type="checkbox" value="JEJU" class="form-check-input" id="regions3"
name="regions">
```

HTML의 `id` 가 타임리프에 의해 동적으로 만들어지기 때문에 `<label for="id 값">` 으로 `label` 의 대상이 되는 `id` 값을 임의로 지정하는 것은 곤란하다. 타임리프는 `ids.prev(...)`, `ids.next(...)` 을 제공해서 동적으로 생성되는 `id` 값을 사용할 수 있도록 한다.

### 타임리프 HTML 생성 결과

```
<!-- multi checkbox -->
<div>
 <div>등록 지역</div>
 <div class="form-check form-check-inline">
 <input type="checkbox" value="SEOUL" class="form-check-input"
 id="regions1" name="regions">
 <input type="hidden" name="_regions" value="on"/>
 <label for="regions1"
 class="form-check-label">서울</label>
 </div>
 <div class="form-check form-check-inline">
 <input type="checkbox" value="BUSAN" class="form-check-input"
 id="regions2" name="regions">
```

```

<input type="hidden" name="_regions" value="on"/>
<label for="regions2"
 class="form-check-label">부산</label>
</div>
<div class="form-check form-check-inline">
 <input type="checkbox" value="JEJU" class="form-check-input"
 id="regions3" name="regions">
 <input type="hidden" name="_regions" value="on"/>
 <label for="regions3"
 class="form-check-label">제주</label>
</div>
</div>
<!-- -->

```

<label for="id 값">에 지정된 `id` 가 `checkbox`에서 동적으로 생성된 `regions1`, `regions2`, `regions3`에 맞추어 순서대로 입력된 것을 확인할 수 있다.

## 로그 출력

`FormItemController.addItem()`에 코드 추가

```
log.info("item.regions={}, item.getRegions());
```

## 서울, 부산 선택

```
regions=SEOUL&_regions=on®ions=BUSAN&_regions=on&_regions=on
```

로그: `item.regions=[SEOUL, BUSAN]`

## 지역 선택X

```
_regions=on&_regions=on&_regions=on
```

로그: `item.regions=[]`

`_regions` 는 앞서 설명한 기능이다. 웹 브라우저에서 체크를 하나도 하지 않았을 때, 클라이언트가 서버에 아무런 데이터를 보내지 않는 것을 방지한다. 참고로 `_regions` 조차 보내지 않으면 결과는 `null` 이 된다. `_regions` 가 체크박스 숫자만큼 생성될 필요는 없지만, 타임리프가 생성되는 옵션 수 만큼 생성해서 그런 것이니 무시하자.

## item.html - 추가

```
<!-- multi checkbox -->

<div>
 <div>등록 지역</div>
 <div th:each="region : ${regions}" class="form-check form-check-inline">
 <input type="checkbox" th:field="${item.regions}" th:value="$
{region.key}" class="form-check-input" disabled>
 <label th:for="#ids.prev('regions')}"
 th:text="${region.value}" class="form-check-label">서울</label>
 </div>
</div>
```

**주의:** `item.html` 에는 `th:object` 를 사용하지 않았기 때문에 `th:field` 부분에 `${item.regions}` 으로 적어주어야 한다.

`disabled` 를 사용해서 상품 상세에서는 체크 박스가 선택되지 않도록 했다.

## 타임리프의 체크 확인

`checked="checked"`

멀티 체크 박스에서 등록 지역을 선택해서 저장하면, 조회시에 `checked` 속성이 추가된 것을 확인할 수 있다.

타임리프는 `th:field` 에 지정한 값과 `th:value` 의 값을 비교해서 체크를 자동으로 처리해준다.

## editForm.html - 추가

```
<!-- multi checkbox -->

<div>
 <div>등록 지역</div>
 <div th:each="region : ${regions}" class="form-check form-check-inline">
 <input type="checkbox" th:field="${item.regions}" th:value="$
{region.key}" class="form-check-input">
```

```

<label th:for="${#ids.prev('regions')}"
 th:text="${region.value}" class="form-check-label">서울</label>
</div>
</div>

```

## 라디오 버튼

라디오 버튼은 여러 선택지 중에 하나를 선택할 때 사용할 수 있다. 이번시간에는 라디오 버튼을 자바 ENUM을 활용해서 개발해보자.

- 상품 종류
  - 도서, 식품, 기타
  - 라디오 버튼으로 하나만 선택할 수 있다.

## FormItemController - 추가

```

@ModelAttribute("itemTypes")
public ItemType[] itemTypes() {
 return ItemType.values();
}

```

`itemTypes` 를 등록 폼, 조회, 수정 폼에서 모두 사용하므로 `@ModelAttribute` 의 특별한 사용법을 적용하자.

`ItemType.values()` 를 사용하면 해당 ENUM의 모든 정보를 배열로 반환한다. 예) [BOOK, FOOD, ETC]

상품 등록 폼에 기능을 추가해보자.

## addForm.html - 추가

```

<!-- radio button -->
<div>
 <div>상품 종류</div>
 <div th:each="type : ${itemTypes}" class="form-check form-check-inline">
 <input type="radio" th:field="*{itemType}" th:value="${type.name()}">

```

```

class="form-check-input">
 <label th:for="${#ids.prev('itemType')}" th:text="${type.description}"
class="form-check-label">
 BOOK
</label>
</div>
</div>

```

## 실행 결과, 품 전송

itemType=FOOD //음식 선택, 선택하지 않으면 아무 값도 넘어가지 않는다.

## 로그 추가

```
log.info("item.itemType={}, item.getItemType());
```

## 실행 로그

item.itemType=FOOD: 값이 있을 때  
item.itemType=null: 값이 없을 때

체크 박스는 수정시 체크를 해제하면 아무 값도 넘어가지 않기 때문에, 별도의 히든 필드로 이런 문제를 해결했다. 라디오 버튼은 이미 선택이 되어 있다면, 수정시에도 항상 하나를 선택하도록 되어 있으므로 체크 박스와 달리 별도의 히든 필드를 사용할 필요가 없다.

상품 상세와 수정에도 라디오 버튼을 넣어주자.

## item.html

```

<!-- radio button -->
<div>
 <div>상품 종류</div>
 <div th:each="type : ${itemTypes}" class="form-check form-check-inline">
 <input type="radio" th:field="${item.itemType}" th:value="${type.name()}" class="form-check-input" disabled>

```

```

<label th:for="${#ids.prev('itemType')}" th:text="${type.description}"
class="form-check-label">
 BOOK
</label>
</div>
</div>

```

**주의:** item.html에는 th:object 를 사용하지 않았기 때문에 th:field 부분에 \${item.itemType} 으로 적어주어야 한다.

disabled 를 사용해서 상품 상세에서는 라디오 버튼이 선택되지 않도록 했다.

## editForm.html

```

<!-- radio button -->
<div>
 <div>상품 종류</div>
 <div th:each="type : ${itemTypes}" class="form-check form-check-inline">
 <input type="radio" th:field="*{itemType}" th:value="${type.name()}"
class="form-check-input">
 <label th:for="${#ids.prev('itemType')}" th:text="${type.description}"
class="form-check-label">
 BOOK
 </label>
 </div>
 </div>

```

## 타임리프로 생성된 HTML

```

<!-- radio button -->
<div>
 <div>상품 종류</div>
 <div class="form-check form-check-inline">
 <input type="radio" value="BOOK" class="form-check-input"
id="itemType1" name="itemType">
 <label for="itemType1" class="form-check-label">도서</label>
 </div>

```

```

<div class="form-check form-check-inline">
 <input type="radio" value="FOOD" class="form-check-input" id="itemType2" name="itemType" checked="checked">
 <label for="itemType2" class="form-check-label">식품</label>
 </div>
 <div class="form-check form-check-inline">
 <input type="radio" value="ETC" class="form-check-input" id="itemType3" name="itemType">
 <label for="itemType3" class="form-check-label">기타</label>
 </div>
</div>

```

선택한 식품(FOOD)에 checked="checked" 가 적용된 것을 확인할 수 있다.

## 타임리프에서 ENUM 직접 사용하기

이렇게 모델에 ENUM을 담아서 전달하는 대신에 타임리프는 자바 객체에 직접 접근할 수 있다.

```

@ModelAttribute("itemTypes")
public ItemType[] itemTypes() {
 return ItemType.values();
}

```

## 타임리프에서 ENUM 직접 접근

```
<div th:each="type : ${T(hello.itemservice.domain.item.ItemType).values()}">
```

`${T(hello.itemservice.domain.item.ItemType).values() }` 스프링EL 문법으로 ENUM을 직접 사용할 수 있다. ENUM에 `values()` 를 호출하면 해당 ENUM의 모든 정보가 배열로 반환된다.

그런데 이렇게 사용하면 ENUM의 패키지 위치가 변경되거나 할때 자바 컴파일러가 타임리프까지 컴파일 오류를 잡을 수 없으므로 추천하지는 않는다.

## 셀렉트 박스

셀렉트 박스는 여러 선택지 중에 하나를 선택할 때 사용할 수 있다. 이번시간에는 셀렉트 박스를 자바 객체를 활용해서 개발해보자.

- 배송 방식
  - 빠른 배송
  - 일반 배송
  - 느린 배송
  - 셀렉트 박스로 하나만 선택할 수 있다.

### FormItemController - 추가

```
@ModelAttribute("deliveryCodes")
public List<DeliveryCode> deliveryCodes() {
 List<DeliveryCode> deliveryCodes = new ArrayList<>();
 deliveryCodes.add(new DeliveryCode("FAST", "빠른 배송"));
 deliveryCodes.add(new DeliveryCode("NORMAL", "일반 배송"));
 deliveryCodes.add(new DeliveryCode("SLOW", "느린 배송"));
 return deliveryCodes;
}
```

DeliveryCode라는 자바 객체를 사용하는 방법으로 진행하겠다.

DeliveryCode를 등록 폼, 조회, 수정 폼에서 모두 사용하므로 @ModelAttribute의 특별한 사용법을 적용하자.

참고: @ModelAttribute 가 있는 deliveryCodes() 메서드는 컨트롤러가 호출 될 때 마다 사용되므로 deliveryCodes 객체도 계속 생성된다. 이런 부분은 미리 생성해두고 재사용하는 것이 더 효율적이다.

### addForm.html - 추가

```
<!-- SELECT -->
<div>
 <div>배송 방식</div>
 <select th:field="*{deliveryCode}" class="form-select">
 <option value="">==배송 방식 선택==</option>
 <option th:each="deliveryCode : ${deliveryCodes}" th:value="$
```

```

{deliveryCode.code}">
 th:text="${deliveryCode.displayName}">FAST</option>
</select>
</div>

<hr class="my-4">

```

## 타임리프로 생성된 HTML

```

<!-- SELECT -->
<div>
 <DIV>배송 방식</DIV>
 <select class="form-select" id="deliveryCode" name="deliveryCode">
 <option value="">==배송 방식 선택==</option>
 <option value="FAST">빠른 배송</option>
 <option value="NORMAL">일반 배송</option>
 <option value="SLOW">느린 배송</option>
 </select>
</div>

```

상품 상세와 수정에도 셀렉트 박스를 넣어주자.

## item.html

```

<!-- SELECT -->
<div>
 <div>배송 방식</div>
 <select th:field="${item.deliveryCode}" class="form-select" disabled>
 <option value="">==배송 방식 선택==</option>
 <option th:each="deliveryCode : ${deliveryCodes}" th:value="${deliveryCode.code}">
 th:text="${deliveryCode.displayName}">FAST</option>
 </select>
</div>

<hr class="my-4">

```

**주의:** item.html 에는 th:object 를 사용하지 않았기 때문에 th:field 부분에 \$ \${item.deliveryCode} 으로 적어주어야 한다.  
disabled 를 사용해서 상품 상세에서는 셀렉트 박스가 선택되지 않도록 했다.

## editForm.html

```
<!-- SELECT -->

<div>
 <div>배송 방식</div>
 <select th:field="*{deliveryCode}" class="form-select">
 <option value="">==배송 방식 선택==</option>
 <option th:each="deliveryCode : ${deliveryCodes}" th:value="$
{deliveryCode.code}"
 th:text="${deliveryCode.displayName}">FAST</option>
 </select>
</div>

<hr class="my-4">
```

## 타임리프로 생성된 HTML

```
<!-- SELECT -->

<div>
 <DIV>배송 방식</DIV>
 <select class="form-select" id="deliveryCode" name="deliveryCode">
 <option value="">==배송 방식 선택==</option>
 <option value="FAST" selected="selected">빠른 배송</option>
 <option value="NORMAL">일반 배송</option>
 <option value="SLOW">느린 배송</option>
 </select>
</div>
```

**selected="selected"**  
빠른 배송을 선택한 예시인데, 선택된 셀렉트 박스가 유지되는 것을 확인할 수 있다.

정리

### 3. 메시지, 국제화

#인강/5. 스프링 MVC 2/강의#

#### 목차

- 3. 메시지, 국제화 - 프로젝트 설정
- 3. 메시지, 국제화 - 메시지, 국제화 소개
- 3. 메시지, 국제화 - 스프링 메시지 소스 설정
- 3. 메시지, 국제화 - 스프링 메시지 소스 사용
- 3. 메시지, 국제화 - 웹 애플리케이션에 메시지 적용하기
- 3. 메시지, 국제화 - 웹 애플리케이션에 국제화 적용하기
- 3. 메시지, 국제화 - 정리

#### 프로젝트 설정

이전 프로젝트에 이어서 메시지, 국제화 기능을 학습해보자.

스프링 통합과 품에서 개발한 상품 관리 프로젝트를 일부 수정해서 `message-start`라는 프로젝트에 넣어두었다.

참고로 메시지, 국제화 예제에 집중하기 위해서 복잡한 체크, 셀렉트 박스 관리 기능은 제거했다.

#### 프로젝트 설정 순서

1. `message-start`의 폴더 이름을 `message`로 변경하자.

2. **프로젝트 임포트**

File → Open → 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.

3. `ItemServiceApplication.main()`을 실행해서 프로젝트가 정상 수행되는지 확인하자.

#### 실행

- `http://localhost:8080`
- `http://localhost:8080/message/items`

#### 메시지, 국제화 소개

## 메시지

악덕? 기획자가 화면에 보이는 문구가 마음에 들지 않는다고, 상품명이라는 단어를 모두 상품이름으로 고쳐달라고 하면 어떻게 해야할까?

여러 화면에 보이는 상품명, 가격, 수량 등, `label`에 있는 단어를 변경하려면 다음 화면들을 다 찾아가면서 모두 변경해야 한다. 지금처럼 화면 수가 적으면 문제가 되지 않지만 화면이 수십개 이상이라면 수십개의 파일을 모두 고쳐야 한다.

- `addForm.html`, `editForm.html`, `item.html`, `items.html`

왜냐하면 해당 HTML 파일에 메시지가 하드코딩 되어 있기 때문이다.

이런 다양한 메시지를 한 곳에서 관리하도록 하는 기능을 메시지 기능이라 한다.

예를 들어서 `messages.properties`라는 메시지 관리용 파일을 만들고

```
item=상품
item.id=상품 ID
item.itemName=상품명
item.price=가격
item.quantity=수량
```

각 HTML들은 다음과 같이 해당 데이터를 key 값으로 불러서 사용하는 것이다.

### **addForm.html**

```
<label for="itemName" th:text="#{item.itemName}"></label>
```

### **editForm.html**

```
<label for="itemName" th:text="#{item.itemName}"></label>
```

## 국제화

메시지에서 한 발 더 나가보자.

메시지에서 설명한 메시지 파일(`messages.properties`)을 각 나라별로 별도로 관리하면 서비스를 국제화 할 수 있다.

예를 들어서 다음과 같이 2개의 파일을 만들어서 분류한다.

```
messages_en.properties
```

```
item=Item
item.id=Item ID
```

```
item.itemName=Item Name
item.price=price
item.quantity=quantity
```

#### messages\_ko.properties

```
item=상품
item.id=상품 ID
item.itemName=상품명
item.price=가격
item.quantity=수량
```

영어를 사용하는 사람이면 `messages_en.properties`를 사용하고,  
한국어를 사용하는 사람이면 `messages_ko.properties`를 사용하게 개발하면 된다.

이렇게 하면 사이트를 국제화 할 수 있다.

한국에서 접근한 것인지 영어에서 접근한 것인지는 인식하는 방법은 HTTP `accept-language` 헤더 값을 사용하거나 사용자가 직접 언어를 선택하도록 하고, 쿠키 등을 사용해서 처리하면 된다.

메시지와 국제화 기능을 직접 구현할 수도 있겠지만, 스프링은 기본적인 메시지와 국제화 기능을 모두 제공한다. 그리고 타임리프도 스프링이 제공하는 메시지와 국제화 기능을 편리하게 통합해서 제공한다. 지금부터 스프링이 제공하는 메시지와 국제화 기능을 알아보자.

## 스프링 메시지 소스 설정

스프링은 기본적인 메시지 관리 기능을 제공한다.

메시지 관리 기능을 사용하려면 스프링이 제공하는 `MessageSource`를 스프링 빈으로 등록하면 되는데, `MessageSource`는 인터페이스이다. 따라서 구현체인  `ResourceBundleMessageSource`를 스프링 빈으로 등록하면 된다.

### 직접 등록

```

@Bean
public MessageSource messageSource() {
 ResourceBundleMessageSource messageSource = new
 ResourceBundleMessageSource();
 messageSource.setBasename("messages", "errors");
 messageSource.setDefaultEncoding("utf-8");
 return messageSource;
}

```

- **basename** : 설정 파일의 이름을 지정한다.
  - **messages**로 지정하면 **messages.properties** 파일을 읽어서 사용한다.
  - 추가로 국제화 기능을 적용하려면 **messages\_en.properties**, **messages\_ko.properties**와 같이 파일명 마지막에 언어 정보를 주면된다. 만약 찾을 수 있는 국제화 파일이 없으면 **messages.properties**(언어정보가 없는 파일명)를 기본으로 사용한다.
  - 파일의 위치는 **/resources/messages.properties**에 두면 된다.
  - 여러 파일을 한번에 지정할 수 있다. 여기서는 **messages**, **errors** 둘을 지정했다.
- **defaultEncoding** : 인코딩 정보를 지정한다. **utf-8**을 사용하면 된다.

## 스프링 부트

스프링 부트를 사용하면 스프링 부트가 **MessageSource**를 자동으로 스프링 빈으로 등록한다.

### 스프링 부트 메시지 소스 설정

스프링 부트를 사용하면 다음과 같이 메시지 소스를 설정할 수 있다.

**application.properties**

```
spring.messages.basename=messages,config.i18n.messages
```

### 스프링 부트 메시지 소스 기본 값

**spring.messages.basename=messages**

**MessageSource**를 스프링 빈으로 등록하지 않고, 스프링 부트와 관련된 별도의 설정을 하지 않으면 **messages**라는 이름으로 기본 등록된다. 따라서 **messages\_en.properties**, **messages\_ko.properties**, **messages.properties** 파일만 등록하면 자동으로 인식된다.

## 메시지 파일 만들기

메시지 파일을 만들어보자. 국제화 테스트를 위해서 `messages_en` 파일도 추가하자.

- `messages.properties` : 기본 값으로 사용(한글)
- `messages_en.properties` : 영어 국제화 사용

**주의!** 파일명은 `massage`가 아니라 `messages`다! 마지막 `s`에 주의하자

/resources/messages.properties

### **messages.properties**

```
hello=안녕
hello.name=안녕 {0}
```

/resources/messages\_en.properties

### **messages\_en.properties**

```
hello=hello
hello.name=hello {0}
```

## 스프링 메시지 소스 사용

### **MessageSource** 인터페이스

```
public interface MessageSource {

 String getMessage(String code, @Nullable Object[] args, @Nullable String
defaultMessage, Locale locale);

 String getMessage(String code, @Nullable Object[] args, Locale locale) throws
NoSuchMessageException;
```

`MessgaeSource` 인터페이스를 보면 코드를 포함한 일부 파라미터로 메시지를 읽어오는 기능을 제공한다.

스프링이 제공하는 메시지 소스를 어떻게 사용하는지 테스트 코드를 통해서 학습해보자.

```
test/java/hello/itemservice/message.MessageSourceTest.java
```

```
package hello.itemservice.message;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.MessageSource;

import static org.assertj.core.api.Assertions.*;

@SpringBootTest
public class MessageSourceTest {

 @Autowired
 MessageSource ms;

 @Test
 void helloMessage() {
 String result = ms.getMessage("hello", null, null);
 assertThat(result).isEqualTo("안녕");
 }
}
```

- `ms.getMessage("hello", null, null)`
  - **code:** `hello`
  - **args:** `null`
  - **locale:** `null`

가장 단순한 테스트는 메시지 코드로 `hello`를 입력하고 나머지 값은 `null`을 입력했다.

`locale` 정보가 없으면 `basename`에서 설정한 기본 이름 메시지 파일을 조회한다. `basename`으로 `messages`를 지정 했으므로 `messages.properties` 파일에서 데이터 조회한다.

## MessageSourceTest 추가 - 메시지가 없는 경우, 기본 메시지

```
@Test
```

```

void notFoundMessageCode() {
 assertThatThrownBy(() -> ms.getMessage("no_code", null, null))
 .isInstanceOf(NoSuchMessageException.class);
}

@Test
void notFoundMessageCodeDefaultMessage() {
 String result = ms.getMessage("no_code", null, "기본 메시지", null);
 assertThat(result).isEqualTo("기본 메시지");
}

```

- 메시지가 없는 경우에는 `NoSuchMessageException`이 발생한다.
- 메시지가 없어도 기본 메시지(`defaultMessage`)를 사용하면 기본 메시지가 반환된다.

## MessageSourceTest 추가 - 매개변수 사용

```

@Test
void argumentMessage() {
 String result = ms.getMessage("hello.name", new Object[]{"Spring"}, null);
 assertThat(result).isEqualTo("안녕 Spring");
}

```

다음 메시지의 {0} 부분은 매개변수를 전달해서 치환할 수 있다.

`hello.name=안녕 {0}` → Spring 단어를 매개변수로 전달 → 안녕 Spring

## 국제화 파일 선택

locale 정보를 기반으로 국제화 파일을 선택한다.

- Locale이 `en_US`의 경우 `messages_en_US` → `messages_en` → `messages` 순서로 찾는다.
- `Locale`에 맞추어 구체적인 것이 있으면 구체적인 것을 찾고, 없으면 디폴트를 찾는다고 이해하면 된다.

## MessageSourceTest 추가 - 국제화 파일 선택1

```

@Test
void defaultLang() {
 assertThat(ms.getMessage("hello", null, null)).isEqualTo("안녕");
 assertThat(ms.getMessage("hello", null, Locale.KOREA)).isEqualTo("안녕");
}

```

```
}
```

- `ms.getMessage("hello", null, null)` : locale 정보가 없으므로 `messages` 를 사용
- `ms.getMessage("hello", null, Locale.KOREA)` : locale 정보가 있지만, `message_ko` 가 없으므로 `messages` 를 사용

## MessageSourceTest 추가 - 국제화 파일 선택2

```
@Test
void enLang() {
 assertThat(ms.getMessage("hello", null,
 Locale.ENGLISH)).isEqualTo("hello");
}
```

- `ms.getMessage("hello", null, Locale.ENGLISH)` : locale 정보가 `Locale.ENGLISH` 이므로 `messages_en` 을 찾아서 사용

## 웹 애플리케이션에 메시지 적용하기

실제 웹 애플리케이션에 메시지를 적용해보자.

먼저 메시지를 추가 등록하자.

`messages.properties`

```
label.item=상품
label.item.id=상품 ID
label.item.itemName=상품명
label.item.price=가격
label.item.quantity=수량

page.items=상품 목록
page.item=상품 상세
page.addItem=상품 등록
page.updateItem=상품 수정
```

```
button.save=저장
```

```
button.cancel=취소
```

## 타임리프 메시지 적용

타임리프의 메시지 표현식 `#{...}` 를 사용하면 스프링의 메시지를 편리하게 조회할 수 있다.

예를 들어서 방금 등록한 상품이라는 이름을 조회하려면 `#{label.item}` 이라고 하면 된다.

### 렌더링 전

```
<div th:text="#{label.item}"></h2>
```

### 렌더링 후

```
<div>상품</h2>
```

타임리프 템플릿 파일에 메시지를 적용해보자.

### 적용 대상

```
addForm.html
```

```
editForm.html
```

```
item.html
```

```
items.html
```

### addForm.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
 <link th:href="@{/css/bootstrap.min.css}"
 href="../css/bootstrap.min.css" rel="stylesheet">
```

```
<style>
 .container {
 max-width: 560px;
 }
</style>

</head>
<body>

<div class="container">

 <div class="py-5 text-center">
 <h2 th:text="#{page.addItem}">상품 등록</h2>
 </div>

 <h4 class="mb-3">상품 입력</h4>

 <form action="item.html" th:action th:object="${item}" method="post">
 <div>
 <label for="itemName" th:text="#{label.item.itemName}">상품명</label>
 <input type="text" id="itemName" th:field="*{itemName}"
 class="form-control" placeholder="이름을 입력하세요">
 </div>
 <div>
 <label for="price" th:text="#{label.item.price}">가격</label>
 <input type="text" id="price" th:field="*{price}" class="form-
control" placeholder="가격을 입력하세요">
 </div>
 <div>
 <label for="quantity" th:text="#{label.item.quantity}">수량</label>
 <input type="text" id="quantity" th:field="*{quantity}"
 class="form-control" placeholder="수량을 입력하세요">
 </div>
 <hr class="my-4">
 <div class="row">
 <div class="col">
 <button class="w-100 btn btn-primary btn-lg" type="submit">
 등록
 </button>
 </div>
 </div>
 </form>
</div>
```

```

th:text="#{button.save}">저장</button>

</div>
<div class="col">
 <button class="w-100 btn btn-secondary btn-lg"
 onclick="location.href='items.html'"
 th:onclick="|location.href='@{/message/items}' |"
 type="button" th:text="#{button.cancel}">취소</button>
</div>
</div>

</form>

</div> <!-- /container -->
</body>
</html>

```

### 페이지 이름에 적용

- <h2>상품 등록 폼</h2>
  - <h2 th:text="#{page.addItem}">상품 등록</h2>

### 레이블에 적용

- <label for="itemName">상품명</label>
  - <label for="itemName" th:text="#{label.item.itemName}">상품명</label>
  - <label for="price" th:text="#{label.item.price}">가격</label>
  - <label for="quantity" th:text="#{label.item.quantity}">수량</label>

### 버튼에 적용

- <button type="submit">상품 등록</button>
  - <button type="submit" th:text="#{button.save}">저장</button>
  - <button type="button" th:text="#{button.cancel}">취소</button>

### editForm.html

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">

```

```
<link th:href="@{/css/bootstrap.min.css}"
 href="../css/bootstrap.min.css" rel="stylesheet">

<style>
 .container {
 max-width: 560px;
 }
</style>
</head>
<body>

<div class="container">

 <div class="py-5 text-center">
 <h2 th:text="#{page.updateItem}">상품 수정</h2>
 </div>

 <form action="item.html" th:action th:object="${item}" method="post">
 <div>
 <label for="id" th:text="#{label.item.id}">상품 ID</label>
 <input type="text" id="id" th:field="*{id}" class="form-control" readonly>
 </div>
 <div>
 <label for="itemName" th:text="#{label.item.itemName}">상품명</label>
 <input type="text" id="itemName" th:field="*{itemName}" class="form-control">
 </div>
 <div>
 <label for="price" th:text="#{label.item.price}">가격</label>
 <input type="text" id="price" th:field="*{price}" class="form-control">
 </div>
 <div>
 <label for="quantity" th:text="#{label.item.quantity}">수량</label>
 <input type="text" id="quantity" th:field="*{quantity}" class="form-control">
 </div>
 </form>
</div>
```

```

<hr class="my-4">

<div class="row">
 <div class="col">
 <button class="w-100 btn btn-primary btn-lg" type="submit"
th:text="#{button.save}">저장</button>
 </div>
 <div class="col">
 <button class="w-100 btn btn-secondary btn-lg"
 onclick="location.href='item.html'"
 th:onclick="|location.href=@{/message/items/{itemId}
(itemId=${item.id})}|"
 type="button" th:text="#{button.cancel}">취소</button>
 </div>
</div>

</form>

</div> <!-- /container -->
</body>
</html>

```

## item.html

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
 <link th:href="@{/css/bootstrap.min.css}"
 href="../css/bootstrap.min.css" rel="stylesheet">
 <style>
 .container {
 max-width: 560px;
 }
 </style>
</head>
<body>

```

```
<div class="container">

 <div class="py-5 text-center">
 <h2 th:text="#{page.item}">상품 상세</h2>
 </div>

 <!-- 추가 -->
 <h2 th:if="${param.status}" th:text="저장 완료"></h2>

 <div>
 <label for="itemId" th:text="#{label.item.id}">상품 ID</label>
 <input type="text" id="itemId" name="itemId" class="form-control"
value="1" th:value="${item.id}" readonly>
 </div>
 <div>
 <label for="itemName" th:text="#{label.item.itemName}">상품명</label>
 <input type="text" id="itemName" name="itemName" class="form-control"
value="상품A" th:value="${item.itemName}" readonly>
 </div>
 <div>
 <label for="price" th:text="#{label.item.price}">가격</label>
 <input type="text" id="price" name="price" class="form-control"
value="10000" th:value="${item.price}" readonly>
 </div>
 <div>
 <label for="quantity" th:text="#{label.item.quantity}">수량</label>
 <input type="text" id="quantity" name="quantity" class="form-control"
value="10" th:value="${item.quantity}" readonly>
 </div>

 <hr class="my-4">

 <div class="row">
 <div class="col">
 <button class="w-100 btn btn-primary btn-lg"
 onclick="location.href='editForm.html'"
 th:onclick="|location.href='@{/message/items/{itemId}/
edit(itemId=${item.id})}'|"
 >
```

```

 type="button" th:text="#{page.updateItem}">상품 수정</button>
 </div>
 <div class="col">
 <button class="w-100 btn btn-secondary btn-lg"
 onclick="location.href='items.html'"
 th:onclick="|location.href='@{/message/items}'|"
 type="button" th:text="#{page.items}">목록으로</button>
 </div>
</div> <!-- /container -->
</body>
</html>

```

## items.html

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
 <link th:href="@{/css/bootstrap.min.css}"
 href="../css/bootstrap.min.css" rel="stylesheet">
</head>
<body>

<div class="container" style="max-width: 600px">
 <div class="py-5 text-center">
 <h2 th:text="#{page.items}">상품 목록</h2>
 </div>

 <div class="row">
 <div class="col">
 <button class="btn btn-primary float-end"
 onclick="location.href='addForm.html'"
 th:onclick="|location.href='@{/message/items/add}'|"
 type="button" th:text="#{page.addItem}">상품 등록</button>
 </div>
 </div>

```

```

</div>

<hr class="my-4">
<div>
 <table class="table">
 <thead>
 <tr>
 <th th:text="#{label.item.id}">ID</th>
 <th th:text="#{label.item.itemName}">상품명</th>
 <th th:text="#{label.item.price}">가격</th>
 <th th:text="#{label.item.quantity}">수량</th>
 </tr>
 </thead>
 <tbody>
 <tr th:each="item : ${items}">
 <td>회원id</td>
 <td>상품명</td>
 <td th:text="${item.price}">10000</td>
 <td th:text="${item.quantity}">10</td>
 </tr>
 </tbody>
 </table>
</div>
</div> <!-- /container -->

</body>
</html>

```

다음 부분을 잘 확인해서 메시지를 사용하도록 적용하자.

```

<th>ID</th>
<th>상품명</th>
<th>가격</th>
<th>수량</th>

```

```
<th th:text="#{label.item.id}">ID</th>
<th th:text="#{label.item.itemName}">상품명</th>
<th th:text="#{label.item.price}">가격</th>
<th th:text="#{label.item.quantity}">수량</th>
```

## 실행

잘 동작하는지 확인하기 위해 `messages.properties` 파일의 내용을 가격 → 금액과 같이 변경해서 확인해보자. 정상 동작하면 다시 돌려두자.

참고로 파라미터는 다음과 같이 사용할 수 있다.

```
hello.name=안녕 {0}
<p th:text="#{hello.name(${item.itemName})}"></p>
```

## 정리

지금까지 메시지를 효율적으로 관리하는 방법을 알아보았다. 이제 여기에 더해서 국제화를 웹 애플리케이션에 어떻게 적용하는지 알아보자.

## 웹 애플리케이션에 국제화 적용하기

이번에는 웹 애플리케이션에 국제화를 적용해보자. 먼저 영어 메시지를 추가하자.

`messages_en.properties`

```
label.item=Item
label.item.id=Item ID
label.item.itemName=Item Name
label.item.price=price
label.item.quantity=quantity
```

```
page.items=Item List
page.item=Item Detail
```

```
page.addItem=Item Add
page.updateItem=Item Update

button.save=Save
button.cancel=Cancel
```

사실 이것으로 국제화 작업은 거의 끝났다. 앞에서 템플릿 파일에는 모두 `# {...}` 를 통해서 메시지를 사용하도록 적용해두었기 때문이다.

### 웹으로 확인하기

웹 브라우저의 언어 설정 값을 변경하면서 국제화 적용을 확인해보자.

크롬 브라우저 → 설정 → 언어를 검색하고, 우선 순위를 변경하면 된다.

우선순위를 영어로 변경하고 테스트해보자.

웹 브라우저의 언어 설정 값을 변경하면 요청시 `Accept-Language` 의 값이 변경된다.

`Accept-Language` 는 클라이언트가 서버에 기대하는 언어 정보를 담아서 요청하는 HTTP 요청 헤더이다.  
(더 자세한 내용은 모든 개발자를 위한 HTTP 웹 기본지식 강의를 참고하자.)

### 스프링의 국제화 메시지 선택

앞서 `MessageSource` 테스트에서 보았듯이 메시지 기능은 `Locale` 정보를 알아야 언어를 선택할 수 있다.

결국 스프링도 `Locale` 정보를 알아야 언어를 선택할 수 있는데, 스프링은 언어 선택시 기본으로 `Accept-Language` 헤더의 값을 사용한다.

### LocaleResolver

스프링은 `Locale` 선택 방식을 변경할 수 있도록 `LocaleResolver` 라는 인터페이스를 제공하는데,  
스프링 부트는 기본으로 `Accept-Language` 를 활용하는 `AcceptHeaderLocaleResolver` 를 사용한다.

### LocaleResolver 인터페이스

```
public interface LocaleResolver {

 Locale resolveLocale(HttpServletRequest request);
 void setLocale(HttpServletRequest request, @Nullable HttpServletResponse
 response, @Nullable Locale locale);
```

```
}
```

### LocaleResolver 변경

만약 Locale 선택 방식을 변경하려면 LocaleResolver의 구현체를 변경해서 쿠키나 세션 기반의 Locale 선택 기능을 사용할 수 있다. 예를 들어서 고객이 직접 Locale을 선택하도록 하는 것이다. 관련해서 LocaleResolver를 검색하면 수 많은 예제가 나오니 필요한 분들은 참고하자.

## 정리

## 4. 검증1 - Validation

#인강/5. 스프링 MVC 2/강의#

### 목차

- 4. 검증1 - Validation - 검증 요구사항
- 4. 검증1 - Validation - 프로젝트 설정 V1
- 4. 검증1 - Validation - 검증 직접 처리 - 소개
- 4. 검증1 - Validation - 검증 직접 처리 - 개발
- 4. 검증1 - Validation - 프로젝트 준비 V2
- 4. 검증1 - Validation - BindingResult1
- 4. 검증1 - Validation - BindingResult2
- 4. 검증1 - Validation - FieldError, ObjectError
- 4. 검증1 - Validation - 오류 코드와 메시지 처리1
- 4. 검증1 - Validation - 오류 코드와 메시지 처리2
- 4. 검증1 - Validation - 오류 코드와 메시지 처리3
- 4. 검증1 - Validation - 오류 코드와 메시지 처리4
- 4. 검증1 - Validation - 오류 코드와 메시지 처리5
- 4. 검증1 - Validation - 오류 코드와 메시지 처리6
- 4. 검증1 - Validation - Validator 분리1
- 4. 검증1 - Validation - Validator 분리2
- 4. 검증1 - Validation - 정리

### 검증 요구사항

상품 관리 시스템에 새로운 요구사항이 추가되었다.

#### 요구사항: 검증 로직 추가

- 타입 검증
  - 가격, 수량에 문자가 들어가면 검증 오류 처리
- 필드 검증
  - 상품명: 필수, 공백X
  - 가격: 1000원 이상, 1백만원 이하
  - 수량: 최대 9999
- 특정 필드의 범위를 넘어서는 검증
  - 가격 \* 수량의 합은 10,000원 이상

지금까지 만든 웹 애플리케이션은 폼 입력시 숫자를 문자로 작성하거나해서 검증 오류가 발생하면 오류 화면으로 바로 이동한다. 이렇게 되면 사용자는 처음부터 해당 폼으로 다시 이동해서 입력을 해야 한다. 아마도 이런 서비스라면 사용자는 금방 떠나버릴 것이다. 웹 서비스는 폼 입력시 오류가 발생하면, 고객이 입력한 데이터를 유지한 상태로 어떤 오류가 발생했는지 친절하게 알려주어야 한다.

**컨트롤러의 중요한 역할 중 하나는 HTTP 요청이 정상인지 검증하는 것이다.** 그리고 정상 로직보다 이런 검증 로직을 잘 개발하는 것이 어쩌면 더 어려울 수 있다.

#### 참고: 클라이언트 검증, 서버 검증

- 클라이언트 검증은 조작할 수 있으므로 보안에 취약하다.
- 서버만으로 검증하면, 즉각적인 고객 사용성이 부족해진다.
- 둘을 적절히 섞어서 사용하되, 최종적으로 서버 검증은 필수
- API 방식을 사용하면 API 스펙을 잘 정의해서 검증 오류를 API 응답 결과에 잘 남겨주어야 함

먼저 검증을 직접 구현해보고, 뒤에서 스프링과 타임리프가 제공하는 검증 기능을 활용해보자.

## 프로젝트 설정 V1

이전 프로젝트에 이어서 검증(Validation) 기능을 학습해보자.

이전 프로젝트를 일부 수정해서 validation-start라는 프로젝트에 넣어두었다.

#### 프로젝트 설정 순서

1. validation-start의 폴더 이름을 validation로 변경하자.
2. **프로젝트 임포트**  
File → Open → 해당 프로젝트의 build.gradle을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.
3. ItemServiceApplication.main()을 실행해서 프로젝트가 정상 수행되는지 확인하자.

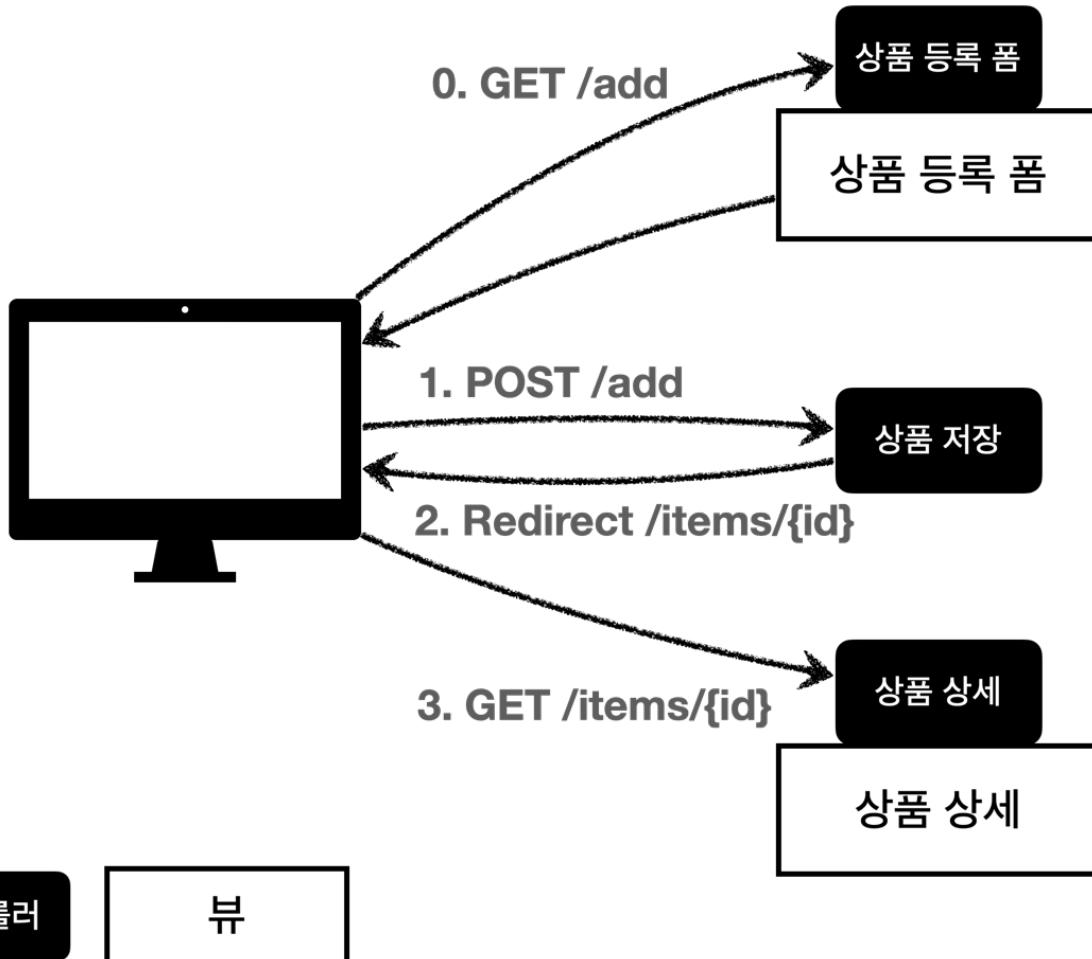
#### 실행

- <http://localhost:8080>
- <http://localhost:8080/validation/v1/items>

## 검증 직접 처리 - 소개

상품 저장 성공

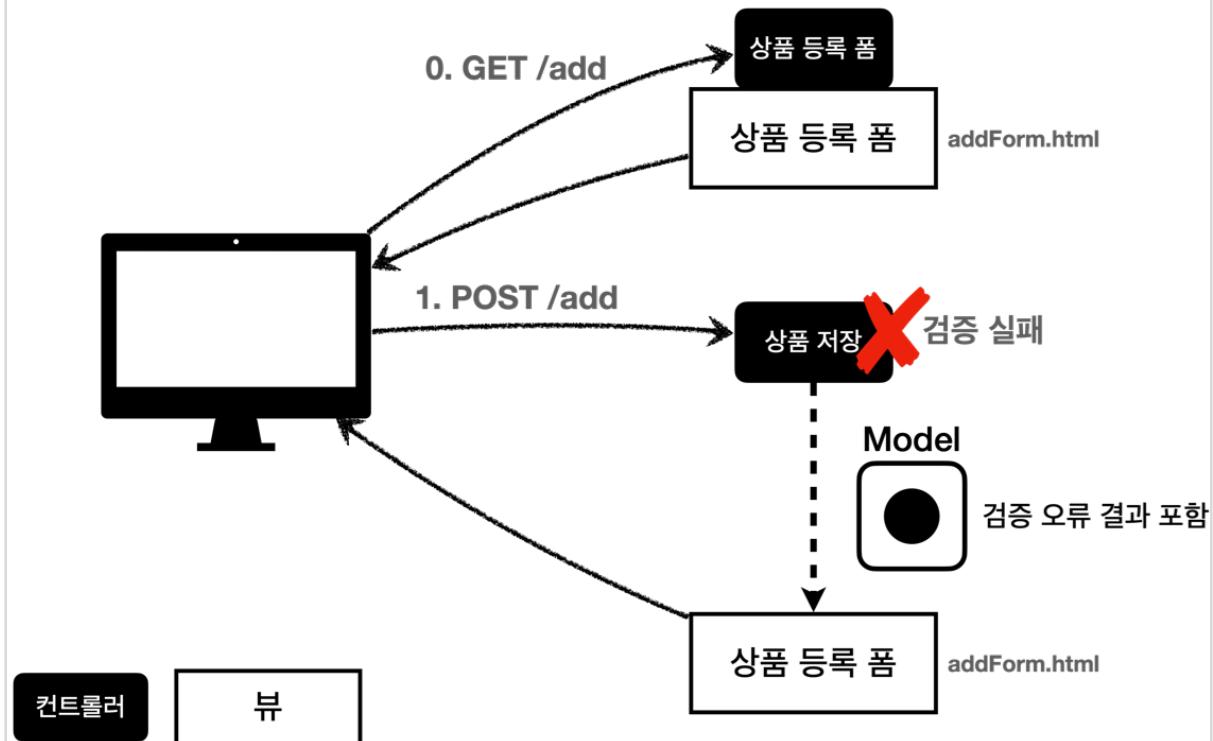
### 상품 저장 성공



사용자가 상품 등록 폼에서 정상 범위의 데이터를 입력하면, 서버에서는 검증 로직이 통과하고, 상품을 저장하고, 상품 상세 화면으로 redirect한다.

상품 저장 검증 실패

## 상품 저장 검증 실패



고객이 상품 등록 폼에서 상품명을 입력하지 않거나, 가격, 수량 등이 너무 작거나 커서 검증 범위를 넘어서면, 서버 검증 로직이 실패해야 한다. 이렇게 검증에 실패한 경우 고객에게 다시 상품 등록 폼을 보여주고, 어떤 값을 잘못 입력했는지 친절하게 알려주어야 한다.

이제 요구사항에 맞추어 검증 로직을 직접 개발해보자.

### 검증 직접 처리 - 개발

#### 상품 등록 검증

먼저 상품 등록 검증 코드를 작성해보자.

#### ValidationItemV1Controller - addItem() 수정

```
@PostMapping("/add")
public String addItem(@ModelAttribute Item item, RedirectAttributes
redirectAttributes, Model model) {

 //검증 오류 결과를 보관
 Map<String, String> errors = new HashMap<>();

 //검증 로직
 if (!StringUtils.hasText(item.getItemName())) {
 errors.put("itemName", "상품 이름은 필수입니다.");
 }
}
```

```

 }

 if (item.getPrice() == null || item.getPrice() < 1000 || item.getPrice() >
1000000) {
 errors.put("price", "가격은 1,000 ~ 1,000,000 까지 허용합니다.");
 }

 if (item.getQuantity() == null || item.getQuantity() >= 9999) {
 errors.put("quantity", "수량은 최대 9,999 까지 허용합니다.");
 }

}

//특정 필드가 아닌 복합 를 검증

if (item.getPrice() != null && item.getQuantity() != null) {
 int resultPrice = item.getPrice() * item.getQuantity();

 if (resultPrice < 10000) {
 errors.put("globalError", "가격 * 수량의 합은 10,000원 이상이어야 합니다.
현재 값 = " + resultPrice);
 }
}

//검증에 실패하면 다시 입력 폼으로

if (!errors.isEmpty()) {
 model.addAttribute("errors", errors);
 return "validation/v1/addForm";
}

//성공 로직

Item savedItem = itemRepository.save(item);
redirectAttributes.addAttribute("itemId", savedItem.getId());
redirectAttributes.addAttribute("status", true);
return "redirect:/validation/v1/items/{itemId}";
}

```

## 검증 오류 보관

```
Map<String, String> errors = new HashMap<>();
```

만약 검증시 오류가 발생하면 어떤 검증에서 오류가 발생했는지 정보를 담아둔다.

## 검증 로직

```
if (!StringUtils.hasText(item.getItemName())) {
 errors.put("itemName", "상품 이름은 필수입니다.");
}
```

import org.springframework.util.StringUtils; 추가 필요

검증시 오류가 발생하면 `errors`에 담아둔다. 이때 어떤 필드에서 오류가 발생했는지 구분하기 위해 오류가 발생한 필드명을 `key`로 사용한다. 이후 뷰에서 이 데이터를 사용해서 고객에게 친절한 오류 메시지를 출력할 수 있다.

### 특정 필드의 범위를 넘어서는 검증 로직

```
//특정 필드의 범위를 넘어서는 검증 로직

if (item.getPrice() != null && item.getQuantity() != null) {
 int resultPrice = item.getPrice() * item.getQuantity();
 if (resultPrice < 10000) {
 errors.put("globalError", "가격 * 수량의 합은 10,000원 이상이어야 합니다. 현재 값
= " + resultPrice);
 }
}
```

특정 필드를 넘어서는 오류를 처리해야 할 수도 있다. 이때는 필드 이름을 넣을 수 없으므로 `globalError`라는 `key`를 사용한다.

### 검증에 실패하면 다시 입력 폼으로

```
if (!errors.isEmpty()) {
 model.addAttribute("errors", errors);
 return "validation/v1/addForm";
}
```

만약 검증에서 오류 메시지가 하나라도 있으면 오류 메시지를 출력하기 위해 `model` `errors`를 담고, 입력 폼이 있는 뷰 템플릿으로 보낸다.

### addForm.html

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
 <link th:href="@{/css/bootstrap.min.css}"
 href="../css/bootstrap.min.css" rel="stylesheet">
 <style>
 .container {
 max-width: 560px;
 }
 .field-error {
 border-color: #dc3545;
 color: #dc3545;
 }
 </style>
</head>
<body>

<div class="container">

 <div class="py-5 text-center">
 <h2 th:text="#{page.addItem}">상품 등록</h2>
 </div>

 <form action="item.html" th:action th:object="${item}" method="post">
 <div th:if="${errors?.containsKey('globalError')}">
 <p class="field-error" th:text="${errors['globalError']}">전체 오류
 메시지</p>
 </div>

 <div>
 <label for="itemName" th:text="#{label.item.itemName}">상품명</label>
 <input type="text" id="itemName" th:field="*{itemName}"
 th:class="${errors?.containsKey('itemName')} ? 'form-control
field-error' : 'form-control'">
 <div class="field-error" th:if="${errors?.containsKey('itemName')}">

```

```

th:text="${errors['itemName']}">
 상품명 오류

```

```

 </div>

```

```

</div>

```

```

<div>
 <label for="price" th:text="#{label.item.price}">가격</label>
 <input type="text" id="price" th:field="*{price}"
 th:class="${errors?.containsKey('price')} ? 'form-control
 field-error' : 'form-control''>
 class="form-control" placeholder="가격을 입력하세요">

```

```

 <div class="field-error" th:if="${errors?.containsKey('price')}">
th:text="${errors['price']}>
 가격 오류

```

```

 </div>

```

```

 </div>

```

```

 <div>
 <label for="quantity" th:text="#{label.item.quantity}">수량</label>
 <input type="text" id="quantity" th:field="*{quantity}"
 th:class="${errors?.containsKey('quantity')} ? 'form-control
 field-error' : 'form-control''>
 class="form-control" placeholder="수량을 입력하세요">

```

```

 <div class="field-error" th:if="${errors?.containsKey('quantity')}">
th:text="${errors['quantity']}>
 수량 오류

```

```

 </div>

```

```

 </div>

```

```

<hr class="my-4">

<div class="row">
 <div class="col">
 <button class="w-100 btn btn-primary btn-lg" type="submit"
th:text="#{button.save}">저장</button>
 </div>
 <div class="col">
 <button class="w-100 btn btn-secondary btn-lg"
 onclick="location.href='items.html'">
 th:onclick="|location.href='@{/validation/v1/items}' |"
 type="button" th:text="#{button.cancel}">취소</button>

```

```
</div>
</div>

</form>

</div> <!-- /container -->
</body>
</html>
```

## css 추가

```
.field-error {
 border-color: #dc3545;
 color: #dc3545;
}
```

이 부분은 오류 메시지를 빨간색으로 강조하기 위해 추가했다.

## 글로벌 오류 메시지

```
<div th:if="${errors?.containsKey('globalError')}">
 <p class="field-error" th:text="${errors['globalError']}>전체 오류 메시지</p>
</div>
```

오류 메시지는 `errors`에 내용이 있을 때만 출력하면 된다. 타임리프의 `th:if`를 사용하면 조건에 만족할 때만 해당 HTML 태그를 출력할 수 있다.

## 참고 Safe Navigation Operator

만약 여기에서 `errors` 가 `null`이라면 어떻게 될까?

생각해보면 등록폼에 진입한 시점에는 `errors` 가 없다.

따라서 `errors.containsKey()` 를 호출하는 순간 `NullPointerException`이 발생한다.

`errors?.` 은 `errors` 가 `null` 일때 `NullPointerException`이 발생하는 대신, `null`을 반환하는 문법이다.

`th:if`에서 `null`은 실패로 처리되므로 오류 메시지가 출력되지 않는다.

이것은 스프링의 SpringEL이 제공하는 문법이다. 자세한 내용은 다음을 참고하자.

<https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#expressions-operator-safe-navigation>

가격 1000원, 수량 1개를 선택하면 다음과 같은 HTML 결과 화면을 볼 수 있다.

```
<div>
 <p class="field-error">가격 * 수량의 합은 10,000원 이상이어야 합니다. 현재 값 = 1000</p>
</div>
```

## 필드 오류 처리

```
<input type="text" th:classappend="${errors?.containsKey('itemName')} ? 'field-error' : '_'" class="form-control">
```

`classappend` 를 사용해서 해당 필드에 오류가 있으면 `field-error`라는 클래스 정보를 더해서 폼의 색깔을 빨간색으로 강조한다. 만약 값이 없으면 `_` (No-Operation)을 사용해서 아무것도 하지 않는다.

## 필드 오류 처리 - 입력 폼 색상 적용

```
<input type="text" class="form-control field-error">
```

## 필드 오류 처리 - 메시지

```
<div class="field-error" th:if="${errors?.containsKey('itemName')}" th:text="${errors['itemName']}>
 상품명 오류
</div>
```

글로벌 오류 메시지에서 설명한 내용과 동일하고, 필드 오류를 대상으로 한다.

## 실행

상품 등록을 실행하고 검증이 잘 동작 하는지 확인해보자.

- <http://localhost:8080/validation/v1/items/add>

상품 수정의 검증은 더 효율적인 검증 처리 방법을 학습한 다음에 진행한다.

## 정리

- 만약 검증 오류가 발생하면 입력 폼을 다시 보여준다.
- 검증 오류들을 고객에게 친절하게 안내해서 다시 입력할 수 있게 한다.
- 검증 오류가 발생해도 고객이 입력한 데이터가 유지된다.

## 남은 문제점

- 뷰 템플릿에서 중복 처리가 많다. 뭔가 비슷하다.
- 타입 오류 처리가 안된다. `Item` 의 `price`, `quantity` 같은 숫자 필드는 타입이 `Integer` 이므로 문자 타입으로 설정하는 것이 불가능하다. 숫자 타입에 문자가 들어오면 오류가 발생한다. 그런데 이러한 오류는 스프링MVC에서 컨트롤러에 진입하기도 전에 예외가 발생하기 때문에, 컨트롤러가 호출되지도 않고, 400 예외가 발생하면서 오류 페이지를 띄워준다.
- `Item` 의 `price` 에 문자를 입력하는 것처럼 타입 오류가 발생해도 고객이 입력한 문자를 화면에 남겨야 한다. 만약 컨트롤러가 호출된다고 가정해도 `Item` 의 `price` 는 `Integer` 이므로 문자를 보관할 수가 없다. 결국 문자는 바인딩이 불가능하므로 고객이 입력한 문자가 사라지게 되고, 고객은 본인이 어떤 내용을 입력해서 오류가 발생했는지 이해하기 어렵다.
- 결국 고객이 입력한 값도 어딘가에 별도로 관리가 되어야 한다.

지금부터 스프링이 제공하는 검증 방법을 하나씩 알아보자.

## 프로젝트 준비 V2

앞서 만든 기능을 유지하기 위해, 컨트롤러와 템플릿 파일을 복사하자.

### ValidationItemControllerV2 컨트롤러 생성

- `hello.itemservice.web.validation.ValidationItemControllerV1` 복사
- `hello.itemservice.web.validation.ValidationItemControllerV2` 붙여넣기
- URL 경로 변경: `validation/v1/` → `validation/v2/`

### 템플릿 파일 복사

- `validation/v1` 디렉토리의 모든 템플릿 파일을 `validation/v2` 디렉토리로 복사
- `/resources/templates/validation/v1/` → `/resources/templates/validation/v2/`

- addForm.html
  - editForm.html
  - item.html
  - items.html
- /resources/templates/validation/v2/ 하위 4개 파일 모두 URL 경로 변경: validation/v1/ → validation/v2/
    - addForm.html
    - editForm.html
    - item.html
    - items.html

## 실행

<http://localhost:8080/validation/v2/items>

실행 후 웹 브라우저의 URL이 validation/v2 으로 잘 유지되는지 확인해자.

## BindingResult1

지금부터 스프링이 제공하는 검증 오류 처리 방법을 알아보자. 여기서 핵심은 **BindingResult**이다. 우선 코드로 확인해보자.

### ValidationItemControllerV2 - addItemV1

```

@PostMapping("/add")
public String addItemV1(@ModelAttribute Item item, BindingResult bindingResult,
RedirectAttributes redirectAttributes) {

 if (!StringUtils.hasText(item.getItemName())) {
 bindingResult.addError(new FieldError("item", "itemName", "상품 이름은
필수입니다."));
 }

 if (item.getPrice() == null || item.getPrice() < 1000 || item.getPrice() >
1000000) {
 bindingResult.addError(new FieldError("item", "price", "가격은 1,000 ~
1,000,000 까지 허용합니다."));
 }
}

```

```

 if (item.getQuantity() == null || item.getQuantity() > 10000) {
 bindingResult.addError(new FieldError("item", "quantity", "수량은 최대
9,999 까지 허용합니다."));
 }

 //특정 필드 예외가 아닌 전체 예외
 if (item.getPrice() != null && item.getQuantity() != null) {
 int resultPrice = item.getPrice() * item.getQuantity();
 if (resultPrice < 10000) {
 bindingResult.addError(new ObjectError("item", "가격 * 수량의 합은
10,000원 이상이어야 합니다. 현재 값 = " + resultPrice));
 }
 }
 }

 if (bindingResult.hasErrors()) {
 log.info("errors={}", bindingResult);
 return "validation/v2/addForm";
 }

 //성공 로직
 Item savedItem = itemRepository.save(item);
 redirectAttributes.addAttribute("itemId", savedItem.getId());
 redirectAttributes.addAttribute("status", true);
 return "redirect:/validation/v2/items/{itemId}";
}

```

## 코드 변경

- 메서드 이름 변경: `addItem()` → `addItemV1()`
- `@Slf4j` : 로그 출력을 위해 추가

## 주의

`BindingResult bindingResult` 파라미터의 위치는 `@ModelAttribute Item item` 다음에 와야 한다.

## 필드 오류 - `FieldError`

```

if (!StringUtils.hasText(item.getItemName())) {
 bindingResult.addError(new FieldError("item", "itemName", "상품 이름은

```

```
 필수입니다."));
```

```
}
```

## FieldError 생성자 요약

```
public FieldError(String objectName, String field, String defaultMessage) {}
```

필드에 오류가 있으면 `FieldError` 객체를 생성해서 `bindingResult`에 담아두면 된다.

- `objectName` : `@ModelAttribute`의 이름
- `field` : 오류가 발생한 필드 이름
- `defaultMessage` : 오류 기본 메시지

## 글로벌 오류 - ObjectError

```
bindingResult.addError(new ObjectError("item", "가격 * 수량의 합은 10,000원 이상이어야 합니다. 현재 값 = " + resultPrice));
```

## ObjectError 생성자 요약

```
public ObjectError(String objectName, String defaultMessage) {}
```

특정 필드를 넘어서는 오류가 있으면 `ObjectError` 객체를 생성해서 `bindingResult`에 담아두면 된다.

- `objectName` : `@ModelAttribute`의 이름
- `defaultMessage` : 오류 기본 메시지

## validation/v2/addForm.html 수정

```
<form action="item.html" th:action th:object="${item}" method="post">

 <div th:if="#{fields.hasGlobalErrors()}">
 <p class="field-error" th:each="err : #{fields.globalErrors()}" th:text="${err}">글로벌 오류 메시지</p>
 </div>
```

```

<div>
 <label for="itemName" th:text="#{label.item.itemName}">상품명</label>
 <input type="text" id="itemName" th:field="*{itemName}"
 th:errorclass="field-error" class="form-control"
 placeholder="이름을 입력하세요">
 <div class="field-error" th:errors="*{itemName}">
 상품명 오류
 </div>
 </div>
 <div>
 <label for="price" th:text="#{label.item.price}">가격</label>
 <input type="text" id="price" th:field="*{price}"
 th:errorclass="field-error" class="form-control"
 placeholder="가격을 입력하세요">
 <div class="field-error" th:errors="*{price}">
 가격 오류
 </div>
 </div>
 <div>
 <label for="quantity" th:text="#{label.item.quantity}">수량</label>
 <input type="text" id="quantity" th:field="*{quantity}"
 th:errorclass="field-error" class="form-control"
 placeholder="수량을 입력하세요">
 <div class="field-error" th:errors="*{quantity}">
 수량 오류
 </div>
 </div>

```

## 타임리프 스프링 검증 오류 통합 기능

타임리프는 스프링의 `BindingResult` 를 활용해서 편리하게 검증 오류를 표현하는 기능을 제공한다.

- `#fields` : `#fields` 로 `BindingResult` 가 제공하는 검증 오류에 접근할 수 있다.
- `th:errors` : 해당 필드에 오류가 있는 경우에 태그를 출력한다. `th:if` 의 편의 버전이다.
- `th:errorclass` : `th:field` 에서 지정한 필드에 오류가 있으면 `class` 정보를 추가한다.
  
- 검증과 오류 메시지 공식 매뉴얼
  - <https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html#validation-and>

## error-messages

### 글로벌 오류 처리

```
<div th:if="#{fields.hasGlobalErrors()}">
 <p class="field-error" th:each="err : ${fields.globalErrors()}" th:text="${err}">전체 오류 메시지</p>
</div>
```

### 필드 오류 처리

```
<input type="text" id="itemName" th:field="*{itemName}"
 th:errorclass="field-error" class="form-control" placeholder="이름을
입력하세요">

<div class="field-error" th:errors="*{itemName}">
 상품명 오류
</div>
```

## BindingResult2

- 스프링이 제공하는 검증 오류를 보관하는 객체이다. 검증 오류가 발생하면 여기에 보관하면 된다.
- BindingResult 가 있으면 @ModelAttribute 에 데이터 바인딩 시 오류가 발생해도 컨트롤러가 호출된다!

### 예) @ModelAttribute에 바인딩 시 타입 오류가 발생하면?

- BindingResult 가 없으면 → 400 오류가 발생하면서 컨트롤러가 호출되지 않고, 오류 페이지로 이동한다.
- BindingResult 가 있으면 → 오류 정보( FieldError )를 BindingResult 에 담아서 컨트롤러를 정상 호출한다.

### BindingResult에 검증 오류를 적용하는 3가지 방법

- @ModelAttribute 의 객체에 타입 오류 등으로 바인딩이 실패하는 경우 스프링이 FieldError 생성해서 BindingResult 에 넣어준다.

- 개발자가 직접 넣어준다.
- `Validator` 사용 → 이것은 뒤에서 설명

### 타입 오류 확인

숫자가 입력되어야 할 곳에 문자를 입력해서 타입을 다르게 해서 `BindingResult` 를 호출하고 `bindingResult` 의 값을 확인해보자.

### 주의

- `BindingResult` 는 검증할 대상 바로 다음에 와야한다. 순서가 중요하다. 예를 들어서 `@ModelAttribute Item item`, 바로 다음에 `BindingResult` 가 와야 한다.
- `BindingResult` 는 Model에 자동으로 포함된다.

### BindingResult와 Errors

- `org.springframework.validation.Errors`
- `org.springframework.validation.BindingResult`

`BindingResult` 는 인터페이스이고, `Errors` 인터페이스를 상속받고 있다.

실제 넘어오는 구현체는 `BeanPropertyBindingResult` 라는 것인데, 둘다 구현하고 있으므로 `BindingResult` 대신에 `Errors` 를 사용해도 된다. `Errors` 인터페이스는 단순한 오류 저장과 조회 기능을 제공한다. `BindingResult` 는 여기에 더해서 추가적인 기능들을 제공한다. `addError()` 도 `BindingResult` 가 제공하므로 여기서는 `BindingResult` 를 사용하자. 주로 관례상 `BindingResult` 를 많이 사용한다.

### 정리

`BindingResult`, `FieldError`, `ObjectError` 를 사용해서 오류 메시지를 처리하는 방법을 알아보았다. 그런데 오류가 발생하는 경우 고객이 입력한 내용이 모두 사라진다. 이 문제를 해결해보자.

### FieldError, ObjectError

#### 목표

- 사용자 입력 오류 메시지가 화면에 남도록 하자.
  - 예) 가격을 1000원 미만으로 설정시 입력한 값이 남아있어야 한다.
- `FieldError`, `ObjectError`에 대해서 더 자세히 알아보자.

### ValidationItemControllerV2 - addItemV2

```

@PostMapping("/add")
public String addItemV2(@ModelAttribute Item item, BindingResult bindingResult,
RedirectAttributes redirectAttributes) {

 if (!StringUtils.hasText(item.getItemName())) {
 bindingResult.addError(new FieldError("item", "itemName",
item.getItemName(), false, null, null, "상품 이름은 필수입니다."));
 }

 if (item.getPrice() == null || item.getPrice() < 1000 || item.getPrice() >
1000000) {
 bindingResult.addError(new FieldError("item", "price", item.getPrice(),
false, null, null, "가격은 1,000 ~ 1,000,000 까지 허용합니다."));
 }

 if (item.getQuantity() == null || item.getQuantity() > 10000) {
 bindingResult.addError(new FieldError("item", "quantity",
item.getQuantity(), false, null, null, "수량은 최대 9,999 까지 허용합니다."));
 }

 //특정 필드 예외가 아닌 전체 예외
 if (item.getPrice() != null && item.getQuantity() != null) {
 int resultPrice = item.getPrice() * item.getQuantity();
 if (resultPrice < 10000) {
 bindingResult.addError(new ObjectError("item", null, null, "가격 *
수량의 합은 10,000원 이상이어야 합니다. 현재 값 = " + resultPrice));
 }
 }

 if (bindingResult.hasErrors()) {
 log.info("errors={}", bindingResult);
 return "validation/v2/addForm";
 }

 //성공 로직
 Item savedItem = itemRepository.save(item);
 redirectAttributes.addAttribute("itemId", savedItem.getId());
 redirectAttributes.addAttribute("status", true);
 return "redirect:/validation/v2/items/{itemId}";
}

```

## 코드 변경

- `addItemV1()` 의 `@PostMapping("/add")` 을 주석 처리하자.

## FieldError 생성자

`FieldError` 는 두 가지 생성자를 제공한다.

```
public FieldError(String objectName, String field, String defaultMessage);
public FieldError(String objectName, String field, @Nullable Object
rejectedValue, boolean bindingFailure, @Nullable String[] codes, @Nullable
Object[] arguments, @Nullable String defaultMessage)
```

## 파라미터 목록

- `objectName` : 오류가 발생한 객체 이름
- `field` : 오류 필드
- `rejectedValue` : 사용자가 입력한 값(거절된 값)
- `bindingFailure` : 타입 오류 같은 바인딩 실패인지, 검증 실패인지 구분 값
- `codes` : 메시지 코드
- `arguments` : 메시지에서 사용하는 인자
- `defaultMessage` : 기본 오류 메시지

`ObjectError` 도 유사하게 두 가지 생성자를 제공한다. 코드를 참고하자.

## 오류 발생시 사용자 입력 값 유지

```
new FieldError("item", "price", item.getPrice(), false, null, null, "가격은 1,000 ~
1,000,000 까지 허용합니다.")
```

사용자의 입력 데이터가 컨트롤러의 `@ModelAttribute` 에 바인딩되는 시점에 오류가 발생하면 모델 객체에 사용자 입력 값을 유지하기 어렵다. 예를 들어서 가격에 숫자가 아닌 문자가 입력된다면 가격은 `Integer` 타입이므로 문자를 보관할 수 있는 방법이 없다. 그래서 오류가 발생한 경우 사용자 입력 값을 보관하는 별도의 방법이 필요하다. 그리고 이렇게 보관한 사용자 입력 값을 검증 오류 발생시 화면에 다시 출력하면 된다.

`FieldError` 는 오류 발생시 사용자 입력 값을 저장하는 기능을 제공한다.

여기서 `rejectedValue` 가 바로 오류 발생시 사용자 입력 값을 저장하는 필드다.

`bindingFailure` 는 타입 오류 같은 바인딩이 실패했는지 여부를 적어주면 된다. 여기서는 바인딩이 실패한 것은 아니기 때문에 `false` 를 사용한다.

## 타임리프의 사용자 입력 값 유지

th:field="\*{price}"

타임리프의 th:field 는 매우 똑똑하게 동작하는데, 정상 상황에는 모델 객체의 값을 사용하지만, 오류가 발생하면 FieldError에서 보관한 값을 사용해서 값을 출력한다.

## 스프링의 바인딩 오류 처리

타입 오류로 바인딩에 실패하면 스프링은 FieldError를 생성하면서 사용자가 입력한 값을 넣어둔다.

그리고 해당 오류를 BindingResult에 담아서 컨트롤러를 호출한다. 따라서 타입 오류 같은 바인딩 실패시에도 사용자의 오류 메시지를 정상 출력할 수 있다.

## 오류 코드와 메시지 처리 1

### 목표

오류 메시지를 체계적으로 다루어보자.

### FieldError 생성자

FieldError는 두 가지 생성자를 제공한다.

```
public FieldError(String objectName, String field, String defaultMessage);
public FieldError(String objectName, String field, @Nullable Object
rejectedValue, boolean bindingFailure, @Nullable String[] codes, @Nullable
Object[] arguments, @Nullable String defaultMessage)
```

### 파라미터 목록

- objectName : 오류가 발생한 객체 이름
- field : 오류 필드
- rejectedValue : 사용자가 입력한 값(거절된 값)
- bindingFailure : 타입 오류 같은 바인딩 실패인지, 검증 실패인지 구분 값
- codes : 메시지 코드
- arguments : 메시지에서 사용하는 인자
- defaultMessage : 기본 오류 메시지

FieldError, ObjectError의 생성자는 errorCode, arguments를 제공한다. 이것은 오류 발생시 오류 코드로 메시지를 찾기 위해 사용된다.

## errors 메시지 파일 생성

messages.properties 를 사용해도 되지만, 오류 메시지를 구분하기 쉽게 errors.properties 라는 별도의 파일로 관리해보자.

먼저 스프링 부트가 해당 메시지 파일을 인식할 수 있게 다음 설정을 추가한다. 이렇게하면

messages.properties, errors.properties 두 파일을 모두 인식한다. (생략하면  
messages.properties 를 기본으로 인식한다.)

## 스프링 부트 메시지 설정 추가

application.properties

```
spring.messages.basename=messages,errors
```

## errors.properties 추가

src/main/resources/errors.properties

```
required.item.itemName=상품 이름은 필수입니다.
```

```
range.item.price=가격은 {0} ~ {1} 까지 허용합니다.
```

```
max.item.quantity=수량은 최대 {0} 까지 허용합니다.
```

```
totalPriceMin=가격 * 수량의 합은 {0}원 이상이어야 합니다. 현재 값 = {1}
```

| 참고: errors\_en.properties 파일을 생성하면 오류 메시지도 국제화 처리를 할 수 있다.

이제 errors에 등록한 메시지를 사용하도록 코드를 변경해보자.

## ValidationItemControllerV2 - addItemV3() 추가

```
@PostMapping("/add")
public String addItemV3(@ModelAttribute Item item, BindingResult bindingResult,
RedirectAttributes redirectAttributes) {

 if (!StringUtils.hasText(item.getItemName())) {
 bindingResult.addError(new FieldError("item", "itemName",
item.getItemName(), false, new String[]{"required.item.itemName"}, null,
null));
 }
}
```

```

 }

 if (item.getPrice() == null || item.getPrice() < 1000 || item.getPrice() >
1000000) {
 bindingResult.addError(new FieldError("item", "price", item.getPrice(),
false, new String[]{"range.item.price"}, new Object[]{1000, 1000000}, null));
 }

 if (item.getQuantity() == null || item.getQuantity() > 10000) {
 bindingResult.addError(new FieldError("item", "quantity",
item.getQuantity(), false, new String[]{"max.item.quantity"}, new Object[]
{9999}, null));
 }
}

//특정 필드 예외가 아닌 전체 예외

if (item.getPrice() != null && item.getQuantity() != null) {
 int resultPrice = item.getPrice() * item.getQuantity();
 if (resultPrice < 10000) {
 bindingResult.addError(new ObjectError("item", new String[]
{"totalPriceMin"}, new Object[]{10000, resultPrice}, null));
 }
}

if (bindingResult.hasErrors()) {
 log.info("errors={}", bindingResult);
 return "validation/v2/addForm";
}

//성공 로직

Item savedItem = itemRepository.save(item);
redirectAttributes.addAttribute("itemId", savedItem.getId());
redirectAttributes.addAttribute("status", true);
return "redirect:/validation/v2/items/{itemId}";
}

```

## 코드 변경

- `addItemV2()` 의 `@PostMapping` 부분 주석 처리

```
//range.item.price=가격은 {0} ~ {1} 까지 허용합니다.
new FieldError("item", "price", item.getPrice(), false, new String[]
{"range.item.price"}, new Object[]{1000, 1000000})
```

- codes : required.item.itemName 를 사용해서 메시지 코드를 지정한다. 메시지 코드는 하나가 아니라 배열로 여러 값을 전달할 수 있는데, 순서대로 매칭해서 처음 매칭되는 메시지가 사용된다.
- arguments : Object[]{1000, 1000000} 를 사용해서 코드의 {0}, {1} 로 치환할 값을 전달한다.

### 실행

실행해보면 메시지, 국제화에서 학습한 MessageSource 를 찾아서 메시지를 조회하는 것을 확인할 수 있다.

## 오류 코드와 메시지 처리2

### 목표

- FieldError, ObjectError 는 다루기 너무 번거롭다.
- 오류 코드도 좀 더 자동화 할 수 있지 않을까? 예) item.itemName 처럼?

컨트롤러에서 BindingResult 는 검증해야 할 객체인 target 바로 다음에 온다. 따라서 BindingResult 는 이미 본인이 검증해야 할 객체인 target 을 알고 있다.

다음을 컨트롤러에서 실행해보자.

```
log.info("objectName={}", bindingResult.getObjectName());
log.info("target={}", bindingResult.getTarget());
```

### 출력 결과

```
objectName=item //@ModelAttribute name
target=Item(id=null, itemName=상품, price=100, quantity=1234)
```

rejectValue(), reject()

`BindingResult` 가 제공하는 `rejectValue()`, `reject()` 를 사용하면 `FieldError`, `ObjectError` 를 직접 생성하지 않고, 깔끔하게 검증 오류를 다룰 수 있다.

`rejectValue()`, `reject()` 를 사용해서 기존 코드를 단순화해보자.

## ValidationItemControllerV2 - addItemV4() 추가

```
@PostMapping("/add")
public String addItemV4(@ModelAttribute Item item, BindingResult bindingResult,
RedirectAttributes redirectAttributes) {

 log.info("objectName={}, bindingResult.getObjectName()");
 log.info("target={}, bindingResult.getTarget());

 if (!StringUtils.hasText(item.getItemName())) {
 bindingResult.rejectValue("itemName", "required");
 }

 if (item.getPrice() == null || item.getPrice() < 1000 || item.getPrice() >
1000000) {
 bindingResult.rejectValue("price", "range", new Object[]{1000,
1000000}, null);
 }

 if (item.getQuantity() == null || item.getQuantity() > 10000) {
 bindingResult.rejectValue("quantity", "max", new Object[]{9999}, null);
 }

 //특정 필드 예외가 아닌 전체 예외
 if (item.getPrice() != null && item.getQuantity() != null) {
 int resultPrice = item.getPrice() * item.getQuantity();
 if (resultPrice < 10000) {
 bindingResult.reject("totalPriceMin", new Object[]{10000,
resultPrice}, null);
 }
 }

 if (bindingResult.hasErrors()) {
 log.info("errors={}", bindingResult);
 return "validation/v2/addForm";
 }
}
```

```

//성공 로직

Item savedItem = itemRepository.save(item);
redirectAttributes.addAttribute("itemId", savedItem.getId());
redirectAttributes.addAttribute("status", true);
return "redirect:/validation/v2/items/{itemId}";
}

```

### 코드 변경

- `addItemV2()` 의 `@PostMapping` 부분 주석 처리
- 

### 실행

오류 메시지가 정상 출력된다. 그런데 `errors.properties`에 있는 코드를 직접 입력하지 않았는데 어떻게 된 것일까?

### `rejectValue()`

```

void rejectValue(@Nullable String field, String errorCode,
 @Nullable Object[] errorArgs, @Nullable String defaultMessage);

```

- `field` : 오류 필드명
- `errorCode` : 오류 코드(이 오류 코드는 메시지에 등록된 코드가 아니다. 뒤에서 설명할 `messageResolver`를 위한 오류 코드이다.)
- `errorArgs` : 오류 메시지에서 `{0}` 을 치환하기 위한 값
- `defaultMessage` : 오류 메시지를 찾을 수 없을 때 사용하는 기본 메시지

```
bindingResult.rejectValue("price", "range", new Object[]{1000, 1000000}, null)
```

앞에서 `BindingResult` 는 어떤 객체를 대상으로 검증하는지 `target`을 이미 알고 있다고 했다. 따라서 `target(item)`에 대한 정보는 없어도 된다. 오류 필드명은 동일하게 `price`를 사용했다.

### 축약된 오류 코드

`FieldError()` 를 직접 다룰 때는 오류 코드를 `range.item.price` 와 같이 모두 입력했다. 그런데 `rejectValue()` 를 사용하고 부터는 오류 코드를 `range` 로 간단하게 입력했다. 그래도 오류 메시지를 잘

찾아서 출력한다. 무언가 규칙이 있는 것처럼 보인다. 이 부분을 이해하려면 `MessageCodesResolver`를 이해해야 한다. 왜 이런식으로 오류 코드를 구성하는지 바로 다음에 자세히 알아보자.

#### errors.properties

```
range.item.price=가격은 {0} ~ {1} 까지 허용합니다.
```

#### reject()

```
void reject(String errorCode, @Nullable Object[] errorArgs, @Nullable String defaultMessage);
```

앞의 내용과 같다.

### 오류 코드와 메시지 처리3

오류 코드를 만들 때 다음과 같이 자세히 만들 수도 있고,

`required.item.itemName` : 상품 이름은 필수입니다.

`range.item.price` : 상품의 가격 범위 오류입니다.

또는 다음과 같이 단순하게 만들 수도 있다.

`required` : 필수 값입니다.

`range` : 범위 오류입니다.

단순하게 만들면 범용성이 좋아서 여러곳에서 사용할 수 있지만, 메시지를 세밀하게 작성하기 어렵다.

반대로 너무 자세하게 만들면 범용성이 떨어진다. 가장 좋은 방법은 범용성으로 사용하다가, 세밀하게 작성해야 하는 경우에는 세밀한 내용이 적용되도록 메시지에 단계를 두는 방법이다.

예를 들어서 `required`라고 오류 코드를 사용한다고 가정해보자.

다음과 같이 `required`라는 메시지만 있으면 이 메시지를 선택해서 사용하는 것이다.

```
required: 필수 값입니다.
```

그런데 오류 메시지에 `required.item.itemName`과 같이 객체명과 필드명을 조합한 세밀한 메시지

코드가 있으면 이 메시지를 높은 우선순위로 사용하는 것이다.

```
#Level1
required.item.itemName: 상품 이름은 필수 입니다.

#Level2
required: 필수 값 입니다.
```

물론 이렇게 객체명과 필드명을 조합한 메시지가 있는지 우선 확인하고, 없으면 좀 더 범용적인 메시지를 선택하도록 추가 개발을 해야겠지만, 범용성 있게 잘 개발해두면, 메시지의 추가 만으로 매우 편리하게 오류 메시지를 관리할 수 있을 것이다.

스프링은 **MessageCodesResolver**라는 것으로 이러한 기능을 지원한다.

## 오류 코드와 메시지 처리4

우선 테스트 코드로 **MessageCodesResolver**를 알아보자.

### MessageCodesResolverTest

```
package hello.itemservice.validation;

import org.junit.jupiter.api.Test;
import org.springframework.validation.DefaultMessageCodesResolver;
import org.springframework.validation.MessageCodesResolver;

import static org.assertj.core.api.Assertions.assertThat;

public class MessageCodesResolverTest {

 MessageCodesResolver codesResolver = new DefaultMessageCodesResolver();

 @Test
 void messageCodesResolverObject() {
 String[] messageCodes = codesResolver.resolveMessageCodes("required",
 "item");
```

```

 assertThat(messageCodes).containsExactly("required.item", "required");
 }

 @Test
 void messageCodesResolverField() {
 String[] messageCodes = codesResolver.resolveMessageCodes("required",
 "item", "itemName", String.class);
 assertThat(messageCodes).containsExactly(
 "required.item.itemName",
 "required.itemName",
 "required.java.lang.String",
 "required"
);
 }
}

```

## MessageCodesResolver

- 검증 오류 코드로 메시지 코드들을 생성한다.
- `MessageCodesResolver` 인터페이스이고 `DefaultMessageCodesResolver`는 기본 구현체이다.
- 주로 다음과 함께 사용 `ObjectError`, `FieldError`

## DefaultMessageCodesResolver의 기본 메시지 생성 규칙

### 객체 오류

객체 오류의 경우 다음 순서로 2가지 생성

- 1.: code + "." + object name
- 2.: code

예) 오류 코드: required, object name: item

- 1.: required.item
- 2.: required

### 필드 오류

필드 오류의 경우 다음 순서로 4가지 메시지 코드 생성

```
1.: code + "." + object name + "." + field
2.: code + "." + field
3.: code + "." + field type
4.: code
```

예) 오류 코드: typeMismatch, object name "user", field "age", field type: int

1. "typeMismatch.user.age"
2. "typeMismatch.age"
3. "typeMismatch.int"
4. "typeMismatch"

## 동작 방식

- `rejectValue()`, `reject()`는 내부에서 `MessageCodesResolver`를 사용한다. 여기에서 메시지 코드들을 생성한다.
- `FieldError`, `ObjectError`의 생성자를 보면, 오류 코드를 하나가 아니라 여러 오류 코드를 가질 수 있다. `MessageCodesResolver`를 통해서 생성된 순서대로 오류 코드를 보관한다.
- 이 부분을 `BindingResult`의 로그를 통해서 확인해보자.
  - `codes [range.item.price, range.price, range.java.lang.Integer, range]`

**FieldError** `rejectValue("itemName", "required")`

다음 4가지 오류 코드를 자동으로 생성

- `required.item.itemName`
- `required.itemName`
- `required.java.lang.String`
- `required`

**ObjectError** `reject("totalPriceMin")`

다음 2가지 오류 코드를 자동으로 생성

- `totalPriceMin.item`
- `totalPriceMin`

## 오류 메시지 출력

타임리프 화면을 렌더링 할 때 `th:errors` 가 실행된다. 만약 이때 오류가 있다면 생성된 오류 메시지 코드를 순서대로 돌아가면서 메시지를 찾는다. 그리고 없으면 디폴트 메시지를 출력한다.

## 오류 코드와 메시지 처리5

### 오류 코드 관리 전략

**핵심은 구체적인 것에서! 덜 구체적인 것으로!**

MessageCodesResolver 는 required.item.itemName 처럼 구체적인 것을 먼저 만들어주고, required 처럼 덜 구체적인 것을 가장 나중에 만든다.

이렇게 하면 앞서 말한 것처럼 메시지와 관련된 공통 전략을 편리하게 도입할 수 있다.

### 왜 이렇게 복잡하게 사용하는가?

모든 오류 코드에 대해서 메시지를 각각 다 정의하면 개발자 입장에서 관리하기 너무 힘들다.

크게 중요하지 않은 메시지는 범용성 있는 required 같은 메시지로 끝내고, 정말 중요한 메시지는 꼭 필요할 때 구체적으로 적어서 사용하는 방식이 더 효과적이다.

### 이제 우리 애플리케이션에 이런 오류 코드 전략을 도입해보자

우선 다음처럼 만들어보자.

errors.properties

```
#required.item.itemName=상품 이름은 필수입니다.
#range.item.price=가격은 {0} ~ {1} 까지 허용합니다.
#max.item.quantity=수량은 최대 {0} 까지 허용합니다.
#totalPriceMin=가격 * 수량의 합은 {0}원 이상이어야 합니다. 현재 값 = {1}

#=ObjectError=
#Level1
totalPriceMin.item=상품의 가격 * 수량의 합은 {0}원 이상이어야 합니다. 현재 값 = {1}

#Level2 - 생략
totalPriceMin=전체 가격은 {0}원 이상이어야 합니다. 현재 값 = {1}

#=FieldError=
#Level1
required.item.itemName=상품 이름은 필수입니다.
range.item.price=가격은 {0} ~ {1} 까지 허용합니다.
max.item.quantity=수량은 최대 {0} 까지 허용합니다.
```

```
#Level2 - 생략
```

```
#Level3
```

```
required.java.lang.String = 필수 문자입니다.
required.java.lang.Integer = 필수 숫자입니다.
min.java.lang.String = {0} 이상의 문자를 입력해주세요.
min.java.lang.Integer = {0} 이상의 숫자를 입력해주세요.
range.java.lang.String = {0} ~ {1} 까지의 문자를 입력해주세요.
range.java.lang.Integer = {0} ~ {1} 까지의 숫자를 입력해주세요.
max.java.lang.String = {0} 까지의 숫자를 허용합니다.
max.java.lang.Integer = {0} 까지의 숫자를 허용합니다.
```

```
#Level4
```

```
required = 필수 값 입니다.
min= {0} 이상이어야 합니다.
range= {0} ~ {1} 범위를 허용합니다.
max= {0} 까지 허용합니다.
```

크게 객체 오류와 필드 오류를 나누었다. 그리고 범용성에 따라 레벨을 나누어두었다.

`itemName`의 경우 `required` 검증 오류 메시지가 발생하면 다음 코드 순서대로 메시지가 생성된다.

1. `required.item.itemName`
2. `required.itemName`
3. `required.java.lang.String`
4. `required`

그리고 이렇게 생성된 메시지 코드를 기반으로 순서대로 `MessageSource`에서 메시지에서 찾는다.

구체적인 것에서 덜 구체적인 순서대로 찾는다. 메시지에 1번이 없으면 2번을 찾고, 2번이 없으면 3번을 찾는다.

이렇게 되면 만약에 크게 중요하지 않은 오류 메시지는 기존에 정의된 것을 그냥 **재활용** 하면 된다!

## 실행

- Level1 전부 주석해보자
- Level2,3 전부 주석해보자
- Level4 전부 주석해보자 → 못찾으면 코드에 작성한 디폴트 메시지를 사용한다.
- Object 오류도 Level1, Level2로 재활용 가능하다.

## ValidationUtils

### ValidationUtils 사용 전

```
if (!StringUtils.hasText(item.getItemName())) {
 bindingResult.rejectValue("itemName", "required", "기본: 상품 이름은
 필수입니다.");
}
```

### ValidationUtils 사용 후

다음과 같이 한줄로 가능, 제공하는 기능은 Empty, 공백 같은 단순한 기능만 제공

```
ValidationUtils.rejectIfEmptyOrWhitespace(bindingResult, "itemName",
 "required");
```

### 정리

1. rejectValue() 호출
2. MessageCodesResolver 를 사용해서 검증 오류 코드로 메시지 코드들을 생성
3. new FieldError() 를 생성하면서 메시지 코드들을 보관
4. th:errors 에서 메시지 코드들로 메시지를 순서대로 메시지에서 찾고, 노출

## 오류 코드와 메시지 처리6

스프링이 직접 만든 오류 메시지 처리

검증 오류 코드는 다음과 같이 2가지로 나눌 수 있다.

- 개발자가 직접 설정한 오류 코드 → rejectValue() 를 직접 호출
- 스프링이 직접 검증 오류에 추가한 경우(주로 타입 정보가 맞지 않음)

지금까지 학습한 메시지 코드 전략의 강점을 지금부터 확인해보자.

`price` 필드에 문자 "A"를 입력해보자.

로그를 확인해보면 `BindingResult`에 `FieldError`가 담겨있고, 다음과 같은 메시지 코드들이 생성된 것을 확인할 수 있다.

```
codes [typeMismatch.item.price, typeMismatch.price, typeMismatch.java.lang.Integer, typeMismatch]
```

다음과 같이 4가지 메시지 코드가 입력되어 있다.

- `typeMismatch.item.price`
- `typeMismatch.price`
- `typeMismatch.java.lang.Integer`
- `typeMismatch`

그렇다. 스프링은 타입 오류가 발생하면 `typeMismatch`라는 오류 코드를 사용한다. 이 오류 코드가 `MessageCodesResolver`를 통하여 4가지 메시지 코드가 생성된 것이다.

### 실행해보자.

아직 `errors.properties`에 메시지 코드가 없기 때문에 스프링이 생성한 기본 메시지가 출력된다.

```
Failed to convert property value of type java.lang.String to required type
java.lang.Integer for property price; nested exception is
java.lang.NumberFormatException: For input string: "A"
```

`error.properties`에 다음 내용을 추가하자

```
#추가
typeMismatch.java.lang.Integer=숫자를 입력해주세요.
typeMismatch=타입 오류입니다.
```

### 다시 실행해보자

결과적으로 소스코드를 하나도 건들지 않고, 원하는 메시지를 단계별로 설정할 수 있다.

### 정리

메시지 코드 생성 전략은 그냥 만들어진 것이 아니다. 조금 뒤에서 Bean Validation을 학습하면 그 진가를 더 확인할 수 있다.

## Validator 분리1

### 목표

- 복잡한 검증 로직을 별도로 분리하자.

컨트롤러에서 검증 로직이 차지하는 부분은 매우 크다. 이런 경우 별도의 클래스로 역할을 분리하는 것이 좋다. 그리고 이렇게 분리한 검증 로직을 재사용 할 수도 있다.

ItemValidator 를 만들자.

```
package hello.itemservice.web.validation;

import hello.itemservice.domain.item.Item;
import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

@Component
public class ItemValidator implements Validator {

 @Override
 public boolean supports(Class<?> clazz) {
 return Item.class.isAssignableFrom(clazz);
 }

 @Override
 public void validate(Object target, Errors errors) {

 Item item = (Item) target;

 ValidationUtils.rejectIfEmptyOrWhitespace(errors, "itemName",
 "required");

 if (item.getPrice() == null || item.getPrice() < 1000 || item.getPrice() > 1000000) {
 errors.rejectValue("price", "range", new Object[]{1000, 1000000},
 null);
 }
 }
}
```

```

 if (item.getQuantity() == null || item.getQuantity() > 10000) {
 errors.rejectValue("quantity", "max", new Object[]{9999}, null);
 }

 //특정 필드 예외가 아닌 전체 예외
 if (item.getPrice() != null && item.getQuantity() != null) {
 int resultPrice = item.getPrice() * item.getQuantity();
 if (resultPrice < 10000) {
 errors.reject("totalPriceMin", new Object[]{10000,
resultPrice}, null);
 }
 }
}

```

스프링은 검증을 체계적으로 제공하기 위해 다음 인터페이스를 제공한다.

```

public interface Validator {
 boolean supports(Class<?> clazz);
 void validate(Object target, Errors errors);
}

```

- `supports()` : 해당 검증기를 지원하는 여부 확인(뒤에서 설명)
- `validate(Object target, Errors errors)` : 검증 대상 객체와 `BindingResult`

### ItemValidator 직접 호출하기

#### ValidationItemControllerV2 - addItemV5()

```

private final ItemValidator itemValidator;

@PostMapping("/add")
public String addItemV5(@ModelAttribute Item item, BindingResult bindingResult,
RedirectAttributes redirectAttributes) {

 itemValidator.validate(item, bindingResult);
}

```

```

if (bindingResult.hasErrors()) {
 log.info("errors={}", bindingResult);
 return "validation/v2/addForm";
}

//성공 로직

Item savedItem = itemRepository.save(item);
redirectAttributes.addAttribute("itemId", savedItem.getId());
redirectAttributes.addAttribute("status", true);
return "redirect:/validation/v2/items/{itemId}";
}

```

### 코드 변경

- `addItemV4()` 의 `@PostMapping` 부분 주석 처리

`ItemValidator` 를 스프링 빈으로 주입 받아서 직접 호출했다.

### 실행

실행해보면 기존과 완전히 동일하게 동작하는 것을 확인할 수 있다. 검증과 관련된 부분이 깔끔하게 분리되었다.

## Validator 분리2

스프링이 `Validator` 인터페이스를 별도로 제공하는 이유는 체계적으로 검증 기능을 도입하기 위해서다. 그런데 앞에서는 검증기를 직접 불러서 사용했고, 이렇게 사용해도 된다. 그런데 `Validator` 인터페이스를 사용해서 검증기를 만들면 스프링의 추가적인 도움을 받을 수 있다.

### WebDataBinder를 통해서 사용하기

`WebDataBinder` 는 스프링의 파라미터 바인딩의 역할을 해주고 검증 기능도 내부에 포함한다.

### ValidationItemControllerV2에 다음 코드를 추가하자

```

@InitBinder
public void init(WebDataBinder dataBinder) {
 log.info("init binder {}", dataBinder);
}

```

```
 dataBinder.addValidators(itemValidator);
}
```

이렇게 `WebDataBinder`에 검증기를 추가하면 해당 컨트롤러에서는 검증기를 자동으로 적용할 수 있다.  
`@InitBinder` → 해당 컨트롤러에만 영향을 준다. 글로벌 설정은 별도로 해야한다. (마지막에 설명)

### @Validated 적용

#### ValidationItemControllerV2 - addItemV6()

```
@PostMapping("/add")
public String addItemV6(@Validated @ModelAttribute Item item, BindingResult
bindingResult, RedirectAttributes redirectAttributes) {

 if (bindingResult.hasErrors()) {
 log.info("errors={}", bindingResult);
 return "validation/v2/addForm";
 }

 //성공 로직
 Item savedItem = itemRepository.save(item);
 redirectAttributes.addAttribute("itemId", savedItem.getId());
 redirectAttributes.addAttribute("status", true);
 return "redirect:/validation/v2/items/{itemId}";
}
```

### 코드 변경

- `addItemV5()`의 `@PostMapping` 부분 주석 처리

validator를 직접 호출하는 부분이 사라지고, 대신에 검증 대상 앞에 `@Validated` 가 붙었다.

### 실행

기존과 동일하게 잘 동작하는 것을 확인할 수 있다.

### 동작 방식

`@Validated` 는 검증기를 실행하라는 애노테이션이다.

이 애노테이션이 붙으면 앞서 `WebDataBinder`에 등록한 검증기를 찾아서 실행한다. 그런데 여러 검증기를 등록한다면 그 중에 어떤 검증기가 실행되어야 할지 구분이 필요하다. 이때 `supports()` 가 사용된다.

여기서는 `supports(Item.class)` 호출되고, 결과가 `true`이므로 `ItemValidator`의 `validate()`가 호출된다.

```
@Component
public class ItemValidator implements Validator {

 @Override
 public boolean supports(Class<?> clazz) {
 return Item.class.isAssignableFrom(clazz);
 }

 @Override
 public void validate(Object target, Errors errors) {...}
}
```

## 글로벌 설정 - 모든 컨트롤러에 다 적용

```
@SpringBootApplication
public class ItemServiceApplication implements WebMvcConfigurer {

 public static void main(String[] args) {
 SpringApplication.run(ItemServiceApplication.class, args);
 }

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

이렇게 글로벌 설정을 추가할 수 있다. 기존 컨트롤러의 `@InitBinder`를 제거해도 글로벌 설정으로 정상 동작하는 것을 확인할 수 있다. 이어지는 다음 강의를 위해서 글로벌 설정은 꼭 제거해두자.

| 주의

글로벌 설정을 하면 다음에 설명할 BeanValidator가 자동 등록되지 않는다. 글로벌 설정 부분은 주석처리 해두자. 참고로 글로벌 설정을 직접 사용하는 경우는 드물다.

### 참고

검증시 `@Validated` `@Valid` 둘다 사용가능하다.

`javax.validation.Valid`를 사용하려면 `build.gradle` 의존관계 추가가 필요하다.

`implementation 'org.springframework.boot:spring-boot-starter-validation'`

`@Validated` 는 스프링 전용 검증 애노테이션이고, `@Valid` 는 자바 표준 검증 애노테이션이다.

자세한 내용은 다음 Bean Validation에서 설명하겠다.

## 정리

## 5. 검증2 - Bean Validation

#인강/5. 스프링 MVC 2/강의#

### 목차

- 5. 검증2 - Bean Validation - Bean Validation - 소개
- 5. 검증2 - Bean Validation - Bean Validation - 시작
- 5. 검증2 - Bean Validation - Bean Validation - 프로젝트 준비 V3
- 5. 검증2 - Bean Validation - Bean Validation - 스프링 적용
- 5. 검증2 - Bean Validation - Bean Validation - 에러 코드
- 5. 검증2 - Bean Validation - Bean Validation - 오브젝트 오류
- 5. 검증2 - Bean Validation - Bean Validation - 수정에 적용
- 5. 검증2 - Bean Validation - Bean Validation - 한계
- 5. 검증2 - Bean Validation - Bean Validation - groups
- 5. 검증2 - Bean Validation - Form 전송 객체 분리 - 프로젝트 준비 V4
- 5. 검증2 - Bean Validation - Form 전송 객체 분리 - 소개
- 5. 검증2 - Bean Validation - Form 전송 객체 분리 - 개발
- 5. 검증2 - Bean Validation - Bean Validation - HTTP 메시지 컨버터
- 5. 검증2 - Bean Validation - 정리

### Bean Validation - 소개

검증 기능을 지금처럼 매번 코드로 작성하는 것은 상당히 번거롭다. 특히 특정 필드에 대한 검증 로직은 대부분 빈 값인지 아닌지, 특정 크기를 넘는지 아닌지와 같이 매우 일반적인 로직이다. 다음 코드를 보자.

```
public class Item {

 private Long id;

 @NotBlank
 private String itemName;

 @NotNull
 @Range(min = 1000, max = 1000000)
 private Integer price;
```

```
@NotNull
@Max(9999)
private Integer quantity;
// ...
}
```

이런 검증 로직을 모든 프로젝트에 적용할 수 있게 공통화하고, 표준화 한 것이 바로 Bean Validation 이다.

Bean Validation을 잘 활용하면, 애노테이션 하나로 검증 로직을 매우 편리하게 적용할 수 있다.

### Bean Validation 이란?

먼저 Bean Validation은 특정한 구현체가 아니라 Bean Validation 2.0(JSR-380)이라는 기술 표준이다. 쉽게 이야기해서 검증 애노테이션과 여러 인터페이스의 모음이다. 마치 JPA가 표준 기술이고 그 구현체로 하이버네이트가 있는 것과 같다.

Bean Validation을 구현한 기술중에 일반적으로 사용하는 구현체는 하이버네이트 Validator이다. 이름이 하이버네이트가 붙어서 그렇지 ORM과는 관련이 없다.

### 하이버네이트 Validator 관련 링크

- 공식 사이트: <http://hibernate.org/validator/>
- 공식 메뉴얼: [https://docs.jboss.org/hibernate/validator/6.2/reference/en-US/html\\_single/](https://docs.jboss.org/hibernate/validator/6.2/reference/en-US/html_single/)
- 검증 애노테이션 모음: [https://docs.jboss.org/hibernate/validator/6.2/reference/en-US/html\\_single/#validator-defineconstraints-spec](https://docs.jboss.org/hibernate/validator/6.2/reference/en-US/html_single/#validator-defineconstraints-spec)

## Bean Validation - 시작

Bean Validation 기능을 어떻게 사용하는지 코드로 알아보자. 먼저 스프링과 통합하지 않고, 순수한 Bean Validation 사용법 부터 테스트 코드로 알아보자.

### Bean Validation 의존관계 추가

#### 의존관계 추가

Bean Validation을 사용하려면 다음 의존관계를 추가해야 한다.

build.gradle

```
implementation 'org.springframework.boot:spring-boot-starter-validation'
```

spring-boot-starter-validation 의존관계를 추가하면 라이브러리가 추가 된다.

## Jakarta Bean Validation

jakarta.validation-api : Bean Validation 인터페이스

hibernate-validator 구현체

테스트 코드 작성

## Item - Bean Validation 애노테이션 적용

```
package hello.itemservice.domain.item;

import lombok.Data;
import org.hibernate.validator.constraints.Range;

import javax.validation.constraints.Max;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;

@Data
public class Item {

 private Long id;

 @NotBlank
 private String itemName;

 @NotNull
 @Range(min = 1000, max = 1000000)
 private Integer price;

 @NotNull
 @Max(9999)
```

```

private Integer quantity;

public Item() {
}

public Item(String itemName, Integer price, Integer quantity) {
 this.itemName = itemName;
 this.price = price;
 this.quantity = quantity;
}

```

### 검증 애노테이션

@NotBlank : 빈값 + 공백만 있는 경우를 허용하지 않는다.

@NotNull : null을 허용하지 않는다.

@Range(min = 1000, max = 1000000) : 범위 안의 값이어야 한다.

@Max(9999) : 최대 9999까지만 허용한다.

### 참고

javax.validation.constraints.NotNull

org.hibernate.validator.constraints.Range

javax.validation 으로 시작하면 특정 구현에 관계없이 제공되는 표준 인터페이스이고,

org.hibernate.validator 로 시작하면 하이버네이트 validator 구현체를 사용할 때만 제공되는 검증 기능이다. 실무에서 대부분 하이버네이트 validator를 사용하므로 자유롭게 사용해도 된다.

### BeanValidationTest - Bean Validation 테스트 코드 작성

```

package hello.itemservice.validation;

import hello.itemservice.domain.item.Item;
import org.junit.jupiter.api.Test;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;

```

```

import javax.validation.ValidatorFactory;
import java.util.Set;

public class BeanValidationTest {

 @Test
 void beanValidation() {
 ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
 Validator validator = factory.getValidator();

 Item item = new Item();
 item.setItemName(" "); //공백
 item.setPrice(0);
 item.setQuantity(10000);

 Set<ConstraintViolation<Item>> violations = validator.validate(item);
 for (ConstraintViolation<Item> violation : violations) {
 System.out.println("violation=" + violation);
 System.out.println("violation.message=" + violation.getMessage());
 }
 }
}

```

## 검증기 생성

다음 코드와 같이 검증기를 생성한다. 이후 스프링과 통합하면 우리가 직접 이런 코드를 작성하지는 않으므로, 이렇게 사용하는구나 정도만 참고하자.

```

ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();

```

## 검증 실행

검증 대상(`item`)을 직접 검증기에 넣고 그 결과를 받는다. `Set`에는 `ConstraintViolation`이라는 검증 오류가 담긴다. 따라서 결과가 비어있으면 검증 오류가 없는 것이다.

```

Set<ConstraintViolation<Item>> violations = validator.validate(item);

```

실행 결과를 보자.

### 실행 결과 (일부 생략)

```
violation={interpolatedMessage='공백일 수 없습니다', propertyPath=itemName,
rootBeanClass=class hello.itemservice.domain.item.Item,
messageTemplate='{javax.validation.constraints.NotBlank.message}'}
violation.message=공백일 수 없습니다

violation={interpolatedMessage='9999 이하여야 합니다', propertyPath=quantity,
rootBeanClass=class hello.itemservice.domain.item.Item,
messageTemplate='{javax.validation.constraints.Max.message}'}
violation.message=9999 이하여야 합니다

violation={interpolatedMessage='1000에서 1000000 사이여야 합니다',
propertyPath=price, rootBeanClass=class hello.itemservice.domain.item.Item,
messageTemplate='{org.hibernate.validator.constraints.Range.message}'}
violation.message=1000에서 1000000 사이여야 합니다
```

**ConstraintViolation** 출력 결과를 보면, 검증 오류가 발생한 객체, 필드, 메시지 정보등 다양한 정보를 확인할 수 있다.

### 정리

이렇게 빈 검증기(Bean Validation)를 직접 사용하는 방법을 알아보았다. 아마 지금까지 배웠던 스프링 MVC 검증 방법에 빈 검증기를 어떻게 적용하면 좋을지 여러가지 생각이 들 것이다. 스프링은 이미 개발자를 위해 빈 검증기를 스프링에 완전히 통합해두었다.

## Bean Validation - 프로젝트 준비 V3

앞서 만든 기능을 유지하기 위해, 컨트롤러와 템플릿 파일을 복사하자.

### ValidationItemControllerV3 컨트롤러 생성

- hello.itemservice.web.validation.ValidationItemControllerV2 복사

- hello.itemservice.web.validation.ValidationItemControllerV3 붙여넣기
- URL 경로 변경: validation/v2/ → validation/v3/

### 템플릿 파일 복사

validation/v2 디렉토리의 모든 템플릿 파일을 validation/v3 디렉토리로 복사

- /resources/templates/validation/v2/ → /resources/templates/validation/v3/
  - addForm.html
  - editForm.html
  - item.html
  - items.html
- /resources/templates/validation/v3/ 하위 4개 파일 모두 URL 경로 변경: validation/v2/ → validation/v3/
  - addForm.html
  - editForm.html
  - item.html
  - items.html

### 실행

<http://localhost:8080/validation/v3/items>

실행 후 웹 브라우저의 URL이 validation/v3 으로 잘 유지되는지 확인해자.

## Bean Validation - 스프링 적용

### ValidationItemControllerV3 코드 수정

```
package hello.itemservice.web.validation;

import hello.itemservice.domain.item.Item;
import hello.itemservice.domain.item.ItemRepository;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
```

```
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import java.util.List;

@Slf4j
@Controller
@RequestMapping("/validation/v3/items")
@RequiredArgsConstructor
public class ValidationItemControllerV3 {

 private final ItemRepository itemRepository;

 @GetMapping
 public String items(Model model) {
 List<Item> items = itemRepository.findAll();
 model.addAttribute("items", items);
 return "validation/v3/items";
 }

 @GetMapping("/{itemId}")
 public String item(@PathVariable long itemId, Model model) {
 Item item = itemRepository.findById(itemId);
 model.addAttribute("item", item);
 return "validation/v3/item";
 }

 @GetMapping("/add")
 public String addForm(Model model) {
 model.addAttribute("item", new Item());
 return "validation/v3/addForm";
 }

 @PostMapping("/add")
 public String addItem(@Validated @ModelAttribute Item item, BindingResult
bindingResult, RedirectAttributes redirectAttributes) {

 if (bindingResult.hasErrors()) {
 log.info("errors={}", bindingResult);
```

```

 return "validation/v3/addForm";
 }

 //성공 로직
 Item savedItem = itemRepository.save(item);
 redirectAttributes.addAttribute("itemId", savedItem.getId());
 redirectAttributes.addAttribute("status", true);
 return "redirect:/validation/v3/items/{itemId}";
}

@GetMapping("/{itemId}/edit")
public String editForm(@PathVariable Long itemId, Model model) {
 Item item = itemRepository.findById(itemId);
 model.addAttribute("item", item);
 return "validation/v3/editForm";
}

@PostMapping("/{itemId}/edit")
public String edit(@PathVariable Long itemId, @ModelAttribute Item item) {
 itemRepository.update(itemId, item);
 return "redirect:/validation/v3/items/{itemId}";
}

}

```

제거: addItemV1() ~ addItemV5()

변경: addItemV6() → addItem()

## 코드 제거

기존에 등록한 **ItemValidator**를 제거해두자! 오류 검증기가 중복 적용된다.

```

private final ItemValidator itemValidator;

@InitBinder
public void init(WebDataBinder dataBinder) {
 log.info("init binder {}", dataBinder);
 dataBinder.addValidators(itemValidator);
}

```

## 실행

<http://localhost:8080/validation/v3/items>

실행해보면 애노테이션 기반의 Bean Validation이 정상 동작하는 것을 확인할 수 있다.

## 참고

특정 필드의 범위를 넘어서는 검증(가격 \* 수량의 합은 10,000원 이상) 기능이 빠졌는데, 이 부분은 조금 뒤에 설명한다.

## 스프링 MVC는 어떻게 Bean Validator를 사용?

스프링 부트가 `spring-boot-starter-validation` 라이브러리를 넣으면 자동으로 Bean Validator를 인지하고 스프링에 통합한다.

## 스프링 부트는 자동으로 글로벌 Validator로 등록한다.

`LocalValidatorFactoryBean` 을 글로벌 Validator로 등록한다. 이 Validator는 `@NotNull` 같은 애노테이션을 보고 검증을 수행한다. 이렇게 글로벌 Validator가 적용되어 있기 때문에, `@Valid`, `@Validated` 만 적용하면 된다.

검증 오류가 발생하면, `FieldError`, `ObjectError` 를 생성해서 `BindingResult` 에 담아준다.

## 주의!

다음과 같이 직접 글로벌 Validator를 직접 등록하면 스프링 부트는 Bean Validator를 글로벌 Validator로 등록하지 않는다. 따라서 애노테이션 기반의 빈 검증기가 동작하지 않는다. 다음 부분은 제거하자.

```
@SpringBootApplication
public class ItemServiceApplication implements WebMvcConfigurer {

 // 글로벌 검증기 추가
 @Override
 public Validator getValidator() {
 return new ItemValidator();
 }
 // ...
}
```

## 참고

검증시 `@Validated` `@Valid` 둘다 사용가능하다.

`javax.validation.Valid` 를 사용하려면 `build.gradle` 의존관계 추가가 필요하다. (이전에 추가했다.)

```
implementation 'org.springframework.boot:spring-boot-starter-validation'
```

`@Validated` 는 스프링 전용 검증 애노테이션이고, `@Valid` 는 자바 표준 검증 애노테이션이다. 둘중 아무거나 사용해도 동일하게 작동하지만, `@Validated` 는 내부에 `groups` 라는 기능을 포함하고 있다. 이 부분은 조금 뒤에 다시 설명하겠다.

## 검증 순서

1. `@ModelAttribute` 각각의 필드에 타입 변환 시도

1. 성공하면 다음으로
2. 실패하면 `typeMismatch` 로 `FieldError` 추가

2. Validator 적용

### 바인딩에 성공한 필드만 Bean Validation 적용

BeanValidator는 바인딩에 실패한 필드는 BeanValidation을 적용하지 않는다.

생각해보면 타입 변환에 성공해서 바인딩에 성공한 필드여야 BeanValidation 적용이 의미 있다.

(일단 모델 객체에 바인딩 받는 값이 정상으로 들어와야 검증도 의미가 있다.)

`@ModelAttribute` → 각각의 필드 타입 변환시도 → 변환에 성공한 필드만 BeanValidation 적용

예)

- `itemName`에 문자 "A" 입력 → 타입 변환 성공 → `itemName` 필드에 BeanValidation 적용
- `price`에 문자 "A" 입력 → "A"를 숫자 타입 변환 시도 실패 → `typeMismatch FieldError` 추가 → `price` 필드는 BeanValidation 적용 X

## Bean Validation - 에러 코드

Bean Validation이 기본으로 제공하는 오류 메시지를 좀 더 자세히 변경하고 싶으면 어떻게 하면 될까?

Bean Validation을 적용하고 `bindingResult`에 등록된 검증 오류 코드를 보자.

오류 코드가 애노테이션 이름으로 등록된다. 마치 `typeMismatch` 와 유사하다.

`NotBlank`라는 오류 코드를 기반으로 `MessageCodesResolver`를 통해 다양한 메시지 코드가 순서대로

생성된다.

### @NotBlank

- NotBlank.item.itemName
- NotBlank.itemName
- NotBlank.java.lang.String
- NotBlank

### @Range

- Range.item.price
- Range.price
- Range.java.lang.Integer
- Range

### 메시지 등록

이제 메시지를 등록해보자.

#### errors.properties

```
#Bean Validation 추가
NotBlank={0} 공백X
Range={0}, {2} ~ {1} 허용
Max={0}, 최대 {1}
```

{0} 은 필드명이고, {1}, {2} ...은 각 애노테이션마다 다르다.

### 실행

실행해보면 방금 등록한 메시지가 정상 적용되는 것을 확인할 수 있다.

### BeanValidation 메시지 찾는 순서

1. 생성된 메시지 코드 순서대로 messageSource에서 메시지 찾기
2. 애노테이션의 message 속성 사용 → @NotBlank(message = "공백! {0}")
3. 라이브러리가 제공하는 기본 값 사용 → 공백일 수 없습니다.

### 애노테이션의 message 사용 예

```
@NotBlank(message = "공백은 입력할 수 없습니다.")
private String itemName;
```

## Bean Validation - 오브젝트 오류

Bean Validation에서 특정 필드( `FieldError` )가 아닌 해당 오브젝트 관련 오류( `ObjectError` )는 어떻게 처리할 수 있을까?

다음과 같이 `@ScriptAssert()` 를 사용하면 된다.

```
@Data
@ScriptAssert(lang = "javascript", script = "_this.price * _this.quantity >= 10000")
public class Item {
 //...
}
```

실행해보면 정상 수행되는 것을 확인할 수 있다. 메시지 코드도 다음과 같이 생성된다.

### 메시지 코드

- `ScriptAssert.item`
- `ScriptAssert`

그런데 실제 사용해보면 제약이 많고 복잡하다. 그리고 실무에서는 검증 기능이 해당 객체의 범위를 넘어서는 경우들도 종종 등장하는데, 그런 경우 대응이 어렵다.

따라서 오브젝트 오류(글로벌 오류)의 경우 `@ScriptAssert` 을 얹지로 사용하는 것 보다는 다음과 같이 오브젝트 오류 관련 부분만 직접 자바 코드로 작성하는 것을 권장한다.

## ValidationItemControllerV3 - 글로벌 오류 추가

```
@PostMapping("/add")
public String addItem(@Validated @ModelAttribute Item item, BindingResult bindingResult, RedirectAttributes redirectAttributes) {

 //특정 필드 예외가 아닌 전체 예외
```

```

 if (item.getPrice() != null && item.getQuantity() != null) {
 int resultPrice = item.getPrice() * item.getQuantity();
 if (resultPrice < 10000) {
 bindingResult.reject("totalPriceMin", new Object[]{10000,
resultPrice}, null);
 }
 }

 if (bindingResult.hasErrors()) {
 log.info("errors={}", bindingResult);
 return "validation/v3/addForm";
 }

 //성공 로직
 Item savedItem = itemRepository.save(item);
 redirectAttributes.addAttribute("itemId", savedItem.getId());
 redirectAttributes.addAttribute("status", true);
 return "redirect:/validation/v3/items/{itemId}";
}

```

**@ScriptAssert** 부분 제거

## Bean Validation - 수정에 적용

상품 수정에도 빈 검증(Bean Validation)을 적용해보자.

수정에도 검증 기능을 추가하자

**ValidationItemControllerV3 - edit()** 변경

```

@PostMapping("/{itemId}/edit")
public String edit(@PathVariable Long itemId, @Validated @ModelAttribute Item
item, BindingResult bindingResult) {

 //특정 필드 예외가 아닌 전체 예외
 if (item.getPrice() != null && item.getQuantity() != null) {
 int resultPrice = item.getPrice() * item.getQuantity();
 }
}

```

```

 if (resultPrice < 10000) {
 bindingResult.reject("totalPriceMin", new Object[]{10000,
resultPrice}, null);
 }
 }

 if (bindingResult.hasErrors()) {
 log.info("errors={}", bindingResult);
 return "validation/v3/editForm";
 }

 itemRepository.update(itemId, item);
 return "redirect:/validation/v3/items/{itemId}";
}

```

- `edit()` : Item 모델 객체에 `@Validated` 를 추가하자.
- 검증 오류가 발생하면 `editForm` 으로 이동하는 코드 추가

#### validation/v3/editForm.html 변경

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
 <link th:href="@{/css/bootstrap.min.css}"
 href="../css/bootstrap.min.css" rel="stylesheet">
<style>
 .container {
 max-width: 560px;
 }
 .field-error {
 border-color: #dc3545;
 color: #dc3545;
 }
</style>
</head>
<body>

```

```
<div class="container">

 <div class="py-5 text-center">
 <h2 th:text="#{page.updateItem}">상품 수정</h2>
 </div>

 <form action="item.html" th:action th:object="${item}" method="post">

 <div th:if="#{fields.hasGlobalErrors()}">
 <p class="field-error" th:each="err : ${#fields.globalErrors()}" th:text="${err}">글로벌 오류 메시지</p>
 </div>

 <div>
 <label for="id" th:text="#{label.item.id}">상품 ID</label>
 <input type="text" id="id" th:field="*{id}" class="form-control" readonly>
 </div>

 <div>
 <label for="itemName" th:text="#{label.item.itemName}">상품명</label>
 <input type="text" id="itemName" th:field="*{itemName}" th:errorclass="field-error" class="form-control" placeholder="이름을 입력하세요">
 <div class="field-error" th:errors="*{itemName}">
 상품명 오류
 </div>
 </div>

 <div>
 <label for="price" th:text="#{label.item.price}">가격</label>
 <input type="text" id="price" th:field="*{price}" th:errorclass="field-error" class="form-control" placeholder="가격을 입력하세요">
 <div class="field-error" th:errors="*{price}">
 가격 오류
 </div>
 </div>

 <div>
 </div>
```

```

 <label for="quantity" th:text="#{label.item.quantity}">수량</label>
 <input type="text" id="quantity" th:field="*{quantity}"
 th:errorclass="field-error" class="form-control"
 placeholder="수량을 입력하세요">
 <div class="field-error" th:errors="*{quantity}">
 수량 오류
 </div>
 </div>

 <hr class="my-4">

 <div class="row">
 <div class="col">
 <button class="w-100 btn btn-primary btn-lg" type="submit"
th:text="#{button.save}">저장</button>
 </div>
 <div class="col">
 <button class="w-100 btn btn-secondary btn-lg"
 onclick="location.href='item.html'"
 th:onclick="|location.href=@{/validation/v3/items/
{itemId}(itemId=${item.id})}|"
 type="button" th:text="#{button.cancel}">취소</button>
 </div>
 </div>
</form>

</div> <!-- /container -->
</body>
</html>

```

- `.field-error` css 추가
- 글로벌 오류 메시지
- 상품명, 가격, 수량 필드에 검증 기능 추가

## 실행

<http://localhost:8080/validation/v3/items>

## Bean Validation - 한계

### 수정시 검증 요구사항

데이터를 등록할 때와 수정할 때는 요구사항이 다를 수 있다.

#### 등록시 기준 요구사항

- 타입 검증
  - 가격, 수량에 문자가 들어가면 검증 오류 처리
- 필드 검증
  - 상품명: 필수, 공백X
  - 가격: 1000원 이상, 1백만원 이하
  - 수량: 최대 9999
- 특정 필드의 범위를 넘어서는 검증
  - 가격 \* 수량의 합은 10,000원 이상

#### 수정시 요구사항

- 등록시에는 `quantity` 수량을 최대 9999까지 등록할 수 있지만 수정시에는 수량을 무제한으로 변경할 수 있다.
- 등록시에는 `id`에 값이 없어도 되지만, 수정시에는 `id` 값이 필수이다.

#### 수정 요구사항 적용

수정시에는 `Item`에서 `id` 값이 필수이고, `quantity`도 무제한으로 적용할 수 있다.

```
package hello.itemservice.domain.item;

@Data
public class Item {

 @NotNull //수정 요구사항 추가
 private Long id;

 @NotBlank
 private String itemName;

 @NotNull
 @Range(min = 1000, max = 1000000)
```

```

private Integer price;

@NotNull
//@Max(9999) //수정 요구사항 추가

private Integer quantity;

// ...
}

```

수정 요구사항을 적용하기 위해 다음을 적용했다.

`id : @NotNull` 추가  
`quantity : @Max(9999)` 제거

### 참고

현재 구조에서는 수정시 `item`의 `id` 값은 항상 들어있도록 로직이 구성되어 있다. 그래서 검증하지 않아도 된다고 생각할 수 있다. 그런데 HTTP 요청은 언제든지 악의적으로 변경해서 요청할 수 있으므로 서버에서 항상 검증해야 한다. 예를 들어서 HTTP 요청을 변경해서 `item`의 `id` 값을 삭제하고 요청할 수도 있다. 따라서 최종 검증은 서버에서 진행하는 것이 안전한다.

### 수정을 실행해보자.

정상 동작을 확인할 수 있다.

그런데 수정은 잘 동작하지만 등록에서 문제가 발생한다.

등록시에는 `id`에 값도 없고, `quantity` 수량 제한 최대 값인 9999도 적용되지 않는 문제가 발생한다.

등록시 화면이 넘어가지 않으면서 다음과 같은 오류를 볼 수 있다.

`'id': rejected value [null];`

왜냐하면 등록시에는 `id`에 값이 없다. 따라서 `@NotNull` `id`를 적용한 것 때문에 검증에 실패하고 다시 폼 화면으로 넘어온다. 결국 등록 자체도 불가능하고, 수량 제한도 걸지 못한다.

결과적으로 `item`은 등록과 수정에서 검증 조건의 충돌이 발생하고, 등록과 수정은 같은 BeanValidation 을 적용할 수 없다. 이 문제를 어떻게 해결할 수 있을까?

## Bean Validation - groups

동일한 모델 객체를 등록할 때와 수정할 때 각각 다르게 검증하는 방법을 알아보자.

## 방법 2가지

- BeanValidation의 groups 기능을 사용한다.
- Item을 직접 사용하지 않고, ItemSaveForm, ItemUpdateForm 같은 폼 전송을 위한 별도의 모델 객체를 만들어서 사용한다.

### BeanValidation groups 기능 사용

이런 문제를 해결하기 위해 Bean Validation은 groups라는 기능을 제공한다.

예를 들어서 등록시에 검증할 기능과 수정시에 검증할 기능을 각각 그룹으로 나누어 적용할 수 있다.

코드로 확인해보자.

#### groups 적용

##### 저장용 groups 생성

```
package hello.itemservice.domain.item;

public interface SaveCheck {
```

```
}
```

##### 수정용 groups 생성

```
package hello.itemservice.domain.item;

public interface UpdateCheck {
```

```
}
```

#### Item - groups 적용

```
package hello.itemservice.domain.item;

import lombok.Data;
import org.hibernate.validator.constraints.Range;

import javax.validation.constraints.Max;
```

```

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;

@Data
public class Item {

 @NotNull(groups = UpdateCheck.class) //수정시에만 적용
 private Long id;

 @NotBlank(groups = {SaveCheck.class, UpdateCheck.class})
 private String itemName;

 @NotNull(groups = {SaveCheck.class, UpdateCheck.class})
 @Range(min = 1000, max = 1000000, groups = {SaveCheck.class,
 UpdateCheck.class})
 private Integer price;

 @NotNull(groups = {SaveCheck.class, UpdateCheck.class})
 @Max(value = 9999, groups = SaveCheck.class) //등록시에만 적용
 private Integer quantity;

 public Item() {
 }

 public Item(String itemName, Integer price, Integer quantity) {
 this.itemName = itemName;
 this.price = price;
 this.quantity = quantity;
 }
}

```

### ValidationItemControllerV3 - 저장 로직에 SaveCheck Groups 적용

```

@PostMapping("/add")
public String addItemV2(@Validated(SaveCheck.class) @ModelAttribute Item item,
BindingResult bindingResult, RedirectAttributes redirectAttributes) {
 //...
}

```

- `addItem()` 를 복사해서 `addItemV2()` 생성, `SaveCheck.class` 적용
- 기존 `addItem()` `@PostMapping("/add")` 주석처리

### ValidationItemControllerV3 - 수정 로직에 UpdateCheck Groups 적용

```
@PostMapping("/{itemId}/edit")
public String editV2(@PathVariable Long itemId, @Validated(UpdateCheck.class)
@ModelAttribute Item item, BindingResult bindingResult) {
 // ...
}
```

- `edit()` 를 복사해서 `editV2()` 생성, `UpdateCheck.class` 적용
- 기존 `edit()` `@PostMapping("/{itemId}/edit")` 주석처리

| 참고: `@Valid`에는 groups를 적용할 수 있는 기능이 없다. 따라서 groups를 사용하려면 `@Validated` 를 사용해야 한다.

#### 실행

<http://localhost:8080/validation/v3/items>

#### 정리

groups 기능을 사용해서 등록과 수정시에 각각 다르게 검증을 할 수 있었다. 그런데 groups 기능을 사용하니 `Item`은 물론이고, 전반적으로 복잡도가 올라갔다.

사실 groups 기능은 실제 잘 사용되지는 않는데, 그 이유는 실무에서는 주로 다음에 등장하는 등록용 품 객체와 수정용 품 객체를 분리해서 사용하기 때문이다.

## Form 전송 객체 분리 - 프로젝트 준비 V4

앞서 만든 기능을 유지하기 위해, 컨트롤러와 템플릿 파일을 복사하자.

### ValidationItemControllerV4 컨트롤러 생성

- `hello.itemservice.web.validation.ValidationItemControllerV3` 복사

- `hello.itemservice.web.validation.ValidationItemControllerV4` 붙여넣기
- URL 경로 변경: `validation/v3/` → `validation/v4/`

### 템플릿 파일 복사

- `validation/v3` 디렉토리의 모든 템플릿 파일을 `validation/v4` 디렉토리로 복사
- `/resources/templates/validation/v3/` → `/resources/templates/validation/v4/`
    - `addForm.html`
    - `editForm.html`
    - `item.html`
    - `items.html`
  - `/resources/templates/validation/v4/` 하위 4개 파일 모두 URL 경로 변경: `validation/v3/` → `validation/v4/`
    - `addForm.html`
    - `editForm.html`
    - `item.html`
    - `items.html`

### 실행

`http://localhost:8080/validation/v4/items`

실행 후 웹 브라우저의 URL이 `validation/v4`으로 잘 유지되는지 확인해자.

## Form 전송 객체 분리 - 소개

`ValidationItemV4Controller`

실무에서는 `groups` 를 잘 사용하지 않는데, 그 이유가 다른 곳에 있다. 바로 등록시 폼에서 전달하는 데이터가 `Item` 도메인 객체와 딱 맞지 않기 때문이다.

소위 "Hello World" 예제에서는 폼에서 전달하는 데이터와 `Item` 도메인 객체가 딱 맞는다. 하지만 실무에서는 회원 등록시 회원과 관련된 데이터만 전달받는 것이 아니라, 약관 정보도 추가로 받는 등 `Item` 과 관계없는 수 많은 부가 데이터가 넘어온다.

그래서 보통 `Item` 을 직접 전달받는 것이 아니라, 복잡한 폼의 데이터를 컨트롤러까지 전달할 별도의 객체를 만들어서 전달한다. 예를 들면 `ItemSaveForm` 이라는 폼을 전달받는 전용 객체를 만들어서 `@ModelAttribute` 로 사용한다. 이것을 통해 컨트롤러에서 폼 데이터를 전달 받고, 이후 컨트롤러에서 필요한 데이터를 사용해서 `Item` 을 생성한다.

## 폼 데이터 전달에 Item 도메인 객체 사용

- HTML Form -> Item -> Controller -> Item -> Repository
  - 장점: Item 도메인 객체를 컨트롤러, 리포지토리 까지 직접 전달해서 중간에 Item을 만드는 과정이 없어서 간단하다.
  - 단점: 간단한 경우에만 적용할 수 있다. 수정시 검증이 중복될 수 있고, groups를 사용해야 한다.

## 폼 데이터 전달을 위한 별도의 객체 사용

- HTML Form -> ItemSaveForm -> Controller -> Item 생성 -> Repository
  - 장점: 전송하는 폼 데이터가 복잡해도 거기에 맞춘 별도의 폼 객체를 사용해서 데이터를 전달 받을 수 있다. 보통 등록과, 수정용으로 별도의 폼 객체를 만들기 때문에 검증이 중복되지 않는다.
  - 단점: 폼 데이터를 기반으로 컨트롤러에서 Item 객체를 생성하는 변환 과정이 추가된다.

수정의 경우 등록과 수정은 완전히 다른 데이터가 넘어온다. 생각해보면 회원 가입시 다루는 데이터와 수정시 다루는 데이터는 범위에 차이가 있다. 예를 들면 등록시에는 로그인 id, 주민번호 등을 받을 수 있지만, 수정시에는 이런 부분이 빠진다. 그리고 검증 로직도 많이 달라진다. 그래서 ItemUpdateForm이라는 별도의 객체로 데이터를 전달받는 것이 좋다.

Item 도메인 객체를 폼 전달 데이터로 사용하고, 그대로 쭉 넘기면 편리하겠지만, 앞에서 설명한 것과 같이 실무에서는 Item의 데이터만 넘어오는 것이 아니라 무수한 추가 데이터가 넘어온다. 그리고 더 나아가서 Item을 생성하는데 필요한 추가 데이터를 데이터베이스나 다른 곳에서 찾아와야 할 수도 있다.

따라서 이렇게 폼 데이터 전달을 위한 별도의 객체를 사용하고, 등록, 수정용 폼 객체를 나누면 등록, 수정이 완전히 분리되기 때문에 groups를 적용할 일은 드물다.

### Q: 이름은 어떻게 지어야 하나요?

이름은 의미있게 지으면 된다. ItemSave라고 해도 되고, ItemSaveForm, ItemSaveRequest, ItemSaveDto 등으로 사용해도 된다. 중요한 것은 일관성이다.

### Q: 등록, 수정용 뷰 템플릿이 비슷한데 합치는게 좋을까요?

한 페이지에 그러니까 뷰 템플릿 파일을 등록과 수정을 합치는게 좋을지 고민이 될 수 있다. 각각 장단점이 있으므로 고민하는게 좋지만, 어설프게 합치면 수 많은 분기분(등록일 때, 수정일 때) 때문에 나중에 유지보수에서 고통을 맛본다.

이런 어설픈 분기분들이 보이기 시작하면 분리해야 할 신호이다.

## Form 전송 객체 분리 - 개발

## ITEM 원복

이제 `Item`의 검증은 사용하지 않으므로 검증 코드를 제거해도 된다.

```
@Data
public class Item {

 private Long id;
 private String itemName;
 private Integer price;
 private Integer quantity;

}
```

## ItemSaveForm - ITEM 저장용 폼

```
package hello.itemservice.web.validation.form;

import lombok.Data;
import org.hibernate.validator.constraints.Range;

import javax.validation.constraints.Max;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;

@Data
public class ItemSaveForm {

 @NotBlank
 private String itemName;

 @NotNull
 @Range(min = 1000, max = 1000000)
 private Integer price;

 @NotNull
 @Max(value = 9999)
 private Integer quantity;
```

```
}
```

## ItemUpdateForm - ITEM 수정용 폼

```
package hello.itemservice.web.validation.form;

import lombok.Data;
import org.hibernate.validator.constraints.Range;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;

@Data
public class ItemUpdateForm {

 @NotNull
 private Long id;

 @NotBlank
 private String itemName;

 @NotNull
 @Range(min = 1000, max = 1000000)
 private Integer price;

 //수정에서는 수량은 자유롭게 변경할 수 있다.
 private Integer quantity;

}
```

이제 등록, 수정용 폼 객체를 사용하도록 컨트롤러를 수정하자.

## ValidationItemControllerV4

```
package hello.itemservice.web.validation;

import hello.itemservice.domain.item.Item;
```

```
import hello.itemservice.domain.item.ItemRepository;
import hello.itemservice.web.validation.form.ItemSaveForm;
import hello.itemservice.web.validation.form.ItemUpdateForm;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import java.util.List;

@Slf4j
@Controller
@RequestMapping("/validation/v4/items")
@RequiredArgsConstructor
public class ValidationItemControllerV4 {

 private final ItemRepository itemRepository;

 @GetMapping
 public String items(Model model) {
 List<Item> items = itemRepository.findAll();
 model.addAttribute("items", items);
 return "validation/v4/items";
 }

 @GetMapping("/{itemId}")
 public String item(@PathVariable long itemId, Model model) {
 Item item = itemRepository.findById(itemId);
 model.addAttribute("item", item);
 return "validation/v4/item";
 }

 @GetMapping("/add")
 public String addForm(Model model) {
 model.addAttribute("item", new Item());
 }
```

```

 return "validation/v4/addForm";
 }

 @PostMapping("/add")
 public String addItem(@Validated @ModelAttribute("item") ItemSaveForm form,
 BindingResult bindingResult, RedirectAttributes redirectAttributes) {

 //특정 필드 예외가 아닌 전체 예외

 if (form.getPrice() != null && form.getQuantity() != null) {
 int resultPrice = form.getPrice() * form.getQuantity();
 if (resultPrice < 10000) {
 bindingResult.reject("totalPriceMin", new Object[]{10000,
resultPrice}, null);
 }
 }

 if (bindingResult.hasErrors()) {
 log.info("errors={}", bindingResult);
 return "validation/v4/addForm";
 }

 //성공 로직

 Item item = new Item();
 item.setItemName(form.getItemName());
 item.setPrice(form.getPrice());
 item.setQuantity(form.getQuantity());

 Item savedItem = itemRepository.save(item);
 redirectAttributes.addAttribute("itemId", savedItem.getId());
 redirectAttributes.addAttribute("status", true);
 return "redirect:/validation/v4/items/{itemId}";
 }

 @GetMapping("/{itemId}/edit")
 public String editForm(@PathVariable Long itemId, Model model) {
 Item item = itemRepository.findById(itemId);
 model.addAttribute("item", item);
 return "validation/v4/editForm";
 }
}

```

```

 @PostMapping("/{itemId}/edit")
 public String edit(@PathVariable Long itemId, @Validated
 @ModelAttribute("item") ItemUpdateForm form, BindingResult bindingResult) {

 //특정 필드 예외가 아닌 전체 예외

 if (form.getPrice() != null && form.getQuantity() != null) {
 int resultPrice = form.getPrice() * form.getQuantity();
 if (resultPrice < 10000) {
 bindingResult.reject("totalPriceMin", new Object[]{10000,
resultPrice}, null);
 }
 }

 if (bindingResult.hasErrors()) {
 log.info("errors={}", bindingResult);
 return "validation/v4/editForm";
 }

 Item itemParam = new Item();
 itemParam.setItemName(form.getItemName());
 itemParam.setPrice(form.getPrice());
 itemParam.setQuantity(form.getQuantity());

 itemRepository.update(itemId, itemParam);
 return "redirect:/validation/v4/items/{itemId}";
 }

}

```

- 기존 코드 제거: `addItem()`, `addItemV2()`
- 기존 코드 제거: `edit()`, `editV2()`
- 추가: `addItem()`, `edit()`

## 폼 객체 바인딩

```

 @PostMapping("/add")
 public String addItem(@Validated @ModelAttribute("item") ItemSaveForm form,

```

```
BindingResult bindingResult, RedirectAttributes redirectAttributes) {
 //...
}
```

Item 대신에 ItemSaveform 을 전달 받는다. 그리고 @Validated 로 검증도 수행하고, BindingResult 로 검증 결과도 받는다.

## 주의

@ModelAttribute("item") 에 item 이름을 넣어준 부분을 주의하자. 이것을 넣지 않으면 ItemSaveForm 의 경우 규칙에 의해 itemSaveForm 이라는 이름으로 MVC Model에 담기게 된다. 이렇게 되면 뷰 템플릿에서 접근하는 th:object 이름도 함께 변경해주어야 한다.

## 폼 객체를 Item으로 변환

```
//성공 로직

Item item = new Item();
item.setItemName(form.getItemName());
item.setPrice(form.getPrice());
item.setQuantity(form.getQuantity());

Item savedItem = itemRepository.save(item);
```

폼 객체의 데이터를 기반으로 Item 객체를 생성한다. 이렇게 폼 객체처럼 중간에 다른 객체가 추가되면 변환하는 과정이 추가된다.

## 수정

```
@PostMapping("/{itemId}/edit")
public String edit(@PathVariable Long itemId, @Validated
@ModelAttribute("item") ItemUpdateForm form, BindingResult bindingResult) {
 //...
}
```

수정의 경우도 등록과 같다. 그리고 폼 객체를 Item 객체로 변환하는 과정을 거친다.

## 실행

<http://localhost:8080/validation/v4/items>

## 정리

Form 전송 객체 분리해서 등록과 수정에 딱 맞는 기능을 구성하고, 검증도 명확히 분리했다.

## Bean Validation - HTTP 메시지 컨버터

`@Valid`, `@Validated` 는 `HttpMessageConverter` (`@RequestBody`)에도 적용할 수 있다.

### 참고

`@ModelAttribute` 는 HTTP 요청 파라미터(URL 쿼리 스트링, POST Form)를 다룰 때 사용한다.

`@RequestBody` 는 HTTP Body의 데이터를 객체로 변환할 때 사용한다. 주로 API JSON 요청을 다룰 때 사용한다.

## ValidationItemApiController 생성

```
package hello.itemservice.web.validation;

import hello.itemservice.web.validation.form.ItemSaveForm;
import lombok.extern.slf4j.Slf4j;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@Slf4j
@RestController
@RequestMapping("/validation/api/items")
public class ValidationItemApiController {

 @PostMapping("/add")
 public Object addItem(@RequestBody @Validated ItemSaveForm form,
 BindingResult bindingResult) {

 log.info("API 컨트롤러 호출");
 }
}
```

```
if (bindingResult.hasErrors()) {
 log.info("검증 오류 발생 errors={}", bindingResult);
 return bindingResult.getAllErrors();
}

log.info("성공 로직 실행");
return form;
}
}
```

**Postman**을 사용해서 테스트 해보자.

### 성공 요청

```
POST http://localhost:8080/validation/api/items/add
{"itemName":"hello", "price":1000, "quantity": 10}
```

**Postman**에서 **Body** → **raw** → **JSON**을 선택해야 한다.

**API의 경우 3가지 경우를 나누어 생각해야 한다.**

- 성공 요청: 성공
- 실패 요청: JSON을 객체로 생성하는 것 자체가 실패함
- 검증 오류 요청: JSON을 객체로 생성하는 것은 성공했고, 검증에서 실패함

### 성공 요청 로그

```
API 컨트롤러 호출
성공 로직 실행
```

### 실패 요청

```
POST http://localhost:8080/validation/api/items/add
{"itemName":"hello", "price":"A", "quantity": 10}
```

`price`의 값에 숫자가 아닌 문자를 전달해서 실패하게 만들어보자.

## 실패 요청 결과

```
{
 "timestamp": "2021-04-20T00:00:00.000+00:00",
 "status": 400,
 "error": "Bad Request",
 "message": "",
 "path": "/validation/api/items/add"
}
```

## 실패 요청 로그

```
.w.s.m.s.DefaultHandlerExceptionResolver : Resolved
[org.springframework.http.converter.HttpMessageNotReadableException: JSON parse
error: Cannot deserialize value of type `java.lang.Integer` from String "A":
not a valid Integer value; nested exception is
com.fasterxml.jackson.databind.exc.InvalidFormatException: Cannot deserialize
value of type `java.lang.Integer` from String "A": not a valid Integer value
at [Source: (PushbackInputStream); line: 1, column: 30] (through reference
chain: hello.itemservice.domain.item.Item["price"])]
```

`HttpMessageConverter`에서 요청 JSON을 `Item` 객체로 생성하는데 실패한다.

이 경우는 `Item` 객체를 만들지 못하기 때문에 컨트롤러 자체가 호출되지 않고 그 전에 예외가 발생한다.  
물론 Validator도 실행되지 않는다.

## 검증 오류 요청

이번에는 `HttpMessageConverter`는 성공하지만 검증(Validator)에서 오류가 발생하는 경우를 확인해보자.

```
POST http://localhost:8080/validation/api/items/add
{"itemName":"hello", "price":1000, "quantity": 10000}
```

수량(`quantity`)이 10000이면 BeanValidation `@Max(9999)`에서 걸린다.

## 검증 오류 결과

```
[
 {
 "codes": [
 "Max.itemSaveForm.quantity",
 "Max.quantity",
 "Max.java.lang.Integer",
 "Max"
],
 "arguments": [
 {
 "codes": [
 "itemSaveForm.quantity",
 "quantity"
],
 "arguments": null,
 "defaultMessage": "quantity",
 "code": "quantity"
 },
 9999
],
 "defaultMessage": "9999 이하여야 합니다",
 "objectName": "itemSaveForm",
 "field": "quantity",
 "rejectedValue": 10000,
 "bindingFailure": false,
 "code": "Max"
 }
]
]
```

`return bindingResult.getAllErrors();` 는 `ObjectError` 와 `FieldError` 를 반환한다. 스프링이 이 객체를 JSON으로 변환해서 클라이언트에 전달했다. 여기서는 예시로 보여주기 위해서 검증 오류 객체들을 그대로 반환했다. 실제 개발할 때는 이 객체들을 그대로 사용하지 말고, 필요한 데이터만 뽑아서 별도의 API 스펙을 정의하고 그에 맞는 객체를 만들어서 반환해야 한다.

## 검증 오류 요청 로그

API 컨트롤러 호출

검증 오류 발생, errors=org.springframework.validation.BeanPropertyBindingResult: 1

```
errors
Field error in object 'itemSaveForm' on field 'quantity': rejected value
[99999]; codes
[Max.itemSaveForm.quantity,Max.quantity,Max.java.lang.Integer,Max]; arguments
[org.springframework.context.support.DefaultMessageSourceResolvable: codes
[itemSaveForm.quantity,quantity]; arguments []; default message
[quantity],9999]; default message [9999 이하여야 합니다]
```

로그를 보면 검증 오류가 정상 수행된 것을 확인할 수 있다.

### @ModelAttribute vs @RequestBody

HTTP 요청 파라미터를 처리하는 `@ModelAttribute`는 각각의 필드 단위로 세밀하게 적용된다. 그래서 특정 필드에 타입이 맞지 않는 오류가 발생해도 나머지 필드는 정상 처리할 수 있었다.

`HttpMessageConverter`는 `@ModelAttribute`와 다르게 각각의 필드 단위로 적용되는 것이 아니라, 전체 객체 단위로 적용된다.

따라서 메시지 컨버터의 작동이 성공해서 `Item` 객체를 만들어야 `@Valid`, `@Validated`가 적용된다.

- `@ModelAttribute`는 필드 단위로 정교하게 바인딩이 적용된다. 특정 필드가 바인딩 되지 않아도 나머지 필드는 정상 바인딩 되고, Validator를 사용한 검증도 적용할 수 있다.
- `@RequestBody`는 `HttpMessageConverter` 단계에서 JSON 데이터를 객체로 변경하지 못하면 이후 단계 자체가 진행되지 않고 예외가 발생한다. 컨트롤러도 호출되지 않고, Validator도 적용할 수 없다.

### 참고

`HttpMessageConverter` 단계에서 실패하면 예외가 발생한다. 예외 발생 시 원하는 모양으로 예외를 처리하는 방법은 예외 처리 부분에서 다룬다.

### 정리

## 6. 로그인 처리1 - 쿠키, 세션

#인강/5. 스프링 MVC 2/강의#

### 목차

- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 요구사항
- 6. 로그인 처리1 - 쿠키, 세션 - 프로젝트 생성
- 6. 로그인 처리1 - 쿠키, 세션 - 홈 화면
- 6. 로그인 처리1 - 쿠키, 세션 - 회원 가입
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 기능
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 쿠키 사용
- 6. 로그인 처리1 - 쿠키, 세션 - 쿠키와 보안 문제
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 세션 동작 방식
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 세션 직접 만들기
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 직접 만든 세션 적용
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 서블릿 HTTP 세션1
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 서블릿 HTTP 세션2
- 6. 로그인 처리1 - 쿠키, 세션 - 세션 정보와 타임아웃 설정
- 6. 로그인 처리1 - 쿠키, 세션 - 정리

### 로그인 요구사항

- 홈 화면 - 로그인 전
  - 회원 가입
  - 로그인
- 홈 화면 - 로그인 후
  - 본인 이름(누구님 환영합니다.)
  - 상품 관리
  - 로그 아웃
- 보안 요구사항
  - 로그인 사용자만 상품에 접근하고, 관리할 수 있음
  - 로그인 하지 않은 사용자가 상품 관리에 접근하면 로그인 화면으로 이동
- 회원 가입, 상품 관리

### 홈 화면 - 로그인 전

# 홈 화면

회원 가입

로그인

홈 화면 - 로그인 후

# 홈 화면

로그인: 테스트 회원

상품 관리

로그아웃

회원 가입

# 회원가입

## 회원 정보 입력

로그인 ID

비밀번호

이름

회원가입

취소

로그인

# 로그인

로그인 ID

비밀번호

로그인

취소

상품 관리

# 상품 목록

상품 등록

상품 ID	상품명	가격	수량
1	itemA	10000	10
2	itemB	20000	20

## 프로젝트 생성

이전 프로젝트에 이어서 로그인 기능을 학습해보자.

이전 프로젝트를 일부 수정해서 `login-start`라는 프로젝트에 넣어두었다.

## 프로젝트 설정 순서

1. `login-start`의 폴더 이름을 `login`로 변경하자.
2. **프로젝트 임포트**  
File → Open → 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, `Open as Project`를 선택하자.
3. `ItemServiceApplication.main()`을 실행해서 프로젝트가 정상 수행되는지 확인하자.

## 실행

- `http://localhost:8080`

실행하면 `HomeController`에서 `/items`로 redirect 한다.

## 패키지 구조 설계

### package 구조

- `hello.login`

- domain
  - item
  - member
  - login
- web
  - item
  - member
  - login

## 도메인이 가장 중요하다.

도메인 = 화면, UI, 기술 인프라 등등의 영역은 제외한 시스템이 구현해야 하는 핵심 비즈니스 업무 영역을 말함

향후 web을 다른 기술로 바꾸어도 도메인은 그대로 유지할 수 있어야 한다.

이렇게 하려면 web은 domain을 알고있지만 domain은 web을 모르도록 설계해야 한다. 이것을 web은 domain을 의존하지만, domain은 web을 의존하지 않는다고 표현한다. 예를 들어 web 패키지를 모두 삭제해도 domain에는 전혀 영향이 없도록 의존관계를 설계하는 것이 중요하다. 반대로 이야기하면 domain은 web을 참조하면 안된다.

## 홈 화면

홈 화면을 개발하자.

### HomeController - home() 설정

```
@GetMapping("/")
public String home() {
 return "home";
}
```

templates/home.html 추가

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
```

```

<link th:href="@{/css/bootstrap.min.css}"
 href="../css/bootstrap.min.css" rel="stylesheet">

</head>
<body>

<div class="container" style="max-width: 600px">
 <div class="py-5 text-center">
 <h2>홈 화면</h2>
 </div>

 <div class="row">
 <div class="col">
 <button class="w-100 btn btn-secondary btn-lg" type="button"
 th:onclick="|location.href='@{/members/add}' |">
 회원 가입
 </button>
 </div>
 <div class="col">
 <button class="w-100 btn btn-dark btn-lg"
 onclick="location.href='items.html'">
 th:onclick="|location.href='@{/login}' |" type="button">
 로그인
 </button>
 </div>
 </div>
</div> <!-- /container -->

</body>
</html>

```

**회원 가입**

**Member**

```
package hello.login.domain.member;

import lombok.Data;

import javax.validation.constraints.NotEmpty;

@Data
public class Member {

 private Long id;

 @NotEmpty
 private String loginId; //로그인 ID

 @NotEmpty
 private String name; //사용자 이름

 @NotEmpty
 private String password;

}
```

## MemberRepository

```
package hello.login.domain.member;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Repository;

import java.util.*;

/**
 * 동시성 문제가 고려되어 있지 않음, 실무에서는 ConcurrentHashMap, AtomicLong 사용 고려
 */
@Slf4j
@Repository
public class MemberRepository {

 private static Map<Long, Member> store = new HashMap<>(); //static 사용
```

```

private static long sequence = 0L; //static 사용

public Member save(Member member) {
 member.setId(++sequence);
 log.info("save: member={}", member);
 store.put(member.getId(), member);
 return member;
}

public Member findById(Long id) {
 return store.get(id);
}

public Optional<Member> findByLoginId(String loginId) {
 return findAll().stream()
 .filter(m -> m.getLoginId().equals(loginId))
 .findFirst();
}

public List<Member> findAll() {
 return new ArrayList<>(store.values());
}

public void clearStore() {
 store.clear();
}

```

## MemberController

```

package hello.login.web.member;

import hello.login.domain.member.Member;
import hello.login.domain.member.MemberRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
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;

import javax.validation.Valid;

@Controller
@RequiredArgsConstructor
@RequestMapping("/members")
public class MemberController {

 private final MemberRepository memberRepository;

 @GetMapping("/add")
 public String addForm(@ModelAttribute("member") Member member) {
 return "members/addMemberForm";
 }

 @PostMapping("/add")
 public String save(@Valid @ModelAttribute Member member, BindingResult result) {
 if (result.hasErrors()) {
 return "members/addMemberForm";
 }

 memberRepository.save(member);
 return "redirect:/";
 }
}

```

`@ModelAttribute("member")` 를 `@ModelAttribute` 로 변경해도 결과는 같다. 여기서는 IDE에서 인식 문제가 있어서 적용했다.

### 회원가입 뷰 템플릿

`templates/members/addMemberForm.html`

```

<!DOCTYPE HTML>

<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
 <link th:href="@{/css/bootstrap.min.css}"
 href="../css/bootstrap.min.css" rel="stylesheet">
 <style>
 .container {
 max-width: 560px;
 }
 .field-error {
 border-color: #dc3545;
 color: #dc3545;
 }
 </style>
</head>
<body>

<div class="container">

 <div class="py-5 text-center">
 <h2>회원 가입</h2>
 </div>

 <h4 class="mb-3">회원 정보 입력</h4>

 <form action="" th:action th:object="${member}" method="post">

 <div th:if="#{fields.hasGlobalErrors()}">
 <p class="field-error" th:each="err : ${#fields.globalErrors()}" th:text="${err}">전체 오류 메시지</p>
 </div>

 <div>
 <label for="loginId">로그인 ID</label>
 <input type="text" id="loginId" th:field="*{loginId}" class="form-control"
 th:errorclass="field-error">
 <div class="field-error" th:errors="*{loginId}" />
 </div>
 </form>
</div>

```

```

 </div>

 <div>
 <label for="password">비밀번호</label>
 <input type="password" id="password" th:field="*{password}"
class="form-control"
 th:errorclass="field-error">
 <div class="field-error" th:errors="*{password}" />
 </div>
 <div>
 <label for="name">이름</label>
 <input type="text" id="name" th:field="*{name}" class="form-
control"
 th:errorclass="field-error">
 <div class="field-error" th:errors="*{name}" />
 </div>

 <hr class="my-4">

 <div class="row">
 <div class="col">
 <button class="w-100 btn btn-primary btn-lg" type="submit">회원
가입</button>
 </div>
 <div class="col">
 <button class="w-100 btn btn-secondary btn-lg"
onclick="location.href='items.html'">
 th:onclick="|location.href='@{/}'|"
 type="button">취소</button>
 </div>
 </div>
 </div>
 </form>

 </div> <!-- /container -->
</body>
</html>

```

## 실행하고 로그로 결과를 확인하자

### 회원용 테스트 데이터 추가

편의상 테스트용 회원 데이터를 추가하자.

```
loginId : test
password : test!
name : 테스터
```

### TestDataInit

```
package hello.login;

import hello.login.domain.item.Item;
import hello.login.domain.item.ItemRepository;
import hello.login.domain.member.Member;
import hello.login.domain.member.MemberRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

@Component
@RequiredArgsConstructor
public class TestDataInit {

 private final ItemRepository itemRepository;
 private final MemberRepository memberRepository;

 /**
 * 테스트용 데이터 추가
 */
 @PostConstruct
 public void init() {
 itemRepository.save(new Item("itemA", 10000, 10));
 itemRepository.save(new Item("itemB", 20000, 20));

 Member member = new Member();
 member.setLoginId("test");
 }
}
```

```
 member.setPassword("test!");
 member.setName("테스터");
 memberRepository.save(member);
 }

}
```

## 로그인 기능

로그인 기능을 개발해보자. 지금은 로그인 ID, 비밀번호를 입력하는 부분에 집중하자.

# 로그인

---

로그인 ID

비밀번호

---

## LoginService

```
package hello.login.domain.login;

import hello.login.domain.member.Member;
import hello.login.domain.member.MemberRepository;
import lombok.RequiredArgsConstructor;
```

```

import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class LoginService {

 private final MemberRepository memberRepository;

 /**
 * @return null이면 로그인 실패
 */

 public Member login(String loginId, String password) {
 return memberRepository.findByLoginId(loginId)
 .filter(m -> m.getPassword().equals(password))
 .orElse(null);
 }
}

```

로그인의 핵심 비즈니스 로직은 회원을 조회한 다음에 파라미터로 넘어온 password와 비교해서 같으면 회원을 반환하고, 만약 password가 다르면 `null`을 반환한다.

## LoginForm

```

package hello.login.web.login;

import lombok.Data;

import javax.validation.constraints.NotEmpty;

@Data
public class LoginForm {

 @NotEmpty
 private String loginId;
 @NotEmpty
 private String password;
}

```

## LoginController

```
package hello.login.web.login;

import hello.login.domain.login.LoginService;
import hello.login.domain.member.Member;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.*;

import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletResponse;
import javax.validation.Valid;
import java.util.Objects;

@Slf4j
@Controller
@RequiredArgsConstructor
public class LoginController {

 private final LoginService loginService;

 @GetMapping("/login")
 public String loginForm(@ModelAttribute("loginForm") LoginForm form) {
 return "login/loginForm";
 }

 @PostMapping("/login")
 public String login(@Valid @ModelAttribute LoginForm form, BindingResult
bindingResult) {
 if (bindingResult.hasErrors()) {
 return "login/loginForm";
 }

 Member loginMember = loginService.login(form.getLoginId(),
form.getPassword());
```

```

 log.info("login? {}", loginMember);

 if (loginMember == null) {
 bindingResult.reject("loginFail", "아이디 또는 비밀번호가 맞지 않습니다.");
 return "login/loginForm";
 }

 //로그인 성공 처리 TODO

 return "redirect:/";
 }

}

```

로그인 컨트롤러는 로그인 서비스를 호출해서 로그인에 성공하면 홈 화면으로 이동하고, 로그인에 실패하면 `bindingResult.reject()`를 사용해서 글로벌 오류(`ObjectError`)를 생성한다. 그리고 정보를 다시 입력하도록 로그인 폼을 뷰 템플릿으로 사용한다.

## 로그인 폼 뷰 템플릿

`templates/login/loginForm.html`

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
 <link th:href="@{/css/bootstrap.min.css}"
 href="../css/bootstrap.min.css" rel="stylesheet">
 <style>
 .container {
 max-width: 560px;
 }
 .field-error {
 border-color: #dc3545;
 color: #dc3545;
 }
 </style>
</head>
<body>

```

```
<div class="container">

 <div class="py-5 text-center">
 <h2>로그인</h2>
 </div>

 <form action="item.html" th:action th:object="${loginForm}" method="post">

 <div th:if="#{fields.hasGlobalErrors()}">
 <p class="field-error" th:each="err : ${#fields.globalErrors()}" th:text="${err}">전체 오류 메시지</p>
 </div>

 <div>
 <label for="loginId">로그인 ID</label>
 <input type="text" id="loginId" th:field="*{loginId}" class="form-control" th:errorclass="field-error">
 <div class="field-error" th:errors="*{loginId}" />
 </div>

 <div>
 <label for="password">비밀번호</label>
 <input type="password" id="password" th:field="*{password}" class="form-control" th:errorclass="field-error">
 <div class="field-error" th:errors="*{password}" />
 </div>

 <hr class="my-4">

 <div class="row">
 <div class="col">
 <button class="w-100 btn btn-primary btn-lg" type="submit">
 로그인</button>
 </div>
 <div class="col">
 <button class="w-100 btn btn-secondary btn-lg" onclick="location.href='items.html'">

```

```
 th:onclick="| location.href='@{/} |"
 type="button">취소</button>
 </div>
</div>

</form>
</div> <!-- /container -->
</body>
</html>
```

로그인 폼 뷰 템플릿에는 특별한 코드는 없다. `loginId`, `password` 가 틀리면 글로벌 오류가 나타난다.

### 실행

실행해보면 로그인이 성공하면 홈으로 이동하고, 로그인에 실패하면 "아이디 또는 비밀번호가 맞지 않습니다."라는 경고와 함께 로그인 폼이 나타난다.

그런데 아직 로그인이 되면 홈 화면에 고객 이름이 보여야 한다는 요구사항을 만족하지 못한다. 로그인의 상태를 유지하면서, 로그인에 성공한 사용자는 홈 화면에 접근시 고객의 이름을 보여주려면 어떻게 해야할까?

## 로그인 처리하기 - 쿠키 사용

### 참고

여기서는 여러분이 쿠키의 기본 개념을 이해하고 있다고 가정한다. 쿠키에 대해서는 **모든 개발자를 위한 HTTP 기본 지식 강의**를 참고하자. 혹시 잘 생각이 안나면 쿠키 관련 내용을 꼭! 복습하고 돌아오자.

쿠키를 사용해서 로그인, 로그아웃 기능을 구현해보자.

### 로그인 상태 유지하기

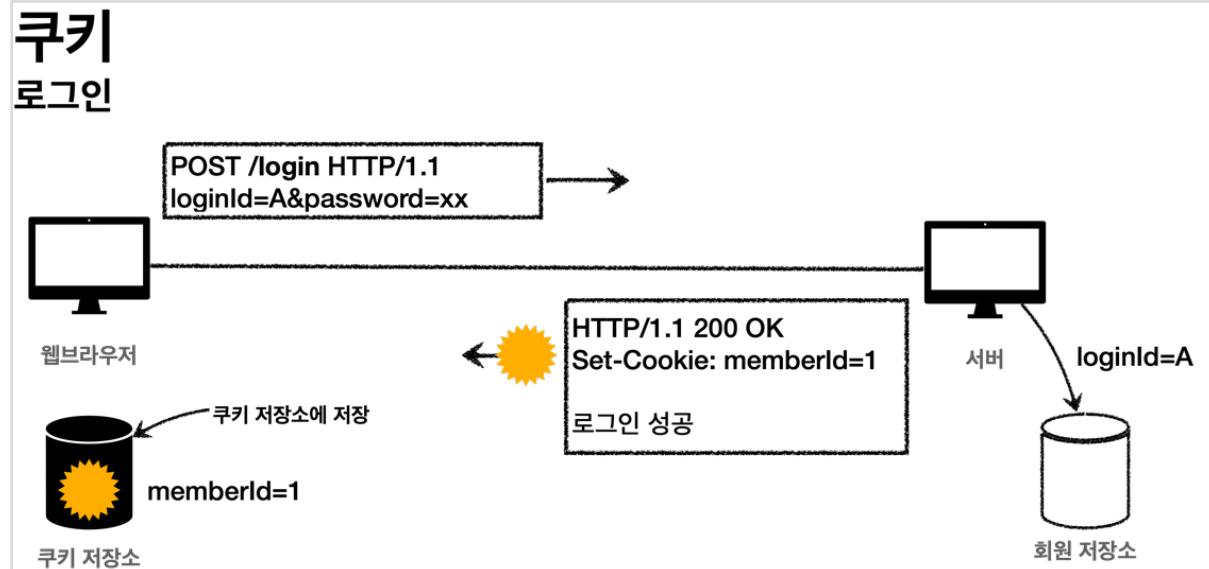
로그인의 상태를 어떻게 유지할 수 있을까?

HTTP 강의에서 일부 설명했지만, 쿼리 파라미터를 계속 유지하면서 보내는 것은 매우 어렵고 번거로운 작업이다. 쿠키를 사용해보자.

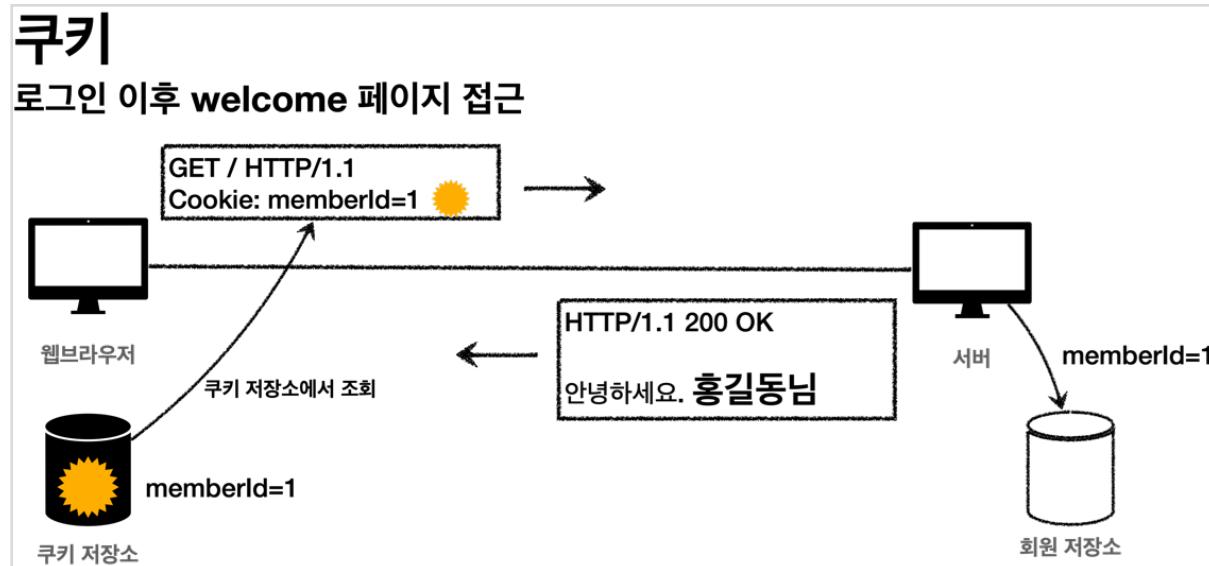
### 쿠키

서버에서 로그인에 성공하면 HTTP 응답에 쿠키를 담아서 브라우저에 전달하자. 그러면 브라우저는 앞으로 해당 쿠키를 지속해서 보내준다.

### 쿠키 생성



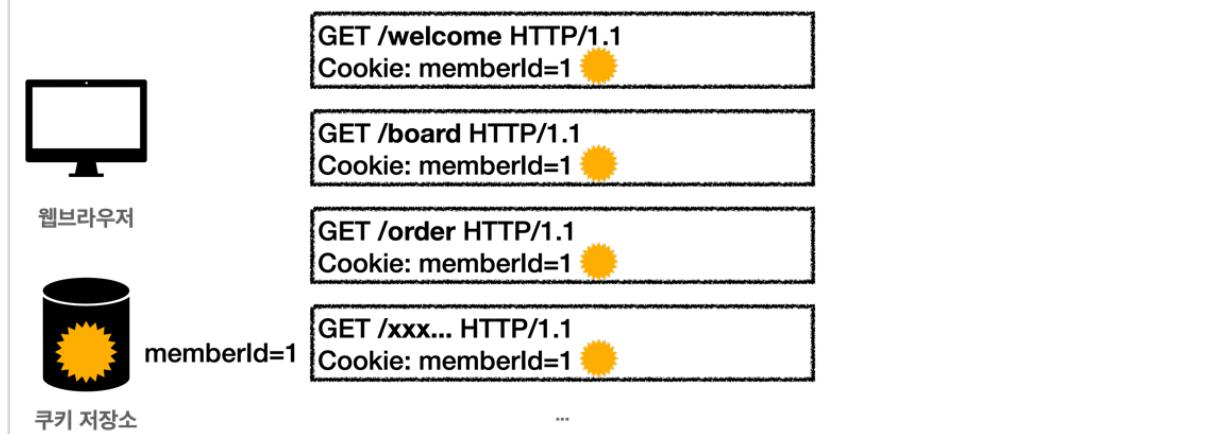
### 클라이언트 쿠키 전달1



### 클라이언트 쿠키 전달2

# 쿠키

모든 요청에 쿠키 정보 자동 포함



쿠키에는 영속 쿠키와 세션 쿠키가 있다.

- 영속 쿠키: 만료 날짜를 입력하면 해당 날짜까지 유지
- 세션 쿠키: 만료 날짜를 생략하면 브라우저 종료시 까지만 유지

브라우저 종료시 로그아웃이 되길 기대하므로, 우리에게 필요한 것은 세션 쿠키이다.

## LoginController - login()

로그인 성공시 세션 쿠키를 생성하자.

```
@PostMapping("/login")
public String login(@Valid @ModelAttribute LoginForm form, BindingResult
bindingResult, HttpServletResponse response) {
 if (bindingResult.hasErrors()) {
 return "login/loginForm";
 }

 Member loginMember = loginService.login(form.getLoginId(),
form.getPassword());
 log.info("login? {}", loginMember);

 if (loginMember == null) {
 bindingResult.reject("loginFail", "아이디 또는 비밀번호가 맞지 않습니다.");
 return "login/loginForm";
 }
}
```

```

//로그인 성공 처리

//쿠키에 시간 정보를 주지 않으면 세션 쿠키(브라우저 종료시 모두 종료)
Cookie idCookie = new Cookie("memberId",
String.valueOf(loginMember.getId()));
response.addCookie(idCookie);

return "redirect:/";
}

```

## 쿠키 생성 로직

```

Cookie idCookie = new Cookie("memberId", String.valueOf(loginMember.getId()));
response.addCookie(idCookie);

```

로그인에 성공하면 쿠키를 생성하고 `HttpServletResponse`에 담는다. 쿠키 이름은 `memberId`이고, 값은 회원의 `id`를 담아둔다. 웹 브라우저는 종료 전까지 회원의 `id`를 서버에 계속 보내줄 것이다.

## 실행

크롬 브라우저를 통해 HTTP 응답 헤더에 쿠키가 추가된 것을 확인할 수 있다.

이제 요구사항에 맞추어 로그인에 성공하면 로그인 한 사용자 전용 홈 화면을 보여주자.

## 홈 - 로그인 처리

```

package hello.login.web;

import hello.login.domain.member.Member;
import hello.login.domain.member.MemberRepository;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.CookieValue;
import org.springframework.web.bind.annotation.GetMapping;

@Slf4j

```

```

@Controller
@RequiredArgsConstructor
public class HomeController {

 private final MemberRepository memberRepository;

 // @GetMapping("/")
 public String home() {
 return "home";
 }

 @GetMapping("/")
 public String homeLogin(
 @CookieValue(name = "memberId", required = false) Long memberId,
 Model model) {

 if (memberId == null) {
 return "home";
 }

 //로그인
 Member loginMember = memberRepository.findById(memberId);
 if (loginMember == null) {
 return "home";
 }

 model.addAttribute("member", loginMember);
 return "loginHome";
 }

}

```

- 기존 `home()`에 있는 `@GetMapping("/")`은 주석 처리하자.
- `@CookieValue`를 사용하면 편리하게 쿠키를 조회할 수 있다.
- 로그인 하지 않은 사용자도 홈에 접근할 수 있기 때문에 `required = false`를 사용한다.

### 로직 분석

- 로그인 쿠키(`memberId`)가 없는 사용자는 기존 `home`으로 보낸다. 추가로 로그인 쿠키가 있어도 회원이

없으면 home 으로 보낸다.

- 로그인 쿠키( memberId )가 있는 사용자는 로그인 사용자 전용 홈 화면인 loginHome 으로 보낸다. 추가로 홈 화면에 회원 관련 정보도 출력해야 해서 member 데이터도 모델에 담아서 전달한다.

## 홈 - 로그인 사용자 전용

templates/loginHome.html

```
<!DOCTYPE HTML>

<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
 <link th:href="@{/css/bootstrap.min.css}"
 href="../css/bootstrap.min.css" rel="stylesheet">
</head>
<body>

<div class="container" style="max-width: 600px">
 <div class="py-5 text-center">
 <h2>홈 화면</h2>
 </div>

 <h4 class="mb-3" th:text="|로그인: ${member.name} |">로그인 사용자 이름</h4>

 <hr class="my-4">

 <div class="row">
 <div class="col">
 <button class="w-100 btn btn-secondary btn-lg" type="button"
 th:onclick="|location.href='@{/items}' |">
 상품 관리
 </button>
 </div>
 <div class="col">
 <form th:action="@{/logout}" method="post">
 <button class="w-100 btn btn-dark btn-lg"
 onclick="location.href='items.html'" type="submit">
 로그아웃
 </button>
 </form>
 </div>
 </div>
</div>
```

```

</div>

</div>

<hr class="my-4">

</div> <!-- /container -->

</body>
</html>

```

- `th:text="|로그인: ${member.name}|"` : 로그인에 성공한 사용자 이름을 출력한다.
- 상품 관리, 로그아웃 버튼을 노출한다.

### 실행

로그인에 성공하면 사용자 이름이 출력되면서 상품 관리, 로그아웃 버튼을 확인할 수 있다. 로그인에 성공시 세션 쿠키가 지속해서 유지되고, 웹 브라우저에서 서버에 요청시 `memberId` 쿠키를 계속 보내준다.

### 로그아웃 기능

이번에는 로그아웃 기능을 만들어보자. 로그아웃 방법은 다음과 같다.

- 세션 쿠키이므로 웹 브라우저 종료시
- 서버에서 해당 쿠키의 종료 날짜를 0으로 지정

### LoginController - logout 기능 추가

```

@PostMapping("/logout")
public String logout(HttpServletRequest response) {
 expireCookie(response, "memberId");
 return "redirect:/";
}

private void expireCookie(HttpServletRequest response, String cookieName) {
 Cookie cookie = new Cookie(cookieName, null);
 cookie.setMaxAge(0);
 response.addCookie(cookie);
}

```

## 실행

로그아웃도 응답 쿠키를 생성하는데 `Max-Age=0` 를 확인할 수 있다. 해당 쿠키는 즉시 종료된다.

## 쿠키와 보안 문제

쿠키를 사용해서 로그인Id를 전달해서 로그인을 유지할 수 있었다. 그런데 여기에는 심각한 보안 문제가 있다.

### 보안 문제

- 쿠키 값은 임의로 변경할 수 있다.
  - 클라이언트가 쿠키를 강제로 변경하면 다른 사용자가 된다.
  - 실제 웹브라우저 개발자모드 → Application → Cookie 변경으로 확인
    - `Cookie: memberId=1` → `Cookie: memberId=2` (다른 사용자의 이름이 보임)
- 쿠키에 보관된 정보는 훔쳐갈 수 있다.
  - 만약 쿠키에 개인정보나, 신용카드 정보가 있다면?
  - 이 정보가 웹 브라우저에도 보관되고, 네트워크 요청마다 계속 클라이언트에서 서버로 전달된다.
  - 쿠키의 정보가 나의 로컬 PC가 털릴 수도 있고, 네트워크 전송 구간에서 털릴 수도 있다.
- 해커가 쿠키를 한번 훔쳐가면 평생 사용할 수 있다.
  - 해커가 쿠키를 훔쳐가서 그 쿠키로 악의적인 요청을 계속 시도할 수 있다.

### 대안

- 쿠키에 중요한 값을 노출하지 않고, 사용자 별로 예측 불가능한 임의의 토큰(랜덤 값)을 노출하고, 서버에서 토큰과 사용자 id를 맵핑해서 인식한다. 그리고 서버에서 토큰을 관리한다.
- 토큰은 해커가 임의의 값을 넣어도 찾을 수 없도록 예상 불가능 해야 한다.
- 해커가 토큰을 털어가도 시간이 지나면 사용할 수 없도록 서버에서 해당 토큰의 만료시간을 짧게(예: 30분) 유지한다. 또는 해킹이 의심되는 경우 서버에서 해당 토큰을 강제로 제거하면 된다.

## 로그인 처리하기 - 세션 동작 방식

### 목표

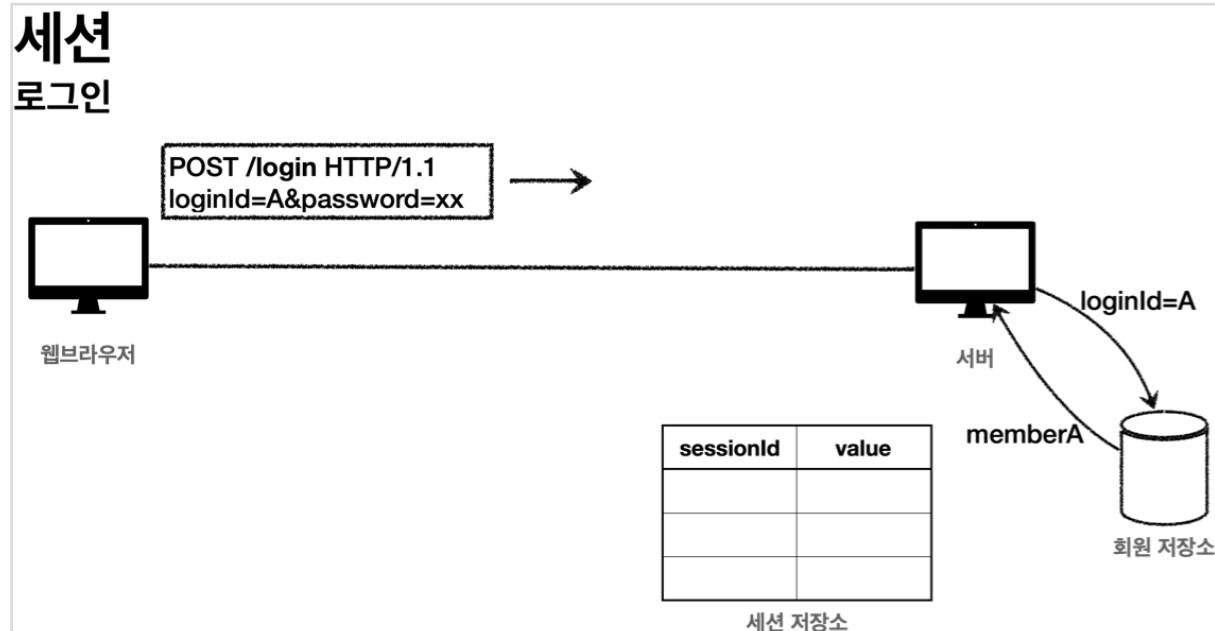
앞서 쿠키에 중요한 정보를 보관하는 방법은 여러가지 보안 이슈가 있었다. 이 문제를 해결하려면 결국 중요한 정보를 모두 서버에 저장해야 한다. 그리고 클라이언트와 서버는 추정 불가능한 임의의 식별자 값으로 연결해야 한다.

이렇게 서버에 중요한 정보를 보관하고 연결을 유지하는 방법을 세션이라 한다.

## 세션 동작 방식

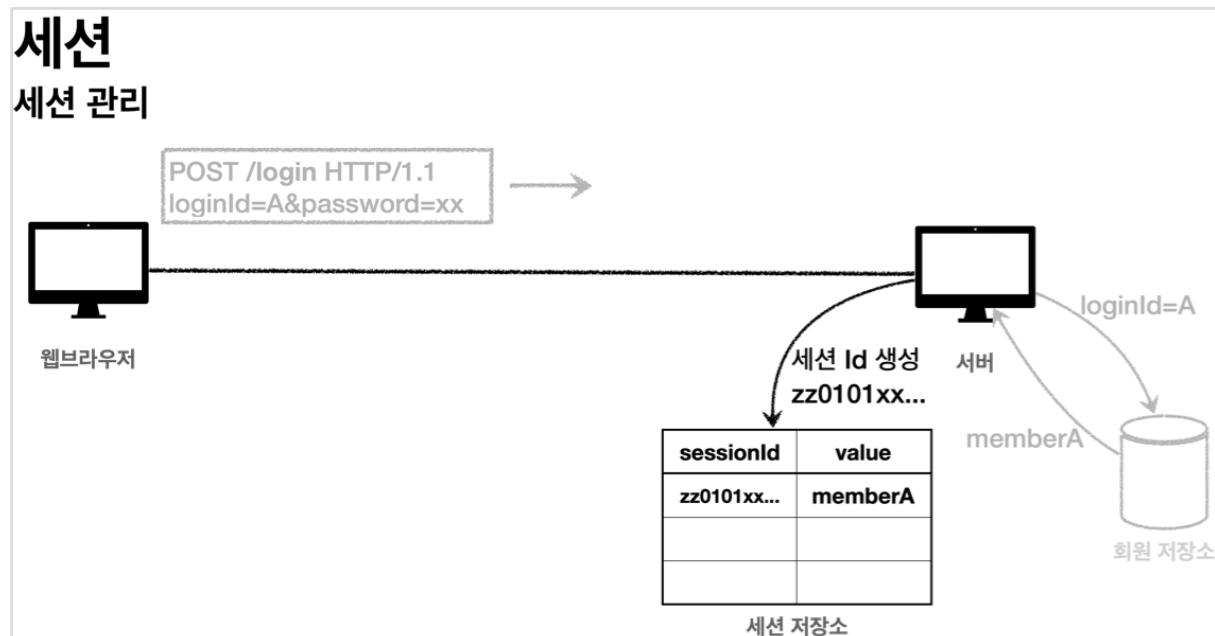
세션을 어떻게 개발할지 먼저 개념을 이해해보자.

### 로그인



- 사용자가 `loginId`, `password` 정보를 전달하면 서버에서 해당 사용자가 맞는지 확인한다.

### 세션 생성

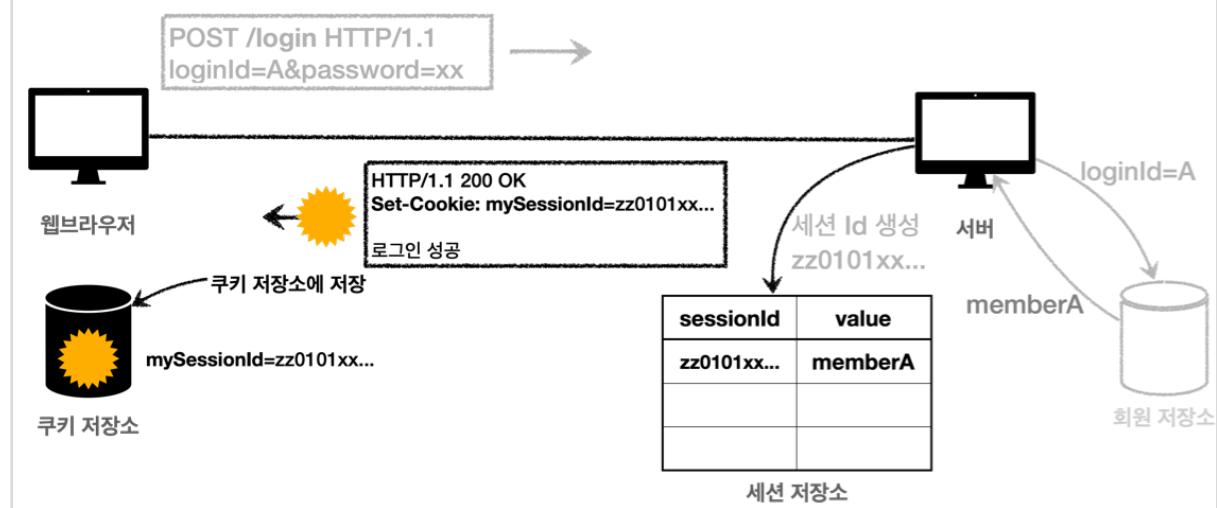


- 세션 ID를 생성하는데, 추정 불가능해야 한다.
- **UUID는 추정이 불가능하다.**
  - Cookie: mySessionId=zz0101xx-bab9-4b92-9b32-dadb280f4b61
- 생성된 세션 ID와 세션에 보관할 값( memberA )을 서버의 세션 저장소에 보관한다.

세션id를 응답 쿠키로 전달

## 세션

### 세션id를 쿠키로 전달



클라이언트와 서버는 결국 쿠키로 연결이 되어야 한다.

- 서버는 클라이언트에 `mySessionId`라는 이름으로 세션ID 만 쿠키에 담아서 전달한다.
- 클라이언트는 쿠키 저장소에 `mySessionId` 쿠키를 보관한다.

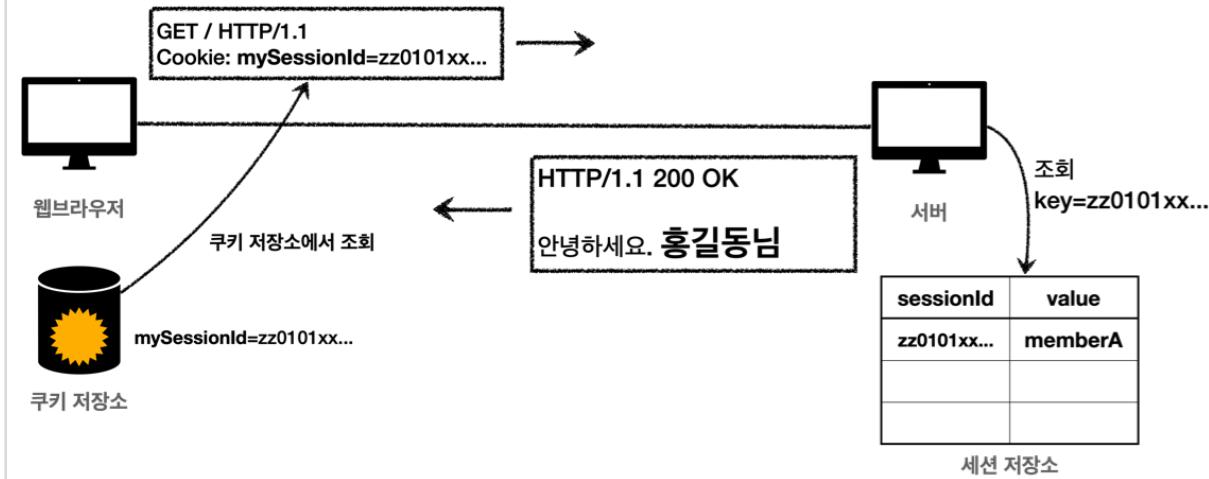
### 중요

- 여기서 중요한 포인트는 회원과 관련된 정보는 전혀 클라이언트에 전달하지 않는다는 것이다.
- 오직 추정 불가능한 세션 ID만 쿠키를 통해 클라이언트에 전달한다.

### 클라이언트의 세션id 쿠키 전달

# 세션

## 로그인 이후 접근



- 클라이언트는 요청시 항상 `mySessionId` 쿠키를 전달한다.
- 서버에서는 클라이언트가 전달한 `mySessionId` 쿠키 정보로 세션 저장소를 조회해서 로그인시 보관한 세션 정보를 사용한다.

## 정리

세션을 사용해서 서버에서 중요한 정보를 관리하게 되었다. 덕분에 다음과 같은 보안 문제들을 해결할 수 있다.

- 쿠키 값을 변조 가능, → 예상 불가능한 복잡한 세션ID를 사용한다.
- 쿠키에 보관하는 정보는 클라이언트 해킹시 털릴 가능성이 있다. → 세션ID가 털려도 여기에는 중요한 정보가 없다.
- 쿠키 탈취 후 사용 → 해커가 토큰을 털어가도 시간이 지나면 사용할 수 없도록 서버에서 세션의 만료시간을 짧게(예: 30분) 유지한다. 또는 해킹이 의심되는 경우 서버에서 해당 세션을 강제로 제거하면 된다.

## 로그인 처리하기 - 세션 직접 만들기

세션을 직접 개발해서 적용해보자.

세션 관리는 크게 다음 3가지 기능을 제공하면 된다.

- 세션 생성
  - sessionId 생성 (임의의 추정 불가능한 랜덤 값)

- 세션 저장소에 sessionId와 보관할 값 저장
- sessionId로 응답 쿠키를 생성해서 클라이언트에 전달
- **세션 조회**
  - 클라이언트가 요청한 sessionId 쿠키의 값으로, 세션 저장소에 보관한 값 조회
- **세션 만료**
  - 클라이언트가 요청한 sessionId 쿠키의 값으로, 세션 저장소에 보관한 sessionId와 값 제거

## SessionManager - 세션 관리

```

package hello.login.web.session;

import org.springframework.stereotype.Component;

import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Arrays;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.ConcurrentHashMap;

/**
 * 세션 관리
 */
@Component
public class SessionManager {

 public static final String SESSION_COOKIE_NAME = "mySessionId";

 private Map<String, Object> sessionStore = new ConcurrentHashMap<>();

 /**
 * 세션 생성
 */
 public void createSession(Object value, HttpServletResponse response) {
 //세션 id를 생성하고, 값을 세션에 저장
 String sessionId = UUID.randomUUID().toString();
 sessionStore.put(sessionId, value);
}

```

```
//쿠키 생성
Cookie mySessionCookie = new Cookie(SESSION_COOKIE_NAME, sessionId);
response.addCookie(mySessionCookie);

}

/***
 * 세션 조회
 */
public Object getSession(HttpServletRequest request) {
 Cookie sessionCookie = findCookie(request, SESSION_COOKIE_NAME);
 if (sessionCookie == null) {
 return null;
 }
 return sessionStore.get(sessionCookie.getValue());
}

/***
 * 세션 만료
 */
public void expire(HttpServletRequest request) {
 Cookie sessionCookie = findCookie(request, SESSION_COOKIE_NAME);
 if (sessionCookie != null) {
 sessionStore.remove(sessionCookie.getValue());
 }
}

private Cookie findCookie(HttpServletRequest request, String cookieName) {
 if (request.getCookies() == null) {
 return null;
 }
 return Arrays.stream(request.getCookies())
 .filter(cookie -> cookie.getName().equals(cookieName))
 .findAny()
 .orElse(null);
}

}
```

로직은 어렵지 않아서 쉽게 이해가 될 것이다.

- `@Component` : 스프링 빈으로 자동 등록한다.
- `ConcurrentHashMap` : `HashMap`은 동시 요청에 안전하지 않다. 동시 요청에 안전한 `ConcurrentHashMap`을 사용했다.

### SessionManagerTest - 테스트

```
package hello.login.web.session;

import hello.login.domain.member.Member;
import org.junit.jupiter.api.Test;
import org.springframework.mock.web.MockHttpServletRequest;
import org.springframework.mock.web.MockHttpServletResponse;

import static org.assertj.core.api.Assertions.assertThat;

class SessionManagerTest {

 SessionManager sessionManager = new SessionManager();

 @Test
 void sessionTest() {

 //세션 생성
 MockHttpServletResponse response = new MockHttpServletResponse();
 Member member = new Member();
 sessionManager.createSession(member, response);

 //요청에 응답 쿠키 저장
 MockHttpServletRequest request = new MockHttpServletRequest();
 request.setCookies(response.getCookies());

 //세션 조회
 Object result = sessionManager.getSession(request);
 assertThat(result).isEqualTo(member);
 }
}
```

```

//세션 만료
sessionManager.expire(request);
Object expired = sessionManager.getSession(request);
assertThat(expired).isNull();
}
}

```

간단하게 테스트를 진행해보자. 여기서는 `HttpServletRequest`, `HttpServletResponse` 객체를 직접 사용할 수 없기 때문에 테스트에서 비슷한 역할을 해주는 가짜 `MockHttpServletRequest`, `MockHttpServletResponse` 를 사용했다.

## 로그인 처리하기 - 직접 만든 세션 적용

지금까지 개발한 세션 관리 기능을 실제 웹 애플리케이션에 적용해보자.

### **LoginController - loginV2()**

```

@PostMapping("/login")
public String loginV2(@Valid @ModelAttribute LoginForm form, BindingResult
bindingResult, HttpServletResponse response) {
 if (bindingResult.hasErrors()) {
 return "login/loginForm";
 }

 Member loginMember = loginService.login(form.getLoginId(),
 form.getPassword());
 log.info("login? {}", loginMember);

 if (loginMember == null) {
 bindingResult.reject("loginFail", "아이디 또는 비밀번호가 맞지 않습니다.");
 return "login/loginForm";
 }

 //로그인 성공 처리
}

```

```

//세션 관리자를 통해 세션을 생성하고, 회원 데이터 보관
sessionManager.createSession(loginMember, response);

return "redirect:/";
}

```

- 기존 `login()` 의 `@PostMapping("/login")` 주석 처리

- `private final SessionManager sessionManager;` 주입
- `sessionManager.createSession(loginMember, response);`

로그인 성공시 세션을 등록한다. 세션에 `loginMember` 를 저장해두고, 쿠키도 함께 발행한다.

### LoginController - logoutV2()

```

@PostMapping("/logout")
public String logoutV2(HttpServletRequest request) {
 sessionManager.expire(request);
 return "redirect:/";
}

```

- 기존 `logout()` 의 `@PostMapping("/logout")` 주석 처리

로그 아웃시 해당 세션의 정보를 제거한다.

### HomeController - homeLoginV2()

```

@GetMapping("/")
public String homeLoginV2(HttpServletRequest request, Model model) {

 //세션 관리자에 저장된 회원 정보 조회
 Member member = (Member)sessionManager.getSession(request);
 if (member == null) {
 return "home";
 }
}

```

```
//로그인
model.addAttribute("member", member);
return "loginHome";
}
```

- `private final SessionManager sessionManager;` 주입
- 기존 `homeLogin()` 의 `@GetMapping("/")` 주석 처리

세션 관리자에서 저장된 회원 정보를 조회한다. 만약 회원 정보가 없으면, 쿠키나 세션이 없는 것 이므로 로그인 되지 않은 것으로 처리한다.

### 정리

이번 시간에는 세션과 쿠키의 개념을 명확하게 이해하기 위해서 직접 만들어보았다. 사실 세션이라는 것이 뭔가 특별한 것이 아니라 단지 쿠키를 사용하는데, 서버에서 데이터를 유지하는 방법일 뿐이라는 것을 이해했을 것이다.

그런데 프로젝트마다 이러한 세션 개념을 직접 개발하는 것은 상당히 불편할 것이다. 그래서 서블릿도 세션 개념을 지원한다.

이제 직접 만드는 세션 말고, 서블릿이 공식 지원하는 세션을 알아보자. 서블릿이 공식 지원하는 세션은 우리가 직접 만든 세션과 동작 방식이 거의 같다. 추가로 세션을 일정시간 사용하지 않으면 해당 세션을 삭제하는 기능을 제공한다.

## 로그인 처리하기 - 서블릿 HTTP 세션1

세션이라는 개념은 대부분의 웹 애플리케이션에 필요한 것이다. 어쩌면 웹이 등장하면서부터 나온 문제이다.

서블릿은 세션을 위해 `HttpSession`이라는 기능을 제공하는데, 지금까지 나온 문제들을 해결해준다. 우리가 직접 구현한 세션의 개념이 이미 구현되어 있고, 더 잘 구현되어 있다.

### HttpSession 소개

서블릿이 제공하는 `HttpSession` 도 결국 우리가 직접 만든 `SessionManager` 와 같은 방식으로 동작한다. 서블릿을 통해 `HttpSession` 을 생성하면 다음과 같은 쿠키를 생성한다. 쿠키 이름이 `JSESSIONID` 이고, 값은 추정 불가능한 랜덤 값이다.

```
Cookie: JSESSIONID=5B78E23B513F50164D6FDD8C97B0AD05
```

## HttpSession 사용

서블릿이 제공하는 `HttpSession`을 사용하도록 개발해보자.

## SessionConst

```
package hello.login.web;

public class SessionConst {
 public static final String LOGIN_MEMBER = "loginMember";
}
```

`HttpSession`에 데이터를 보관하고 조회할 때, 같은 이름이 중복 되어 사용되므로, 상수를 하나 정의했다.

## LoginController - loginV3()

```
@PostMapping("/login")
public String loginV3(@Valid @ModelAttribute LoginForm form, BindingResult
bindingResult, HttpServletRequest request) {
 if (bindingResult.hasErrors()) {
 return "login/loginForm";
 }

 Member loginMember = loginService.login(form.getLoginId(),
form.getPassword());
 log.info("login? {}", loginMember);

 if (loginMember == null) {
 bindingResult.reject("loginFail", "아이디 또는 비밀번호가 맞지 않습니다.");
 return "login/loginForm";
 }

 //로그인 성공 처리

 //세션이 있으면 있는 세션 반환, 없으면 신규 세션 생성
 HttpSession session = request.getSession();
 //세션에 로그인 회원 정보 보관
 session.setAttribute(SessionConst.LOGIN_MEMBER, loginMember);
```

```
 return "redirect:/";
```

```
}
```

- 기존 `loginV2()` 의 `@PostMapping("/login")` 주석 처리

### 세션 생성과 조회

세션을 생성하려면 `request.getSession(true)` 를 사용하면 된다.

```
public HttpSession getSession(boolean create);
```

세션의 `create` 옵션에 대해 알아보자.

- `request.getSession(true)`
  - 세션이 있으면 기존 세션을 반환한다.
  - 세션이 없으면 새로운 세션을 생성해서 반환한다.
- `request.getSession(false)`
  - 세션이 있으면 기존 세션을 반환한다.
  - 세션이 없으면 새로운 세션을 생성하지 않는다. `null` 을 반환한다.
- `request.getSession()` : 신규 세션을 생성하는 `request.getSession(true)` 와 동일하다.

### 세션에 로그인 회원 정보 보관

```
session.setAttribute(SessionConst.LOGIN_MEMBER, loginMember);
```

세션에 데이터를 보관하는 방법은 `request.setAttribute(..)` 와 비슷하다. 하나의 세션에 여러 값을 저장할 수 있다.

### LoginController - logoutV3()

```
@PostMapping("/logout")
public String logoutV3(HttpServletRequest request) {
 //세션을 삭제한다.

 HttpSession session = request.getSession(false);
 if (session != null) {
 session.invalidate();
 }
 return "redirect:/";
}
```

- 기존 `logoutV2()` 의 `@PostMapping("/logout")` 주석 처리

`session.invalidate()`: 세션을 제거한다.

## HomeController - homeLoginV3()

```
@GetMapping("/")
public String homeLoginV3(HttpServletRequest request, Model model) {

 //세션이 없으면 home
 HttpSession session = request.getSession(false);
 if (session == null) {
 return "home";
 }

 Member loginMember = (Member)
 session.getAttribute(SessionConst.LOGIN_MEMBER);
 //세션에 회원 데이터가 없으면 home
 if (loginMember == null) {
 return "home";
 }

 //세션이 유지되면 로그인으로 이동
 model.addAttribute("member", loginMember);
 return "loginHome";
}
```

- 기존 `homeLoginV2()` 의 `@GetMapping("/")` 주석 처리
- `request.getSession(false)`: `request.getSession()` 를 사용하면 기본 값이 `create: true` 이므로, 로그인 하지 않을 사용자도 의미없는 세션이 만들어진다. 따라서 세션을 찾아서 사용하는 시점에는 `create: false` 옵션을 사용해서 세션을 생성하지 않아야 한다.
- `session.getAttribute(SessionConst.LOGIN_MEMBER)` : 로그인 시점에 세션에 보관한 회원 객체를 찾는다.

### 실행

`JSESSIONID` 쿠키가 적절하게 생성되는 것을 확인하자.

## 로그인 처리하기 - 서블릿 HTTP 세션2

### @SessionAttribute

스프링은 세션을 더 편리하게 사용할 수 있도록 `@SessionAttribute` 을 지원한다.

이미 로그인 된 사용자를 찾을 때는 다음과 같이 사용하면 된다. 참고로 이 기능은 세션을 생성하지 않는다.

```
@SessionAttribute(name = "loginMember", required = false) Member loginMember
```

### HomeController - homeLoginV3Spring()

```
@GetMapping("/")
public String homeLoginV3Spring(
 @SessionAttribute(name = SessionConst.LOGIN_MEMBER, required = false)
 Member loginMember,
 Model model) {

 //세션에 회원 데이터가 없으면 home
 if (loginMember == null) {
 return "home";
 }

 //세션이 유지되면 로그인으로 이동
 model.addAttribute("member", loginMember);
 return "loginHome";
}
```

- `homeLoginV3()` 의 `@GetMapping("/")` 주석 처리

세션을 찾고, 세션에 들어있는 데이터를 찾는 번거로운 과정을 스프링이 한번에 편리하게 처리해주는 것을 확인할 수 있다.

### TrackingModes

로그인을 처음 시도하면 URL이 다음과 같이 `jsessionid` 를 포함하고 있는 것을 확인할 수 있다.

```
http://localhost:8080/;jsessionid=F59911518B921DF62D09F0DF8F83F872
```

이것은 웹 브라우저가 쿠키를 지원하지 않을 때 쿠키 대신 URL을 통해서 세션을 유지하는 방법이다. 이 방법을 사용하려면 URL에 이 값을 계속 포함해서 전달해야 한다. 타임리프 같은 템플릿은 엔진을 통해서 링크를 걸면 `jsessionid`를 URL에 자동으로 포함해준다. 서버 입장에서 웹 브라우저가 쿠키를 지원하는지 하지 않는지 최초에는 판단하지 못하므로, 쿠키 값도 전달하고, URL에 `jsessionid`도 함께 전달한다.

URL 전달 방식을 끄고 항상 쿠키를 통해서만 세션을 유지하고 싶으면 다음 옵션을 넣어주면 된다. 이렇게 하면 URL에 `jsessionid`가 노출되지 않는다.

`application.properties`

```
server.servlet.session.tracking-modes=cookie
```

## 세션 정보와 타임아웃 설정

### 세션 정보 확인

세션이 제공하는 정보들을 확인해보자.

### **SessionInfoController**

```
package hello.login.web.session;

import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import java.util.Date;

@Slf4j
@RestController
public class SessionInfoController {
```

```

@GetMapping("/session-info")
public String sessionInfo(HttpServletRequest request) {

 HttpSession session = request.getSession(false);
 if (session == null) {
 return "세션이 없습니다.";
 }

 //세션 데이터 출력
 session.getAttributeNames().asIterator()
 .forEachRemaining(name -> log.info("session name={}, value={}",
name, session.getAttribute(name)));

 log.info("sessionId={}", session.getId());
 log.info("maxInactiveInterval={}", session.getMaxInactiveInterval());
 log.info("creationTime={}", new Date(session.getCreationTime()));
 log.info("lastAccessedTime={}", new
Date(session.getLastAccessedTime()));

 log.info("isNew={}", session isNew);

 return "세션 출력";
}

}

```

- sessionId : 세션Id, JSESSIONID 의 값이다. 예) 34B14F008AA3527C9F8ED620EFD7A4E1
- maxInactiveInterval : 세션의 유효 시간, 예) 1800초, (30분)
- creationTime : 세션 생성일시
- lastAccessedTime : 세션과 연결된 사용자가 최근에 서버에 접근한 시간, 클라이언트에서 서버로 sessionId ( JSESSIONID )를 요청한 경우에 갱신된다.
- isNew : 새로 생성된 세션인지, 아니면 이미 과거에 만들어졌고, 클라이언트에서 서버로 sessionId ( JSESSIONID )를 요청해서 조회된 세션인지 여부

## 세션 타임아웃 설정

세션은 사용자가 로그아웃을 직접 호출해서 session.invalidate() 가 호출 되는 경우에 삭제된다.

그런데 대부분의 사용자는 로그아웃을 선택하지 않고, 그냥 웹 브라우저를 종료한다. 문제는 HTTP가 비연결성(ConnectionLess)이므로 서버 입장에서는 해당 사용자가 웹 브라우저를 종료한 것인지 아닌지를

인식할 수 없다. 따라서 서버에서 세션 데이터를 언제 삭제해야 하는지 판단하기가 어렵다.

이 경우 남아있는 세션을 무한정 보관하면 다음과 같은 문제가 발생할 수 있다.

- 세션과 관련된 쿠키(`JSESSIONID`)를 탈취 당했을 경우 오랜 시간이 지나도 해당 쿠키로 악의적인 요청을 할 수 있다.
- 세션은 기본적으로 메모리에 생성된다. 메모리의 크기가 무한하지 않기 때문에 꼭 필요한 경우만 생성해서 사용해야 한다. 10만명의 사용자가 로그인하면 10만개의 세션이 생성되는 것이다.

### 세션의 종료 시점

세션의 종료 시점을 어떻게 정하면 좋을까? 가장 단순하게 생각해보면, 세션 생성 시점으로부터 30분 정도로 잡으면 될 것 같다. 그런데 문제는 30분이 지나면 세션이 삭제되기 때문에, 열심히 사이트를 돌아다니다가 또 로그인을 해서 세션을 생성해야 한다 그러니까 30분마다 계속 로그인해야 하는 번거로움이 발생한다.

더 나은 대안은 세션 생성 시점이 아니라 사용자가 서버에 최근에 요청한 시간을 기준으로 30분 정도를 유지해주는 것이다. 이렇게 하면 사용자가 서비스를 사용하고 있으면, 세션의 생존 시간이 30분으로 계속 늘어나게 된다. 따라서 30분마다 로그인해야 하는 번거로움이 사라진다. `HttpSession`은 이 방식을 사용한다.

### 세션 타임아웃 설정

스프링 부트로 글로벌 설정

```
application.properties
server.servlet.session.timeout=60 : 60초, 기본은 1800(30분)
(글로벌 설정은 분 단위로 설정해야 한다. 60(1분), 120(2분), ...)
```

특정 세션 단위로 시간 설정

```
session.setMaxInactiveInterval(1800); //1800초
```

### 세션 타임아웃 발생

세션의 타임아웃 시간은 해당 세션과 관련된 `JSESSIONID`를 전달하는 HTTP 요청이 있으면 현재 시간으로 다시 초기화 된다. 이렇게 초기화 되면 세션 타임아웃으로 설정한 시간동안 세션을 추가로 사용할 수 있다.

```
session.getLastAccessedTime() : 최근 세션 접근 시간
```

`LastAccessedTime` 이후로 `timeout` 시간이 지나면, WAS가 내부에서 해당 세션을 제거한다.

### 정리

서블릿의 `HttpSession`이 제공하는 타임아웃 기능 덕분에 세션을 안전하고 편리하게 사용할 수 있다.

실무에서 주의할 점은 세션에는 최소한의 데이터만 보관해야 한다는 점이다. 보관한 데이터 용량 \* 사용자 수로 세션의 메모리 사용량이 급격하게 늘어나서 장애로 이어질 수 있다. 추가로 세션의 시간을 너무 길게 가져가면 메모리 사용이 계속 누적 될 수 있으므로 적당한 시간을 선택하는 것이 필요하다. 기본이 30분이라는 것을 기준으로 고민하면 된다.

## 정리

## 7. 로그인 처리2 - 필터, 인터셉터

#인강/5. 스프링 MVC 2/강의#

- 7. 로그인 처리2 - 필터, 인터셉터 - 서블릿 필터 - 소개
- 7. 로그인 처리2 - 필터, 인터셉터 - 서블릿 필터 - 요청 로그
- 7. 로그인 처리2 - 필터, 인터셉터 - 서블릿 필터 - 인증 체크
- 7. 로그인 처리2 - 필터, 인터셉터 - 스프링 인터셉터 - 소개
- 7. 로그인 처리2 - 필터, 인터셉터 - 스프링 인터셉터 - 요청 로그
- 7. 로그인 처리2 - 필터, 인터셉터 - 스프링 인터셉터 - 인증 체크
- 7. 로그인 처리2 - 필터, 인터셉터 - ArgumentResolver 활용
- 7. 로그인 처리2 - 필터, 인터셉터 - 정리

### 서블릿 필터 - 소개

#### 공통 관심 사항

요구사항을 보면 로그인 한 사용자만 상품 관리 페이지에 들어갈 수 있어야 한다.

앞에서 로그인을 하지 않은 사용자에게는 상품 관리 버튼이 보이지 않기 때문에 문제가 없어 보인다. 그런데 문제는 로그인 하지 않은 사용자도 다음 URL을 직접 호출하면 상품 관리 화면에 들어갈 수 있다는 점이다.

- <http://localhost:8080/items>

상품 관리 컨트롤러에서 로그인 여부를 체크하는 로직을 하나하나 작성하면 되겠지만, 등록, 수정, 삭제, 조회 등등 상품관리의 모든 컨트롤러 로직에 공통으로 로그인 여부를 확인해야 한다. 더 큰 문제는 향후 로그인과 관련된 로직이 변경될 때 이다. 작성한 모든 로직을 다 수정해야 할 수 있다.

이렇게 애플리케이션 여러 로직에서 공통으로 관심이 있는 있는 것을 공통 관심사(cross-cutting concern)라고 한다. 여기서는 등록, 수정, 삭제, 조회 등등 여러 로직에서 공통으로 인증에 대해서 관심을 가지고 있다.

이러한 공통 관심사는 스프링의 AOP로도 해결할 수 있지만, 웹과 관련된 공통 관심사는 지금부터 설명할 서블릿 필터 또는 스프링 인터셉터를 사용하는 것이 좋다. 웹과 관련된 공통 관심사를 처리할 때는 HTTP의 헤더나 URL의 정보들이 필요한데, 서블릿 필터나 스프링 인터셉터는 `HttpServletRequest`를 제공한다.

#### 서블릿 필터 소개

필터는 서블릿이 지원하는 수문장이다. 필터의 특성은 다음과 같다.

## 필터 흐름

```
HTTP 요청 -> WAS -> 필터 -> 서블릿 -> 컨트롤러
```

필터를 적용하면 필터가 호출 된 다음에 서블릿이 호출된다. 그래서 모든 고객의 요청 로그를 남기는 요구사항이 있다면 필터를 사용하면 된다. 참고로 필터는 특정 URL 패턴에 적용할 수 있다. `/*` 이라고 하면 모든 요청에 필터가 적용된다.

참고로 스프링을 사용하는 경우 여기서 말하는 서블릿은 스프링의 디스패처 서블릿으로 생각하면 된다.

## 필터 제한

```
HTTP 요청 -> WAS -> 필터 -> 서블릿 -> 컨트롤러 //로그인 사용자
```

```
HTTP 요청 -> WAS -> 필터(적절하지 않은 요청이라 판단, 서블릿 호출X) //비 로그인 사용자
```

필터에서 적절하지 않은 요청이라고 판단하면 거기에서 끝을 낼 수도 있다. 그래서 로그인 여부를 체크하기에 딱 좋다.

## 필터 체인

```
HTTP 요청 -> WAS -> 필터1 -> 필터2 -> 필터3 -> 서블릿 -> 컨트롤러
```

필터는 체인으로 구성되는데, 중간에 필터를 자유롭게 추가할 수 있다. 예를 들어서 로그를 남기는 필터를 먼저 적용하고, 그 다음에 로그인 여부를 체크하는 필터를 만들 수 있다.

## 필터 인터페이스

```
public interface Filter {

 public default void init(FilterConfig filterConfig) throws ServletException
 {}

 public void doFilter(ServletRequest request, ServletResponse response,
 FilterChain chain) throws IOException, ServletException;

 public default void destroy() {}
}
```

필터 인터페이스를 구현하고 등록하면 서블릿 컨테이너가 필터를 싱글톤 객체로 생성하고, 관리한다.

- `init()`: 필터 초기화 메서드, 서블릿 컨테이너가 생성될 때 호출된다.
- `doFilter()`: 고객의 요청이 올 때마다 해당 메서드가 호출된다. 필터의 로직을 구현하면 된다.
- `destroy()`: 필터 종료 메서드, 서블릿 컨테이너가 종료될 때 호출된다.

## 서블릿 필터 - 요청 로그

필터가 정말 수문장 역할을 잘 하는지 확인하기 위해 가장 단순한 필터인, 모든 요청을 로그로 남기는 필터를 개발하고 적용해보자.

### LogFilter - 로그 필터

```
package hello.login.web.filter;

import lombok.extern.slf4j.Slf4j;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
import java.util.UUID;

@Slf4j
public class LogFilter implements Filter {

 @Override
 public void init(FilterConfig filterConfig) throws ServletException {
 log.info("log filter init");
 }

 @Override
 public void doFilter(ServletRequest request, ServletResponse response,
 FilterChain chain) throws IOException, ServletException {
 HttpServletRequest httpRequest = (HttpServletRequest) request;
 String requestURI = httpRequest.getRequestURI();

 String uuid = UUID.randomUUID().toString();
 log.info("Request URI: " + requestURI + ", UUID: " + uuid);
 }
}
```

```

 try {
 log.info("REQUEST [{}][{}]", uuid, requestURI);
 chain.doFilter(request, response);
 } catch (Exception e) {
 throw e;
 } finally {
 log.info("RESPONSE [{}][{}]", uuid, requestURI);
 }
 }

 @Override
 public void destroy() {
 log.info("log filter destroy");
 }
}

```

- `public class LogFilter implements Filter {}`
  - 필터를 사용하려면 필터 인터페이스를 구현해야 한다.
- `doFilter(ServletRequest request, ServletResponse response, FilterChain chain)`
  - HTTP 요청이 오면 `doFilter` 가 호출된다.
  - `ServletRequest request` 는 HTTP 요청이 아닌 경우까지 고려해서 만든 인터페이스이다. HTTP를 사용하면 `HttpServletRequest httpRequest = (HttpServletRequest) request;` 와 같이 다운 캐스팅 하면 된다.
- `String uuid = UUID.randomUUID().toString();`
  - HTTP 요청을 구분하기 위해 요청당 임의의 `uuid` 를 생성해둔다.
- `log.info("REQUEST [{}][{}]", uuid, requestURI);`
  - `uuid` 와 `requestURI` 를 출력한다.
- `chain.doFilter(request, response);`
  - 이 부분이 가장 중요하다. 다음 필터가 있으면 필터를 호출하고, 필터가 없으면 서블릿을 호출한다. 만약 이 로직을 호출하지 않으면 다음 단계로 진행되지 않는다.

## WebConfig - 필터 설정

```
package hello.login;
```

```

import hello.login.web.filter.LogFilter;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.servlet.Filter;

@Configuration
public class WebConfig {

 @Bean
 public FilterRegistrationBean logFilter() {
 FilterRegistrationBean<Filter> filterRegistrationBean = new
 FilterRegistrationBean<>();
 filterRegistrationBean.setFilter(new LogFilter());
 filterRegistrationBean.setOrder(1);
 filterRegistrationBean.addUrlPatterns("/*");
 return filterRegistrationBean;
 }

}

```

필터를 등록하는 방법은 여러가지가 있지만, 스프링 부트를 사용한다면 `FilterRegistrationBean`을 사용해서 등록하면 된다.

- `setFilter(new LogFilter())` : 등록할 필터를 지정한다.
- `setOrder(1)` : 필터는 체인으로 동작한다. 따라서 순서가 필요하다. 낮을 수록 먼저 동작한다.
- `addUrlPatterns("/*")` : 필터를 적용할 URL 패턴을 지정한다. 한번에 여러 패턴을 지정할 수 있다.

## 참고

URL 패턴에 대한 룰은 필터도 서블릿과 동일하다. 자세한 내용은 서블릿 URL 패턴으로 검색해보자.

## 참고

`@ServletComponentScan @WebFilter(filterName = "logFilter", urlPatterns = "/*")`로 필터 등록이 가능하지만 필터 순서 조절이 안된다. 따라서 `FilterRegistrationBean`을 사용하자.

## 실행 로그

```
hello.login.web.filter.LogFilter: REQUEST [0a2249f2-
```

```
cc70-4db4-98d1-492ccf5629dd] [/items]
hello.login.web.filter.LogFilter: RESPONSE [0a2249f2-
cc70-4db4-98d1-492ccf5629dd] [/items]
```

필터를 등록할 때 `urlPattern` 을 `/*`로 등록했기 때문에 모든 요청에 해당 필터가 적용된다.

## 참고

실무에서 HTTP 요청시 같은 요청의 로그에 모두 같은 식별자를 자동으로 남기는 방법은 logback mdc로 검색해보자.

## 서블릿 필터 - 인증 체크

드디어 인증 체크 필터를 개발해보자. 로그인 되지 않은 사용자는 상품 관리 뿐만 아니라 미래에 개발될 페이지에도 접근하지 못하도록 하자.

### LoginCheckFilter - 인증 체크 필터

```
package hello.login.web.filter;

import hello.login.web.SessionConst;
import lombok.extern.slf4j.Slf4j;
import org.springframework.util.PatternMatchUtils;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;

@Slf4j
public class LoginCheckFilter implements Filter {

 private static final String[] whitelist = {"/", "/members/add", "/login",
 "/logout", "/css/*"};

 @Override
```

```

public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
 HttpServletRequest httpRequest = (HttpServletRequest) request;
 String requestURI = httpRequest.getRequestURI();

 HttpServletResponse httpResponse = (HttpServletResponse) response;

 try {
 log.info("인증 체크 필터 시작 {}", requestURI);

 if (isLoginCheckPath(requestURI)) {
 log.info("인증 체크 로직 실행 {}", requestURI);
 HttpSession session = httpRequest.getSession(false);
 if (session == null ||
 session.getAttribute(SessionConst.LOGIN_MEMBER) == null) {
 log.info("미인증 사용자 요청 {}", requestURI);
 //로그인으로 redirect
 httpResponse.sendRedirect("/login?redirectURL=" +
requestURI);
 return; //여기가 중요, 미인증 사용자는 다음으로 진행하지 않고 끝!
 }
 }

 chain.doFilter(request, response);
 } catch (Exception e) {
 throw e; //예외 로깅 가능 하지만, 톰캣까지 예외를 보내주어야 함
 } finally {
 log.info("인증 체크 필터 종료 {}", requestURI);
 }
}

/**
 * 화이트 리스트의 경우 인증 체크X
 */
private boolean isLoginCheckPath(String requestURI) {
 return !PatternMatchUtils.simpleMatch(whitelist, requestURI);
}

```

```
}
```

- `whitelist = {"/", "/members/add", "/login", "/logout", "/css/*"};`
  - 인증 필터를 적용해도 홈, 회원가입, 로그인 화면, css 같은 리소스에는 접근할 수 있어야 한다. 이렇게 화이트 리스트 경로는 인증과 무관하게 항상 허용한다. 화이트 리스트를 제외한 나머지 모든 경로에는 인증 체크 로직을 적용한다.
- `isLoginCheckPath(requestURI)`
  - 화이트 리스트를 제외한 모든 경우에 인증 체크 로직을 적용한다.
- `httpServletResponse.sendRedirect("/login?redirectURL=" + requestURI);`
  - 미인증 사용자는 로그인 화면으로 리다이렉트 한다. 그런데 로그인 이후에 다시 홈으로 이동해버리면, 원하는 경로를 다시 찾아가야 하는 불편함이 있다. 예를 들어서 상품 관리 화면을 보려고 들어갔다가 로그인 화면으로 이동하면, 로그인 이후에 다시 상품 관리 화면으로 들어가는 것이 좋다. 이런 부분이 개발자 입장에서는 좀 귀찮을 수 있어도 사용자 입장으로 보면 편리한 기능이다. 이러한 기능을 위해 현재 요청한 경로인 `requestURI` 를 `/login`에 쿼리 파라미터로 함께 전달한다. 물론 `/login` 컨트롤러에서 로그인 성공시 해당 경로로 이동하는 기능은 추가로 개발해야 한다.
- `return;` 여기가 중요하다. 필터를 더는 진행하지 않는다. 이후 필터는 물론 서블릿, 컨트롤러가 더는 호출되지 않는다. 앞서 `redirect` 를 사용했기 때문에 `redirect` 가 응답으로 적용되고 요청이 끝난다.

## WebConfig - loginCheckFilter() 추가

```
@Bean
public FilterRegistrationBean loginCheckFilter() {
 FilterRegistrationBean<Filter> filterRegistrationBean = new
 FilterRegistrationBean<>();
 filterRegistrationBean.setFilter(new LoginCheckFilter());
 filterRegistrationBean.setOrder(2);
 filterRegistrationBean.addUrlPatterns("/*");
 return filterRegistrationBean;
}
```

- `setFilter(new LoginCheckFilter())` : 로그인 필터를 등록한다.
- `setOrder(2)` : 순서를 2번으로 잡았다. 로그 필터 다음에 로그인 필터가 적용된다.
- `addUrlPatterns("/*")` : 모든 요청에 로그인 필터를 적용한다.

## RedirectURL 처리

로그인에 성공하면 처음 요청한 URL로 이동하는 기능을 개발해보자.

### LoginController - loginV4()

```
/*
 * 로그인 이후 redirect 처리
 */

@PostMapping("/login")
public String loginV4(
 @Valid @ModelAttribute LoginForm form, BindingResult bindingResult,
 @RequestParam(defaultValue = "/") String redirectURL,
 HttpServletRequest request) {

 if (bindingResult.hasErrors()) {
 return "login/loginForm";
 }

 Member loginMember = loginService.login(form.getLoginId(),
 form.getPassword());
 log.info("login? {}", loginMember);

 if (loginMember == null) {
 bindingResult.reject("loginFail", "아이디 또는 비밀번호가 맞지 않습니다.");
 return "login/loginForm";
 }

 //로그인 성공 처리

 //세션이 있으면 있는 세션 반환, 없으면 신규 세션 생성
 HttpSession session = request.getSession();
 //세션에 로그인 회원 정보 보관
 session.setAttribute(SessionConst.LOGIN_MEMBER, loginMember);

 //redirectURL 적용
 return "redirect:" + redirectURL;
}
```

- loginV3() 의 @PostMapping("/login") 제거

- 로그인 체크 필터에서, 미인증 사용자는 요청 경로를 포함해서 `/login`에 `redirectURL` 요청 파라미터를 추가해서 요청했다. 이 값을 사용해서 로그인 성공시 해당 경로로 고객을 `redirect` 한다.

## 실행

`http://localhost:8080/items`

## 정리

서블릿 필터를 잘 사용한 덕분에 로그인 하지 않은 사용자는 나머지 경로에 들어갈 수 없게 되었다. 공통 관심사를 서블릿 필터를 사용해서 해결한 덕분에 향후 로그인 관련 정책이 변경되어도 이 부분만 변경하면 된다.

## 참고

필터에는 다음에 설명할 스프링 인터셉터는 제공하지 않는, 아주 강력한 기능이 있는데

`chain.doFilter(request, response);` 를 호출해서 다음 필터 또는 서블릿을 호출할 때 `request`, `response` 를 다른 객체로 바꿀 수 있다. `ServletRequest`, `ServletResponse` 를 구현한 다른 객체를 만들어서 넘기면 해당 객체가 다음 필터 또는 서블릿에서 사용된다. 잘 사용하는 기능은 아니니 참고만 해두자.

## 스프링 인터셉터 - 소개

스프링 인터셉터도 서블릿 필터와 같이 웹과 관련된 공통 관심 사항을 효과적으로 해결할 수 있는 기술이다. 서블릿 필터가 서블릿이 제공하는 기술이라면, 스프링 인터셉터는 스프링 MVC가 제공하는 기술이다. 둘다 웹과 관련된 공통 관심 사항을 처리하지만, 적용되는 순서와 범위, 그리고 사용방법이 다르다.

### 스프링 인터셉터 흐름

HTTP 요청 → WAS → 필터 → 서블릿 → 스프링 인터셉터 → 컨트롤러

- 스프링 인터셉터는 디스패처 서블릿과 컨트롤러 사이에서 컨트롤러 호출 직전에 호출 된다.
- 스프링 인터셉터는 스프링 MVC가 제공하는 기능이기 때문에 결국 디스패처 서블릿 이후에 등장하게 된다. 스프링 MVC의 시작점이 디스패처 서블릿이라고 생각해보면 이해가 될 것이다.
- 스프링 인터셉터에도 URL 패턴을 적용할 수 있는데, 서블릿 URL 패턴과는 다르고, 매우 정밀하게 설정할 수 있다.

## 스프링 인터셉터 제한

```
HTTP 요청 -> WAS -> 필터 -> 서블릿 -> 스프링 인터셉터 -> 컨트롤러 //로그인 사용자
HTTP 요청 -> WAS -> 필터 -> 서블릿 -> 스프링 인터셉터(적절하지 않은 요청이라 판단, 컨트롤러 호출
X) // 비 로그인 사용자
```

인터셉터에서 적절하지 않은 요청이라고 판단하면 거기에서 끝을 낼 수도 있다. 그래 로그인 여부를 체크하기에 딱 좋다.

## 스프링 인터셉터 체인

```
HTTP 요청 -> WAS -> 필터 -> 서블릿 -> 인터셉터1 -> 인터셉터2 -> 컨트롤러
```

스프링 인터셉터는 체인으로 구성되는데, 중간에 인터셉터를 자유롭게 추가할 수 있다. 예를 들어서 로그를 남기는 인터셉터를 먼저 적용하고, 그 다음에 로그인 여부를 체크하는 인터셉터를 만들 수 있다.

지금까지 내용을 보면 서블릿 필터와 호출 되는 순서만 다르고, 제공하는 기능은 비슷해 보인다. 앞으로 설명하겠지만, 스프링 인터셉터는 서블릿 필터보다 편리하고, 더 정교하고 다양한 기능을 지원한다.

## 스프링 인터셉터 인터페이스

스프링의 인터셉터를 사용하려면 `HandlerInterceptor` 인터페이스를 구현하면 된다.

```
public interface HandlerInterceptor {

 default boolean preHandle(HttpServletRequest request, HttpServletResponse
 response,
 Object handler) throws Exception {}

 default void postHandle(HttpServletRequest request, HttpServletResponse
 response,
 Object handler, @Nullable ModelAndView modelAndView)
 throws Exception {}

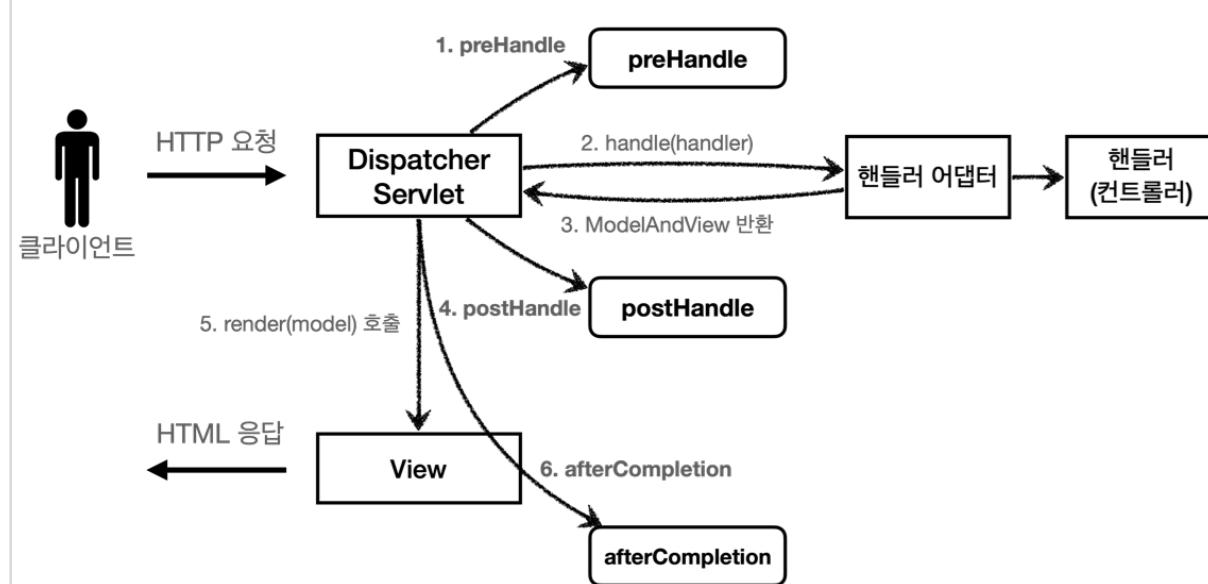
 default void afterCompletion(HttpServletRequest request, HttpServletResponse
 response,
 Object handler, @Nullable Exception ex) throws
 Exception {}
```

}

- 서블릿 필터의 경우 단순하게 `doFilter()` 하나만 제공된다. 인터셉터는 컨트롤러 호출 전(`preHandle`), 호출 후(`postHandle`), 요청 완료 이후(`afterCompletion`)와 같이 단계적으로 잘 세분화 되어 있다.
- 서블릿 필터의 경우 단순히 `request`, `response`만 제공했지만, 인터셉터는 어떤 컨트롤러(`handler`)가 호출되는지 호출 정보도 받을 수 있다. 그리고 어떤 `modelAndView`가 반환되는지 응답 정보도 받을 수 있다.

### 스프링 인터셉터 호출 흐름

#### 스프링 인터셉터

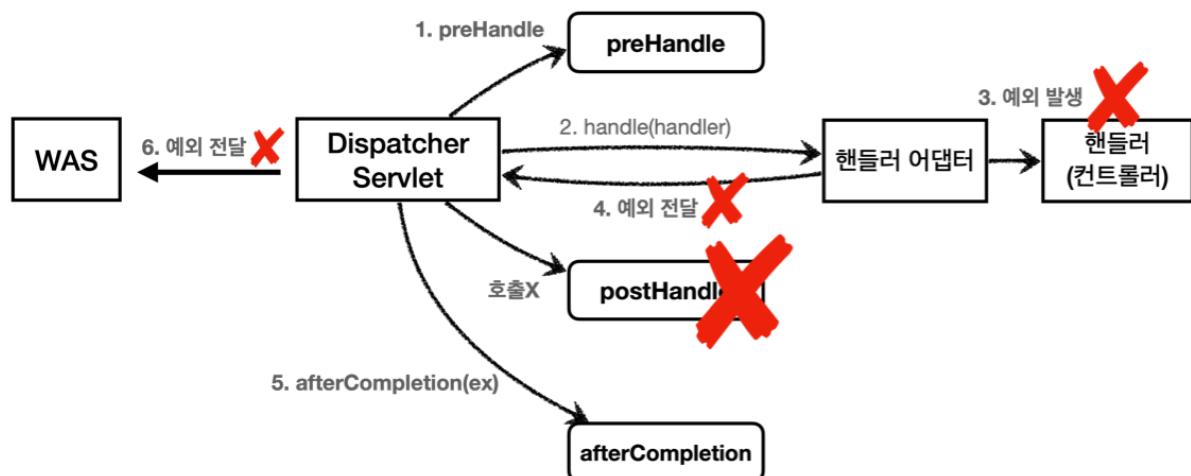


### 정상 흐름

- `preHandle` : 컨트롤러 호출 전에 호출된다. (더 정확히는 핸들러 어댑터 호출 전에 호출된다.)
  - `preHandle`의 응답값이 `true`이면 다음으로 진행하고, `false`이면 더는 진행하지 않는다. `false`인 경우 나머지 인터셉터는 물론이고, 핸들러 어댑터도 호출되지 않는다. 그림에서 1번에서 끝이 나버린다.
- `postHandle` : 컨트롤러 호출 후에 호출된다. (더 정확히는 핸들러 어댑터 호출 후에 호출된다.)
- `afterCompletion` : 뷰가 렌더링 된 이후에 호출된다.

### 스프링 인터셉터 예외 상황

## 스프링 인터셉터 예외



### 예외가 발생시

- **preHandle** : 컨트롤러 호출 전에 호출된다.
- **postHandle** : 컨트롤러에서 예외가 발생하면 **postHandle**은 호출되지 않는다.
- **afterCompletion** : **afterCompletion**은 항상 호출된다. 이 경우 예외( **ex** )를 파라미터로 받아서 어떤 예외가 발생했는지 로그로 출력할 수 있다.

### **afterCompletion**은 예외가 발생해도 호출된다.

- 예외가 발생하면 **postHandle()**는 호출되지 않으므로 예외와 무관하게 공통 처리를 하려면 **afterCompletion()**을 사용해야 한다.
- 예외가 발생하면 **afterCompletion()**에 예외 정보( **ex** )를 포함해서 호출된다.

### 정리

인터셉터는 스프링 MVC 구조에 특화된 필터 기능을 제공한다고 이해하면 된다. 스프링 MVC를 사용하고, 특별히 필터를 꼭 사용해야 하는 상황이 아니라면 인터셉터를 사용하는 것이 더 편리하다.

## 스프링 인터셉터 - 요청 로그

### LogInterceptor - 요청 로그 인터셉터

```
package hello.login.web.interceptor;

import lombok.extern.slf4j.Slf4j;
import org.springframework.web.method.HandlerMethod;
```

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

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.UUID;

@Slf4j
public class LogInterceptor implements HandlerInterceptor {

 public static final String LOG_ID = "logId";

 @Override
 public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {

 String requestURI = request.getRequestURI();

 String uuid = UUID.randomUUID().toString();
 request.setAttribute(LOG_ID, uuid);

 // @RequestMapping: HandlerMethod
 // 정적 리소스: ResourceHttpRequestHandler
 if (handler instanceof HandlerMethod) {
 HandlerMethod hm = (HandlerMethod) handler; // 호출할 컨트롤러 메서드의
모든 정보가 포함되어 있다.

 }

 log.info("REQUEST [{}][{}][{}]", uuid, requestURI, handler);
 return true; // false 진행X
 }

 @Override
 public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
 log.info("postHandle {}", modelAndView);
 }

 @Override
```

```

public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
 String requestURI = request.getRequestURI();
 String logId = (String)request.getAttribute(LOG_ID);
 log.info("RESPONSE [{}][{}]", logId, requestURI);
 if (ex != null) {
 log.error("afterCompletion error!!", ex);
 }
}
}

```

- `String uuid = UUID.randomUUID().toString()`
  - 요청 로그를 구분하기 위한 `uuid`를 생성한다.
- `request.setAttribute(LOG_ID, uuid)`
  - 서블릿 필터의 경우 지역변수로 해결이 가능하지만, 스프링 인터셉터는 호출 시점이 완전히 분리되어 있다. 따라서 `preHandle`에서 지정한 값을 `postHandle`, `afterCompletion`에서 함께 사용하려면 어딘가에 담아두어야 한다. `LogInterceptor`도 싱글톤처럼 사용되기 때문에 맴버변수를 사용하면 위험하다. 따라서 `request`에 담아두었다. 이 값은 `afterCompletion`에서 `request.getAttribute(LOG_ID)`로 찾아서 사용한다.
- `return true`
  - `true`면 정상 호출이다. 다음 인터셉터나 컨트롤러가 호출된다.

```

if (handler instanceof HandlerMethod) {
 HandlerMethod hm = (HandlerMethod) handler; //호출할 컨트롤러 메서드의 모든 정보가
 포함되어 있다.
}

```

## HandlerMethod

핸들러 정보는 어떤 핸들러 매핑을 사용하는가에 따라 달라진다. 스프링을 사용하면 일반적으로 `@Controller`, `@RequestMapping`을 활용한 핸들러 매핑을 사용하는데, 이 경우 핸들러 정보로 `HandlerMethod` 가 넘어온다.

## ResourceHttpRequestHandler

`@Controller` 가 아니라 `/resources/static`와 같은 정적 리소스가 호출 되는 경우 `ResourceHttpRequestHandler` 가 핸들러 정보로 넘어오기 때문에 타입에 따라서 처리가 필요하다.

## postHandle, afterCompletion

종료 로그를 `postHandle` 이 아니라 `afterCompletion`에서 실행한 이유는, 예외가 발생한 경우 `postHandle` 가 호출되지 않기 때문이다. `afterCompletion`은 예외가 발생해도 호출 되는 것을 보장한다.

## WebConfig - 인터셉터 등록

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

 @Override
 public void addInterceptors(InterceptorRegistry registry) {
 registry.addInterceptor(new LogInterceptor())
 .order(1)
 .addPathPatterns("/**")
 .excludePathPatterns("/css/**", "/*.ico", "/error");
 }
 // ...
}
```

인터셉터와 필터가 중복되지 않도록 필터를 등록하기 위한 `logFilter()` 의 `@Bean` 은 주석처리하자.

`WebMvcConfigurer` 가 제공하는 `addInterceptors()` 를 사용해서 인터셉터를 등록할 수 있다.

- `registry.addInterceptor(new LogInterceptor())` : 인터셉터를 등록한다.
- `order(1)` : 인터셉터의 호출 순서를 지정한다. 낮을 수록 먼저 호출된다.
- `addPathPatterns("/**")` : 인터셉터를 적용할 URL 패턴을 지정한다.
- `excludePathPatterns("/css/**", "/*.ico", "/error")` : 인터셉터에서 제외할 패턴을 지정한다.

필터와 비교해보면 인터셉터는 `addPathPatterns`, `excludePathPatterns` 로 매우 정밀하게 URL 패턴을 지정할 수 있다.

## 실행 로그

```
REQUEST [6234a913-f24f-461f-a9e1-85f153b3c8b2] [/members/add]
[hello.login.web.member.MemberController#addForm(Member)]

postHandle [ModelAndView [view="members/addMemberForm";
```

```
model={member=Member(id=null, loginId=null, name=null, password=null),
org.springframework.validation.BindingResult.member=org.springframework.validation.BeanPropertyBindingResult: 0 errors}]]
```

```
RESPONSE [6234a913-f24f-461f-a9e1-85f153b3c8b2] [/members/add]
```

## 스프링의 URL 경로

스프링이 제공하는 URL 경로는 서블릿 기술이 제공하는 URL 경로와 완전히 다르다. 더욱 자세하고, 세밀하게 설정할 수 있다.

자세한 내용은 다음을 참고하자.

## PathPattern 공식 문서

```
? 한 문자 일치
* 경로(/) 안에서 0개 이상의 문자 일치
** 경로 끝까지 0개 이상의 경로(/) 일치
{spring} 경로(/)와 일치하고 spring이라는 변수로 캡처
{spring:[a-z]+} matches the regexp [a-z]+ as a path variable named "spring"
{spring:[a-z]+} regexp [a-z]+ 와 일치하고, "spring" 경로 변수로 캡처
{*spring} 경로가 끝날 때 까지 0개 이상의 경로(/)와 일치하고 spring이라는 변수로 캡처

/pages/t?st.html – matches /pages/test.html, /pages/tXst.html but not /pages/toast.html
/resources/*.png – matches all .png files in the resources directory
/resources/** – matches all files underneath the /resources/ path, including /resources/image.png and /resources/css/spring.css
/resources/{*path} – matches all files underneath the /resources/ path and captures their relative path in a variable named "path"; /resources/image.png will match with "path" → "/image.png", and /resources/css/spring.css will match with "path" → "/css/spring.css"
/resources/{filename:\w+.dat will match /resources/spring.dat and assign the value "spring" to the filename variable
```

링크: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/util/pattern/PathPattern.html>

## 스프링 인터셉터 - 인증 체크

서블릿 필터에서 사용했던 인증 체크 기능을 스프링 인터셉터로 개발해보자.

### LoginCheckInterceptor

```
package hello.login.web.interceptor;

import hello.login.web.SessionConst;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

@Slf4j
public class LoginCheckInterceptor implements HandlerInterceptor {

 @Override
 public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

 String requestURI = request.getRequestURI();

 log.info("인증 체크 인터셉터 실행 {}", requestURI);
 HttpSession session = request.getSession(false);

 if (session == null || session.getAttribute(SessionConst.LOGIN_MEMBER) == null) {
 log.info("미인증 사용자 요청");
 //로그인으로 redirect
 response.sendRedirect("/login?redirectURL=" + requestURI);
 return false;
 }
 }
}
```

```

 return true;
 }

}

```

서블릿 필터와 비교해서 코드가 매우 간결하다. 인증이라는 것은 컨트롤러 호출 전에만 호출되면 된다.  
따라서 `preHandle` 만 구현하면 된다.

### 순서 주의, 세밀한 설정 가능

```

@Configuration
public class WebConfig implements WebMvcConfigurer {

 @Override
 public void addInterceptors(InterceptorRegistry registry) {
 registry.addInterceptor(new LogInterceptor())
 .order(1)
 .addPathPatterns("/**")
 .excludePathPatterns("/css/**", "/*.ico", "/error");

 registry.addInterceptor(new LoginCheckInterceptor())
 .order(2)
 .addPathPatterns("/**")
 .excludePathPatterns(
 "/", "/members/add", "/login", "/logout",
 "/css/**", "/*.ico", "/error"
);
 }

 // ...
}

```

인터셉터와 필터가 중복되지 않도록 필터를 등록하기 위한 `logFilter()`, `loginCheckFilter()` 의  
`@Bean` 은 주석처리하자.

인터셉터를 적용하거나 하지 않을 부분은 `addPathPatterns` 와 `excludePathPatterns` 에 작성하면 된다. 기본적으로 모든 경로에 해당 인터셉터를 적용하되 (`/**`), 홈(`/`), 회원가입(`/members/add`),  
로그인(`/login`), 리소스 조회(`/css/**`), 오류(`/error`)와 같은 부분은 로그인 체크 인터셉터를

적용하지 않는다. 서블릿 필터와 비교해보면 매우 편리한 것을 알 수 있다.

## 정리

서블릿 필터와 스프링 인터셉터는 웹과 관련된 공통 관심사를 해결하기 위한 기술이다.

서블릿 필터와 비교해서 스프링 인터셉터가 개발자 입장에서 훨씬 편리하다는 것을 코드로 이해했을 것이다. 특별한 문제가 없다면 인터셉터를 사용하는 것이 좋다.

## ArgumentResolver 활용

MVC1편 **6. 스프링 MVC - 기본 기능** → 요청 매핑 헤더 어댑터 구조에서 **ArgumentResolver** 를 학습했다.

이번 시간에는 해당 기능을 사용해서 로그인 회원을 조금 편리하게 찾아보자.

### HomeController - 추가

```
@GetMapping("/")
public String homeLoginV3ArgumentResolver(@Login Member loginMember, Model
model) {

 //세션에 회원 데이터가 없으면 home
 if (loginMember == null) {
 return "home";
 }

 //세션이 유지되면 로그인으로 이동
 model.addAttribute("member", loginMember);
 return "loginHome";
}
```

- `homeLoginV3Spring()` 의 `@GetMapping` 주석 처리
- 다음에 설명하는 `@Login` 애노테이션을 만들어야 컴파일 오류가 사라진다.

`@Login` 애노테이션이 있으면 직접 만든 `ArgumentResolver` 가 동작해서 자동으로 세션에 있는 로그인 회원을 찾아주고, 만약 세션에 없다면 `null` 을 반환하도록 개발해보자.

## @Login 애노테이션 생성

```
package hello.login.web.argumentresolver;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface Login {
}
```

- `@Target(ElementType.PARAMETER)` : 파라미터에만 사용
- `@Retention(RetentionPolicy.RUNTIME)` : 리플렉션 등을 활용할 수 있도록 런타임까지 애노테이션 정보가 남아있음

MVC1에서 학습한 `HandlerMethodArgumentResolver`를 구현해보자.

## LoginMemberArgumentResolver 생성

```
package hello.login.web.argumentresolver;

import hello.login.domain.member.Member;
import hello.login.web.SessionConst;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.MethodParameter;
import org.springframework.web.bind.support.WebDataBinderFactory;
import org.springframework.web.context.request.NativeWebRequest;
import org.springframework.web.method.support.HandlerMethodArgumentResolver;
import org.springframework.web.method.support.ModelAndViewContainer;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

@Slf4j
public class LoginMemberArgumentResolver implements
```

```

HandlerMethodArgumentResolver {

 @Override
 public boolean supportsParameter(MethodParameter parameter) {

 log.info("supportsParameter 실행");

 boolean hasLoginAnnotation =
parameter.hasParameterAnnotation(Login.class);
 boolean hasMemberType =
Member.class.isAssignableFrom(parameter.getParameterType()));

 return hasLoginAnnotation && hasMemberType;
 }

 @Override
 public Object resolveArgument(MethodParameter parameter,
ModelAndViewContainer mavContainer, NativeWebRequest webRequest,
WebDataBinderFactory binderFactory) throws Exception {

 log.info("resolveArgument 실행");

 HttpServletRequest request = (HttpServletRequest)
webRequest.getNativeRequest();
 HttpSession session = request.getSession(false);
 if (session == null) {
 return null;
 }

 return session.getAttribute(SessionConst.LOGIN_MEMBER);
 }
}

```

- `supportsParameter()` : `@Login` 애노테이션이 있으면서 `Member` 타입이면 해당 `ArgumentResolver` 가 사용된다.
- `resolveArgument()` : 컨트롤러 호출 직전에 호출 되어서 필요한 파라미터 정보를 생성해준다. 여기서는 세션에 있는 로그인 회원 정보인 `member` 객체를 찾아서 반환해준다. 이후 스프링MVC는 컨트롤러의 메서드를 호출하면서 여기에서 반환된 `member` 객체를 파라미터에 전달해준다.

## WebMvcConfigurer에 설정 추가

```
package hello.login;

import hello.login.web.argumentresolver.LoginMemberArgumentResolver;
import hello.login.web.filter.LogFilter;
import hello.login.web.filter.LoginCheckFilter;
import hello.login.web.interceptor.LogInterceptor;
import hello.login.web.interceptor.LoginCheckInterceptor;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.method.support.HandlerMethodArgumentResolver;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import javax.servlet.Filter;
import java.util.List;

@Configuration
public class WebConfig implements WebMvcConfigurer {

 @Override
 public void addArgumentResolvers(List<HandlerMethodArgumentResolver>
resolvers) {
 resolvers.add(new LoginMemberArgumentResolver());
 }
 //...
}
```

앞서 개발한 `LoginMemberArgumentResolver`를 등록하자.

### 실행

실행해보면, 결과는 동일하지만, 더 편리하게 로그인 회원 정보를 조회할 수 있다. 이렇게

`ArgumentResolver`를 활용하면 공통 작업이 필요할 때 컨트롤러를 더욱 편리하게 사용할 수 있다.

정리

## 8. 예외 처리와 오류 페이지

#인강/5. 스프링 MVC 2/강의#

### 목차

- 8. 예외 처리와 오류 페이지 - 프로젝트 생성
- 8. 예외 처리와 오류 페이지 - 서블릿 예외 처리 - 시작
- 8. 예외 처리와 오류 페이지 - 서블릿 예외 처리 - 오류 화면 제공
- 8. 예외 처리와 오류 페이지 - 서블릿 예외 처리 - 오류 페이지 작동 원리
- 8. 예외 처리와 오류 페이지 - 서블릿 예외 처리 - 필터
- 8. 예외 처리와 오류 페이지 - 서블릿 예외 처리 - 인터셉터
- 8. 예외 처리와 오류 페이지 - 스프링 부트 - 오류 페이지1
- 8. 예외 처리와 오류 페이지 - 스프링 부트 - 오류 페이지2
- 8. 예외 처리와 오류 페이지 - 정리

### 프로젝트 생성

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

<https://start.spring.io>

- 프로젝트 선택
  - Project: Gradle Project
  - Language: Java
  - Spring Boot: 2.5.x
- Project Metadata
  - Group: hello
  - Artifact: exception
  - Name: exception
  - Package name: **hello.exception**
  - Packaging: **Jar**
  - Java: 11

주의: 강의 영상에서 package 선택시 War라고 잘못 이야기했는데, Jar가 맞습니다.

- Dependencies: **Spring Web, Lombok , Thymeleaf, Validation**

### build.gradle

```

plugins {
 id 'org.springframework.boot' version '2.5.1'
 id 'io.spring.dependency-management' version '1.0.11.RELEASE'
 id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
 compileOnly {
 extendsFrom annotationProcessor
 }
}

repositories {
 mavenCentral()
}

dependencies {
 implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
 implementation 'org.springframework.boot:spring-boot-starter-validation'
 implementation 'org.springframework.boot:spring-boot-starter-web'
 compileOnly 'org.projectlombok:lombok'
 annotationProcessor 'org.projectlombok:lombok'
 testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
 useJUnitPlatform()
}

```

- 동작 확인
  - 기본 메인 클래스 실행( `ExceptionApplication.main()` )
  - `http://localhost:8080` 호출해서 Whitelabel Error Page가 나오면 정상 동작

## 서블릿 예외 처리 - 시작

스프링이 아닌 순수 서블릿 컨테이너는 예외를 어떻게 처리하는지 알아보자.

서블릿은 다음 **2가지** 방식으로 예외 처리를 지원한다.

- `Exception`(예외)
- `response.sendError(HTTP 상태 코드, 오류 메시지)`

### Exception(예외)

#### 자바 직접 실행

자바의 메인 메서드를 직접 실행하는 경우 `main`이라는 이름의 쓰레드가 실행된다.

실행 도중에 예외를 잡지 못하고 처음 실행한 `main()` 메서드를 넘어서 예외가 던져지면, 예외 정보를 남기고 해당 쓰레드는 종료된다.

#### 웹 애플리케이션

웹 애플리케이션은 사용자 요청별로 별도의 쓰레드가 할당되고, 서블릿 컨테이너 안에서 실행된다.

애플리케이션에서 예외가 발생했는데, 어디선가 `try ~ catch`로 예외를 잡아서 처리하면 아무런 문제가 없다. 그런데 만약에 애플리케이션에서 예외를 잡지 못하고, 서블릿 밖으로 까지 예외가 전달되면 어떻게 동작할까?

WAS(여기까지 전파) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러(예외발생)

결국 톰캣 같은 WAS 까지 예외가 전달된다. WAS는 예외가 올라오면 어떻게 처리해야 할까?

한번 테스트 해보자.

먼저 스프링 부트가 제공하는 기본 예외 페이지가 있는데 이건 꺼두자(뒤에서 다시 설명하겠다.)

#### application.properties

```
server.error.whitelabel.enabled=false
```

## ServletExController - 서블릿 예외 컨트롤러

```
package hello.exception.servlet;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Slf4j
@Controller
public class ServletExController {

 @GetMapping("/error-ex")
 public void errorEx() {
 throw new RuntimeException("예외 발생!");
 }
}
```

실행해보면 다음처럼 tomcat이 기본으로 제공하는 오류 화면을 볼 수 있다.

HTTP Status 500 – Internal Server Error

웹 브라우저에서 개발자 모드로 확인해보면 HTTP 상태 코드가 500으로 보인다.

Exception의 경우 서버 내부에서 처리할 수 없는 오류가 발생한 것으로 생각해서 HTTP 상태 코드 500을 반환한다.

이번에는 아무사이트나 호출해보자.

`http://localhost:8080/no-page`

HTTP Status 404 – Not Found

톰캣이 기본으로 제공하는 404 오류 화면을 볼 수 있다.

## response.sendError(HTTP 상태 코드, 오류 메시지)

오류가 발생했을 때 `HttpServletResponse` 가 제공하는 `sendError`라는 메서드를 사용해도 된다. 이것을 호출한다고 당장 예외가 발생하는 것은 아니지만, 서블릿 컨테이너에게 오류가 발생했다는 점을 전달할 수 있다.

이 메서드를 사용하면 HTTP 상태 코드와 오류 메시지도 추가할 수 있다.

- `response.sendError(HTTP 상태 코드)`
- `response.sendError(HTTP 상태 코드, 오류 메시지)`

## ServletExController - 추가

```
@GetMapping("/error-404")
public void error404(HttpServletRequest response) throws IOException {
 response.sendError(404, "404 오류!");
}

@GetMapping("/error-500")
public void error500(HttpServletRequest response) throws IOException {
 response.sendError(500);
}
```

## sendError 흐름

```
WAS(sendError 호출 기록 확인) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러
(response.sendError())
```

`response.sendError()` 를 호출하면 `response` 내부에는 오류가 발생했다는 상태를 저장해둔다. 그리고 서블릿 컨테이너는 고객에게 응답 전에 `response` 에 `sendError()` 가 호출되었는지 확인한다. 그리고 호출되었다면 설정한 오류 코드에 맞추어 기본 오류 페이지를 보여준다.

실행해보면 다음처럼 서블릿 컨테이너가 기본으로 제공하는 오류 화면을 볼 수 있다.

- <http://localhost:8080/error-ex>
- <http://localhost:8080/error-404>

- <http://localhost:8080/error-500>

```
HTTP Status 404 – Bad Request
HTTP Status 500 – Internal Server Error
```

## 정리

서블릿 컨테이너가 제공하는 기본 예외 처리 화면은 사용자가 보기 좋다. 의미 있는 오류 화면을 제공해보자.

## 서블릿 예외 처리 - 오류 화면 제공

서블릿 컨테이너가 제공하는 기본 예외 처리 화면은 고객 친화적이지 않다. 서블릿이 제공하는 오류 화면 기능을 사용해보자.

서블릿은 `Exception` (예외)가 발생해서 서블릿 밖으로 전달되거나 또는 `response.sendError()` 가 호출되었을 때 각각의 상황에 맞춘 오류 처리 기능을 제공한다.

이 기능을 사용하면 친절한 오류 처리 화면을 준비해서 고객에게 보여줄 수 있다.

과거에는 `web.xml`이라는 파일에 다음과 같이 오류 화면을 등록했다.

```
<web-app>
 <error-page>
 <error-code>404</error-code>
 <location>/error-page/404.html</location>
 </error-page>
 <error-page>
 <error-code>500</error-code>
 <location>/error-page/500.html</location>
 </error-page>
 <error-page>
 <exception-type>java.lang.RuntimeException</exception-type>
 <location>/error-page/500.html</location>
 </error-page>
</web-app>
```

지금은 스프링 부트를 통해서 서블릿 컨테이너를 실행하기 때문에, 스프링 부트가 제공하는 기능을 사용해서 서블릿 오류 페이지를 등록하면 된다.

### 서블릿 오류 페이지 등록

```
package hello.exception;

import org.springframework.boot.web.server.ConfigurableWebServerFactory;
import org.springframework.boot.web.server.ErrorPage;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;

@Component
public class WebServerCustomizer implements
WebServerFactoryCustomizer<ConfigurableWebServerFactory> {

 @Override
 public void customize(ConfigurableWebServerFactory factory) {

 ErrorPage errorPage404 = new ErrorPage(HttpStatus.NOT_FOUND, "/error-
page/404");
 ErrorPage errorPage500 = new
ErrorPage(HttpStatus.INTERNAL_SERVER_ERROR, "/error-page/500");
 ErrorPage errorPageEx = new ErrorPage(RuntimeException.class, "/error-
page/500");
 factory.addErrorPages(errorPage404, errorPage500, errorPageEx);
 }
}
```

- `response.sendError(404)` : `errorPage404` 호출
- `response.sendError(500)` : `errorPage500` 호출
- `RuntimeException` 또는 그 자식 타입의 예외: `errorPageEx` 호출

500 예외가 서버 내부에서 발생한 오류라는 뜻을 포함하고 있기 때문에 여기서는 예외가 발생한 경우도 500 오류 화면으로 처리했다.

오류 페이지는 예외를 다룰 때 해당 예외와 그 자식 타입의 오류를 함께 처리한다. 예를 들어서 위의 경우 `RuntimeException`은 물론이고 `RuntimeException`의 자식도 함께 처리한다.

오류가 발생했을 때 처리할 수 있는 컨트롤러가 필요하다. 예를 들어서 `RuntimeException` 예외가 발생하면 `errorPageEx`에서 지정한 `/error-page/500`이 호출된다.

해당 오류를 처리할 컨트롤러가 필요하다.

```
package hello.exception.servlet;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Slf4j
@Controller
public class ErrorPageController {

 @RequestMapping("/error-page/404")
 public String errorPage404(HttpServletRequest request, HttpServletResponse response) {
 log.info("errorPage 404");
 return "error-page/404";
 }

 @RequestMapping("/error-page/500")
 public String errorPage500(HttpServletRequest request, HttpServletResponse response) {
 log.info("errorPage 500");
 return "error-page/500";
 }
}
```

## 오류 처리 View

```
/templates/error-page/404.html
```

```
<!DOCTYPE HTML>

<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
 <div class="py-5 text-center">
 <h2>404 오류 화면</h2>
 </div>

 <div>
 <p>오류 화면 입니다.</p>
 </div>

 <hr class="my-4">
</div> <!-- /container -->

</body>
</html>
```

```
/templates/error-page/500.html
```

```
<!DOCTYPE HTML>

<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
 <div class="py-5 text-center">
 <h2>500 오류 화면</h2>
```

```

</div>

<div>
 <p>오류 화면 입니다.</p>
</div>

<hr class="my-4">

</div> <!-- /container -->

</body>
</html>

```

테스트 해보자.

- <http://localhost:8080/error-ex>
- <http://localhost:8080/error-404>
- <http://localhost:8080/error-500>

설정한 오류 페이지가 정상 노출되는 것을 확인할 수 있다.

## 서블릿 예외 처리 - 오류 페이지 작동 원리

서블릿은 `Exception`(예외)가 발생해서 서블릿 밖으로 전달되거나 또는 `response.sendError()` 가 호출되었을 때 설정된 오류 페이지를 찾는다.

### 예외 발생 흐름

WAS(여기까지 전파) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러(예외발생)

### sendError 흐름

WAS(sendError 호출 기록 확인) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러  
`(response.sendError())`

WAS는 해당 예외를 처리하는 오류 페이지 정보를 확인한다.

```
new ErrorPage(RuntimeException.class, "/error-page/500")
```

예를 들어서 `RuntimeException` 예외가 WAS까지 전달되면, WAS는 오류 페이지 정보를 확인한다.

확인해보니 `RuntimeException`의 오류 페이지로 `/error-page/500`이 지정되어 있다. WAS는 오류 페이지를 출력하기 위해 `/error-page/500`를 다시 요청한다.

### 오류 페이지 요청 흐름

```
WAS `/error-page/500` 다시 요청 -> 필터 -> 서블릿 -> 인터셉터 -> 컨트롤러(/error-page/500) -> View
```

### 예외 발생과 오류 페이지 요청 흐름

1. WAS(여기까지 전파) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러(예외발생)
2. WAS `/error-page/500` 다시 요청 -> 필터 -> 서블릿 -> 인터셉터 -> 컨트롤러(/error-page/500) -> View

중요한 점은 웹 브라우저(클라이언트)는 서버 내부에서 이런 일이 일어나는지 전혀 모른다는 점이다. 오직 서버 내부에서 오류 페이지를 찾기 위해 추가적인 호출을 한다.

정리하면 다음과 같다.

1. 예외가 발생해서 WAS까지 전파된다.
2. WAS는 오류 페이지 경로를 찾아서 내부에서 오류 페이지를 호출한다. 이때 오류 페이지 경로로 필터, 서블릿, 인터셉터, 컨트롤러가 모두 다시 호출된다.

필터와 인터셉터가 다시 호출되는 부분은 조금 뒤에 자세히 설명하겠다.

### 오류 정보 추가

WAS는 오류 페이지를 단순히 다시 요청만 하는 것이 아니라, 오류 정보를 `request`의 `attribute`에 추가해서 넘겨준다.

필요하면 오류 페이지에서 이렇게 전달된 오류 정보를 사용할 수 있다.

### ErrorPageController - 오류 출력

```
package hello.exception.servlet;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Slf4j
@Controller
public class ErrorPageController {

 //RequestDispatcher 상수로 정의되어 있음
 public static final String ERROR_EXCEPTION =
 "javax.servlet.error.exception";
 public static final String ERROR_EXCEPTION_TYPE =
 "javax.servlet.error.exception_type";
 public static final String ERROR_MESSAGE = "javax.servlet.error.message";
 public static final String ERROR_REQUEST_URI =
 "javax.servlet.error.request_uri";
 public static final String ERROR_SERVLET_NAME =
 "javax.servlet.error.servlet_name";
 public static final String ERROR_STATUS_CODE =
 "javax.servlet.error.status_code";

 @RequestMapping("/error-page/404")
 public String errorPage404(HttpServletRequest request, HttpServletResponse
 response) {
 log.info("errorPage 404");
 printErrorInfo(request);
 return "error-page/404";
 }

 @RequestMapping("/error-page/500")
 public String errorPage500(HttpServletRequest request, HttpServletResponse
 response) {
 log.info("errorPage 500");
 printErrorInfo(request);
 }
}
```

```

 return "error-page/500";
 }

 private void printErrorInfo(HttpServletRequest request) {
 log.info("ERROR_EXCEPTION: ex=",
request.getAttribute(ERROR_EXCEPTION));
 log.info("ERROR_EXCEPTION_TYPE: {}",

request.getAttribute(ERROR_EXCEPTION_TYPE));
 log.info("ERROR_MESSAGE: {}", request.getAttribute(ERROR_MESSAGE)); // ex의 경우 NestedServletException 스프링이 한번 감싸서 반환
 log.info("ERROR_REQUEST_URI: {}",

request.getAttribute(ERROR_REQUEST_URI));
 log.info("ERROR_SERVLET_NAME: {}",

request.getAttribute(ERROR_SERVLET_NAME));
 log.info("ERROR_STATUS_CODE: {}",

request.getAttribute(ERROR_STATUS_CODE));
 log.info("dispatchType={}", request.getDispatcherType());
 }

}

```

### request.attribute에 서버가 담아준 정보

- javax.servlet.error.exception : 예외
- javax.servlet.error.exception\_type : 예외 타입
- javax.servlet.error.message : 오류 메시지
- javax.servlet.error.request\_uri : 클라이언트 요청 URI
- javax.servlet.error.servlet\_name : 오류가 발생한 서블릿 이름
- javax.servlet.error.status\_code : HTTP 상태 코드

### 서블릿 예외 처리 - 필터

#### 목표

예외 처리에 따른 필터와 인터셉터 그리고 서블릿이 제공하는 `DispatcherType` 이해하기

## 예외 발생과 오류 페이지 요청 흐름

1. WAS(여기까지 전파) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러(예외발생)
2. WAS `/error-page/500` 다시 요청 -> 필터 -> 서블릿 -> 인터셉터 -> 컨트롤러(`/error-page/500`) -> View

오류가 발생하면 오류 페이지를 출력하기 위해 WAS 내부에서 다시 한번 호출이 발생한다. 이때 필터, 서블릿, 인터셉터도 모두 다시 호출된다. 그런데 로그인 인증 체크 같은 경우를 생각해보면, 이미 한번 필터나, 인터셉터에서 로그인 체크를 완료했다. 따라서 서버 내부에서 오류 페이지를 호출한다고 해서 해당 필터나 인터셉터가 한번 더 호출되는 것은 매우 비효율적이다.  
결국 클라이언트로 부터 발생한 정상 요청인지, 아니면 오류 페이지를 출력하기 위한 내부 요청인지 구분할 수 있어야 한다. 서블릿은 이런 문제를 해결하기 위해 `DispatcherType`이라는 추가 정보를 제공한다.

### DispatcherType

필터는 이런 경우를 위해서 `dispatcherTypes`라는 옵션을 제공한다.

이전 강의의 마지막에 다음 로그를 추가했다.

```
log.info("dispatchType={}", request.getDispatcherType())
```

그리고 출력해보면 오류 페이지에서 `dispatchType=ERROR`로 나오는 것을 확인할 수 있다.

고객이 처음 요청하면 `dispatcherType=REQUEST`이다.

이렇듯 서블릿 스펙은 실제 고객이 요청한 것인지, 서버가 내부에서 오류 페이지를 요청하는 것인지

`DispatcherType`으로 구분할 수 있는 방법을 제공한다.

#### javax.servlet.DispatcherType

```
public enum DispatcherType {
 FORWARD,
 INCLUDE,
 REQUEST,
 ASYNC,
 ERROR
}
```

### DispatcherType

- `REQUEST` : 클라이언트 요청
- `ERROR` : 오류 요청

- FORWARD : MVC에서 배웠던 서블릿에서 다른 서블릿이나 JSP를 호출할 때  
RequestDispatcher.forward(request, response);
- INCLUDE : 서블릿에서 다른 서블릿이나 JSP의 결과를 포함할 때  
RequestDispatcher.include(request, response);
- ASYNC : 서블릿 비동기 호출

## 필터와 DispatcherType

필터와 DispatcherType이 어떻게 사용되는지 알아보자.

### LogFilter - DispatcherType 로그 추가

```
package hello.exception.filter;

import lombok.extern.slf4j.Slf4j;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
import java.util.UUID;

@Slf4j
public class LogFilter implements Filter {

 @Override
 public void init(FilterConfig filterConfig) throws ServletException {
 log.info("log filter init");
 }

 @Override
 public void doFilter(ServletRequest request, ServletResponse response,
 FilterChain chain) throws IOException, ServletException {
 HttpServletRequest httpRequest = (HttpServletRequest) request;
 String requestURI = httpRequest.getRequestURI();

 String uuid = UUID.randomUUID().toString();
 }
}
```

```

 try {
 log.info("REQUEST [{}][{}][{}]", uuid,
request.getDispatcherType(), requestURI);
 chain.doFilter(request, response);
 } catch (Exception e) {
 throw e;
 } finally {
 log.info("RESPONSE [{}][{}][{}]", uuid,
request.getDispatcherType(), requestURI);
 }
 }

@Override
public void destroy() {
 log.info("log filter destroy");
}

```

로그를 출력하는 부분에 `request.getDispatcherType()` 을 추가해두었다.

## WebConfig

```

package hello.exception;

import hello.exception.filter.LogFilter;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import javax.servlet.DispatcherType;
import javax.servlet.Filter;

@Configuration
public class WebConfig implements WebMvcConfigurer {

 @Bean

```

```

public FilterRegistrationBean logFilter() {
 FilterRegistrationBean<Filter> filterRegistrationBean = new
 FilterRegistrationBean<>();
 filterRegistrationBean.setFilter(new LogFilter());
 filterRegistrationBean.setOrder(1);
 filterRegistrationBean.addUrlPatterns("/*");
 filterRegistrationBean.setDispatcherTypes(DispatcherType.REQUEST,
 DispatcherType.ERROR);
 return filterRegistrationBean;
}

}

```

```

filterRegistrationBean.setDispatcherTypes(DispatcherType.REQUEST,
DispatcherType.ERROR);

```

이렇게 두 가지를 모두 넣으면 클라이언트 요청은 물론이고, 오류 페이지 요청에서도 필터가 호출된다. 아무것도 넣지 않으면 기본 값이 `DispatcherType.REQUEST`이다. 즉 클라이언트의 요청이 있는 경우에만 필터가 적용된다. 특별히 오류 페이지 경로도 필터를 적용할 것이 아니면, 기본 값을 그대로 사용하면 된다.

물론 오류 페이지 요청 전용 필터를 적용하고 싶으면 `DispatcherType.ERROR`만 지정하면 된다.

## 서블릿 예외 처리 - 인터셉터

인터셉터 종복 호출 제거

### **LogInterceptor - DispatcherType 로그 추가**

```

package hello.exception.interceptor;

import lombok.extern.slf4j.Slf4j;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.UUID;

```

```
@Slf4j
public class LogInterceptor implements HandlerInterceptor {

 public static final String LOG_ID = "logId";

 @Override
 public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {

 String requestURI = request.getRequestURI();

 String uuid = UUID.randomUUID().toString();
 request.setAttribute(LOG_ID, uuid);

 log.info("REQUEST [{}][{}][{}][{}]", uuid,
request.getDispatcherType(), requestURI, handler);
 return true;
 }

 @Override
 public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
 log.info("postHandle [{}]", modelAndView);
 }

 @Override
 public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
 String requestURI = request.getRequestURI();
 String logId = (String)request.getAttribute(LOG_ID);
 log.info("RESPONSE [{}][{}][{}]", logId, request.getDispatcherType(),
requestURI);
 if (ex != null) {
 log.error("afterCompletion error!!", ex);
 }
 }
}
```

앞서 필터의 경우에는 필터를 등록할 때 어떤 `DispatcherType` 인 경우에 필터를 적용할지 선택할 수 있었다. 그런데 인터셉터는 서블릿이 제공하는 기능이 아니라 스프링이 제공하는 기능이다. 따라서 `DispatcherType` 과 무관하게 항상 호출된다.

대신에 인터셉터는 다음과 같이 요청 경로에 따라서 추가하거나 제외하기 쉽게 되어 있기 때문에, 이러한 설정을 사용해서 오류 페이지 경로를 `excludePathPatterns` 를 사용해서 빼주면 된다.

```
package hello.exception;

import hello.exception.filter.LogFilter;
import hello.exception.interceptor.LogInterceptor;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import javax.servlet.DispatcherType;
import javax.servlet.Filter;

@Configuration
public class WebConfig implements WebMvcConfigurer {

 @Override
 public void addInterceptors(InterceptorRegistry registry) {
 registry.addInterceptor(new LogInterceptor())
 .order(1)
 .addPathPatterns("/**")
 .excludePathPatterns(
 "/css/**", "/*.ico",
 "/error", "/error-page/**" //오류 페이지 경로
);
 }

 //Bean
 public FilterRegistrationBean logFilter() {
 FilterRegistrationBean<Filter> filterRegistrationBean = new
 FilterRegistrationBean<>();
 }
}
```

```

 filterRegistrationBean.setFilter(new LogFilter());
 filterRegistrationBean.setOrder(1);
 filterRegistrationBean.addUrlPatterns("/*");
 filterRegistrationBean.setDispatcherTypes(DispatcherType.REQUEST,
 DispatcherType.ERROR);
 return filterRegistrationBean;
 }

}

```

인터셉터와 중복으로 처리되지 않기 위해 앞의 `logFilter()` 의 `@Bean`에 주석을 달아두자.  
여기에서 `/error-page/**`를 제거하면 `error-page/500` 같은 내부 호출의 경우에도 인터셉터가 호출된다.

## 전체 흐름 정리

`/hello` 정상 요청

```
WAS(/hello, dispatchType=REQUEST) -> 필터 -> 서블릿 -> 인터셉터 -> 컨트롤러 -> View
```

`/error-ex` 오류 요청

- 필터는 `DispatchType`으로 중복 호출 제거 (`dispatchType=REQUEST`)
- 인터셉터는 경로 정보로 중복 호출 제거(`excludePathPatterns("/error-page/**")`)

1. WAS(/error-ex, dispatchType=REQUEST) -> 필터 -> 서블릿 -> 인터셉터 -> 컨트롤러
2. WAS(여기까지 전파) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러(예외발생)
3. WAS 오류 페이지 확인
4. WAS(/error-page/500, dispatchType=ERROR) -> 필터(x) -> 서블릿 -> 인터셉터(x) -> 컨트롤러(/error-page/500) -> View

지금까지 예외 처리 페이지를 만들기 위해서 다음과 같은 복잡한 과정을 거쳤다.

- `WebServerCustomizer` 를 만들고
- 예외 종류에 따라서 `ErrorPage` 를 추가하고
- 예외 처리용 컨트롤러 `ErrorPageController` 를 만듬

스프링 부트는 이런 과정을 모두 기본으로 제공한다.

- `ErrorPage` 를 자동으로 등록한다. 이때 `/error` 라는 경로로 기본 오류 페이지를 설정한다.
  - `new ErrorPage("/error")`, 상태코드와 예외를 설정하지 않으면 기본 오류 페이지로 사용된다.
  - 서블릿 밖으로 예외가 발생하거나, `response.sendError(...)` 가 호출되면 모든 오류는 `/error` 를 호출하게 된다.
- `BasicErrorHandler` 라는 스프링 컨트롤러를 자동으로 등록한다.
  - `ErrorPage` 에서 등록한 `/error` 를 매핑해서 처리하는 컨트롤러다.

### 참고

`ErrorMvcAutoConfiguration` 이라는 클래스가 오류 페이지를 자동으로 등록하는 역할을 한다.

### 주의

스프링 부트가 제공하는 기본 오류 메커니즘을 사용하도록 `WebServerCustomizer`에 있는 `@Component` 를 주석 처리하자.

이제 오류가 발생했을 때 오류 페이지로 `/error` 를 기본 요청한다. 스프링 부트가 자동 등록한 `BasicErrorHandler` 는 이 경로를 기본으로 받는다.

### 개발자는 오류 페이지만 등록

`BasicErrorHandler` 는 기본적인 로직이 모두 개발되어 있다.

개발자는 오류 페이지 화면만 `BasicErrorHandler` 가 제공하는 룰과 우선순위에 따라서 등록하면 된다. 정적 HTML이면 정적 리소스, 뷰 템플릿을 사용해서 동적으로 오류 화면을 만들고 싶으면 뷰 템플릿 경로에 오류 페이지 파일을 만들어서 넣어두기만 하면 된다.

### 뷰 선택 우선순위

`BasicErrorHandler` 의 처리 순서

1. 뷰 템플릿
  - `resources/templates/error/500.html`
  - `resources/templates/error/5xx.html`
2. 정적 리소스(`static`, `public`)
  - `resources/static/error/400.html`

- resources/static/error/404.html
- resources/static/error/4xx.html

### 3. 적용 대상이 없을 때 뷰 이름(error)

- resources/templates/error.html

해당 경로 위치에 HTTP 상태 코드 이름의 뷰 파일을 넣어두면 된다.

뷰 템플릿이 정적 리소스보다 우선순위가 높고, 404, 500처럼 구체적인 것이 5xx처럼 덜 구체적인 것 보다 우선순위가 높다.

5xx, 4xx 라고 하면 500대, 400대 오류를 처리해준다.

### 오류 뷰 템플릿 추가

resources/templates/error/4xx.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
 <div class="py-5 text-center">
 <h2>4xx 오류 화면 스프링 부트 제공</h2>
 </div>

 <div>
 <p>오류 화면 입니다.</p>
 </div>

 <hr class="my-4">

</div> <!-- /container -->

</body>
</html>
```

resources/templates/error/404.html

```
<!DOCTYPE HTML>

<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
 <div class="py-5 text-center">
 <h2>404 오류 화면 스프링 부트 제공</h2>
 </div>

 <div>
 <p>오류 화면 입니다.</p>
 </div>

 <hr class="my-4">
</div> <!-- /container -->

</body>
</html>
```

resources/templates/error/500.html

```
<!DOCTYPE HTML>

<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
 <div class="py-5 text-center">
 <h2>500 오류 화면 스프링 부트 제공</h2>
 </div>

 <div>
```

```

<p>오류 화면 입니다.</p>
</div>
<hr class="my-4">

</div> <!-- /container -->

</body>
</html>

```

## 등록한 오류 페이지

```

resources/templates/error/4xx.html
resources/templates/error/404.html
resources/templates/error/500.html

```

### 테스트

- <http://localhost:8080/error-404> → 404.html
- <http://localhost:8080/error-400> → 4xx.html (400 오류 페이지가 없지만 4xx가 있음)
- <http://localhost:8080/error-500> → 500.html
- <http://localhost:8080/error-ex> → 500.html (예외는 500으로 처리)

## 스프링 부트 - 오류 페이지2

### BasicErrorHandler가 제공하는 기본 정보들

BasicErrorHandler 컨트롤러는 다음 정보를 model에 담아서 뷰에 전달한다. 뷰 템플릿은 이 값을 활용해서 출력할 수 있다.

```

* timestamp: Fri Feb 05 00:00:00 KST 2021
* status: 400
* error: Bad Request
* exception: org.springframework.validation.BindException
* trace: 예외 trace
* message: Validation failed for object='data'. Error count: 1
* errors: Errors(BindingResult)
* path: 클라이언트 요청 경로 (`/hello`)

```

오류 정보 추가 - resources/templates/error/500.html

```
<!DOCTYPE HTML>

<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
 <div class="py-5 text-center">
 <h2>500 오류 화면 스프링 부트 제공</h2>
 </div>

 <div>
 <p>오류 화면 입니다.</p>
 </div>

 오류 정보

 <li th:text="${timestamp}">
 <li th:text="${path}">
 <li th:text="${status}">
 <li th:text="${message}">
 <li th:text="${error}">
 <li th:text="${exception}">
 <li th:text="${errors}">
 <li th:text="${trace}">

<hr class="my-4">

</div> <!-- /container -->
```

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

오류 관련 내부 정보들을 고객에게 노출하는 것은 좋지 않다. 고객이 해당 정보를 읽어도 혼란만 더해지고, 보안상 문제가 될 수도 있다.

그래서 `BasicErrorController` 오류 컨트롤러에서 다음 오류 정보를 `model`에 포함할지 여부 선택할 수 있다.

- `application.properties`
- `server.error.include-exception=false`: exception 포함 여부( `true`, `false` )
  - `server.error.include-message=never`: message 포함 여부
  - `server.error.include-stacktrace=never`: trace 포함 여부
  - `server.error.include-binding-errors=never`: errors 포함 여부

`application.properties`

```
server.error.include-exception=true
server.error.include-message=on_param
server.error.include-stacktrace=on_param
server.error.include-binding-errors=on_param
```

기본 값이 `naver`인 부분은 다음 3가지 옵션을 사용할 수 있다.

`never, always, on_param`

- `never`: 사용하지 않음
- `always`: 항상 사용
- `on_param`: 파라미터가 있을 때 사용

`on_param`은 파라미터가 있으면 해당 정보를 노출한다. 디버그 시 문제를 확인하기 위해 사용할 수 있다.

그런데 이 부분도 개발 서버에서 사용할 수 있지만, 운영 서버에서는 권장하지 않는다.

`on_param`으로 설정하고 다음과 같이 HTTP 요청시 파라미터를 전달하면 해당 정보들이 `model`에 담겨서 뷰 템플릿에서 출력된다.

`message=&errors=&trace=`

## 테스트

<http://localhost:8080/error-ex?message=&errors=&trace=>

**실무에서는 이것들을 노출하면 안된다!** 사용자에게는 이쁜 오류 화면과 고객이 이해할 수 있는 간단한 오류 메시지를 보여주고 오류는 서버에 로그로 남겨서 로그로 확인해야 한다.

### 스프링 부트 오류 관련 옵션

- `server.error.whitelabel.enabled=true` : 오류 처리 화면을 못 찾을 시, 스프링 whitelabel 오류 페이지 적용
- `server.error.path=/error` : 오류 페이지 경로, 스프링이 자동 등록하는 서블릿 글로벌 오류 페이지 경로와 `BasicErrorController` 오류 컨트롤러 경로에 함께 사용된다.

### 확장 포인트

에러 공통 처리 컨트롤러의 기능을 변경하고 싶으면 `ErrorController` 인터페이스를 상속 받아서 구현하거나 `BasicErrorController` 상속 받아서 기능을 추가하면 된다.

### 정리

스프링 부트가 기본으로 제공하는 오류 페이지를 활용하면 오류 페이지와 관련된 대부분의 문제는 손쉽게 해결할 수 있다.

### 정리

# 9. API 예외 처리

#인강/5. 스프링 MVC 2/강의#

## 목차

- 9. API 예외 처리 - API 예외 처리 - 시작
- 9. API 예외 처리 - API 예외 처리 - 스프링 부트 기본 오류 처리
- 9. API 예외 처리 - API 예외 처리 - HandlerExceptionResolver 시작
- 9. API 예외 처리 - API 예외 처리 - HandlerExceptionResolver 활용
- 9. API 예외 처리 - API 예외 처리 - 스프링이 제공하는 ExceptionResolver1
- 9. API 예외 처리 - API 예외 처리 - 스프링이 제공하는 ExceptionResolver2
- 9. API 예외 처리 - API 예외 처리 - @ExceptionHandler
- 9. API 예외 처리 - API 예외 처리 - @ControllerAdvice
- 9. API 예외 처리 - 정리

## API 예외 처리 - 시작

### 목표

API 예외 처리는 어떻게 해야 할까?

HTML 페이지의 경우 지금까지 설명했던 것처럼 4xx, 5xx와 같은 오류 페이지지만 있으면 대부분의 문제를 해결할 수 있다.

그런데 API의 경우에는 생각할 내용이 더 많다. 오류 페이지는 단순히 고객에게 오류 화면을 보여주고 끝이지만, API는 각 오류 상황에 맞는 오류 응답 스펙을 정하고, JSON으로 데이터를 내려주어야 한다.

지금부터 API의 경우 어떻게 예외 처리를 하면 좋은지 알아보자.

API도 오류 페이지에서 설명했던 것처럼 처음으로 돌아가서 서블릿 오류 페이지로 방식을 사용해보자.

### WebServerCustomizer 다시 동작

```
package hello.exception;

import org.springframework.boot.web.server.ConfigurableWebServerFactory;
import org.springframework.boot.web.server.ErrorPage;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
```

```

@Component
public class WebServerCustomizer implements
WebServerFactoryCustomizer<ConfigurableWebServerFactory> {
 @Override
 public void customize(ConfigurableWebServerFactory factory) {

 ErrorCode errorCode404 = new ErrorCode(HttpStatus.NOT_FOUND, "/error-
page/404");
 ErrorCode errorCode500 = new
 ErrorCode(HttpStatus.INTERNAL_SERVER_ERROR, "/error-page/500");
 ErrorCode errorCodeEx = new ErrorCode(RuntimeException.class, "/error-
page/500");
 factory.addErrorPages(errorCode404, errorCode500, errorCodeEx);
 }
}

```

WebServerCustomizer 가 다시 사용되도록 하기 위해 @Component 애노테이션에 있는 주석을 풀자  
 이제 WAS에 예외가 전달되거나, response.sendError() 가 호출되면 위에 등록한 예외 페이지 경로가  
 호출된다.

## ApiExceptionController - API 예외 컨트롤러

```

package hello.exception.api;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@Slf4j
@RestController
public class ApiExceptionController {

 @GetMapping("/api/members/{id}")
 public MemberDto getMember(@PathVariable("id") String id) {
}

```

```

 if (id.equals("ex")) {
 throw new RuntimeException("잘못된 사용자");
 }

 return new MemberDto(id, "hello " + id);
}

@Data
@AllArgsConstructor
static class MemberDto {
 private String memberId;
 private String name;
}

```

단순히 회원을 조회하는 기능을 하나 만들었다. 예외 테스트를 위해 URL에 전달된 `id`의 값이 `ex` 이면 예외가 발생하도록 코드를 심어두었다.

### Postman으로 테스트

HTTP Header에 `Accept` 가 `application/json` 인 것을 꼭 확인하자.

#### 정상 호출

`http://localhost:8080/api/members/spring`

```
{
 "memberId": "spring",
 "name": "hello spring"
}
```

#### 예외 발생 호출

`http://localhost:8080/api/members/ex`

```
<!DOCTYPE HTML>
<html>
<head>
```

```
</head>
<body>
...
</body>
```

API를 요청했는데, 정상의 경우 API로 JSON 형식으로 데이터가 정상 반환된다. 그런데 오류가 발생하면 우리가 미리 만들어둔 오류 페이지 HTML이 반환된다. 이것은 기대하는 바가 아니다. 클라이언트는 정상 요청이든, 오류 요청이든 JSON이 반환되기를 기대한다. 웹 브라우저가 아닌 이상 HTML을 직접 받아서 할 수 있는 것은 별로 없다.

문제를 해결하려면 오류 페이지 컨트롤러도 JSON 응답을 할 수 있도록 수정해야 한다.

### ErrorPageController - API 응답 추가

```
@RequestMapping(value = "/error-page/500", produces =
MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Map<String, Object>> errorPage500Api(HttpServletRequest
request, HttpServletResponse response) {
 log.info("API errorPage 500");

 Map<String, Object> result = new HashMap<>();
 Exception ex = (Exception) request.getAttribute(ERROR_EXCEPTION);
 result.put("status", request.getAttribute(ERROR_STATUS_CODE));
 result.put("message", ex.getMessage());

 Integer statusCode = (Integer)
 request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE);
 return new ResponseEntity(result, HttpStatus.valueOf(statusCode));
}
```

`produces = MediaType.APPLICATION_JSON_VALUE`의 뜻은 클라이언트가 요청하는 HTTP Header의 `Accept`의 값이 `application/json` 일 때 해당 메서드가 호출된다는 것이다. 결국 클라이언트가 받고 싶은 미디어타입이 json이면 이 컨트롤러의 메서드가 호출된다.

응답 데이터를 위해서 `Map`을 만들고 `status`, `message` 키에 값을 할당했다. Jackson 라이브러리는 `Map`을 JSON 구조로 변환할 수 있다.

`ResponseEntity` 를 사용해서 응답하기 때문에 메시지 컨버터가 동작하면서 클라이언트에 JSON이 반환된다.

포스트맨을 통해서 다시 테스트 해보자.

HTTP Header에 `Accept` 가 `application/json` 인 것을 꼭 확인하자.

`http://localhost:8080/api/members/ex`

```
{
 "message": "잘못된 사용자",
 "status": 500
}
```

HTTP Header에 `Accept` 가 `application/json` 이 아니면, 기존 오류 응답인 HTML 응답이 출력되는 것을 확인할 수 있다.

## API 예외 처리 - 스프링 부트 기본 오류 처리

API 예외 처리도 스프링 부트가 제공하는 기본 오류 방식을 사용할 수 있다.

스프링 부트가 제공하는 `BasicErrorController` 코드를 보자.

### BasicErrorController 코드

```
@RequestMapping(produces = MediaType.TEXT_HTML_VALUE)
public ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse
response) {}

 @RequestMapping
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {}
```

`/error` 동일한 경로를 처리하는 `errorHtml()`, `error()` 두 메서드를 확인할 수 있다.

- `errorHtml()` : `produces = MediaType.TEXT_HTML_VALUE` : 클라이언트 요청의 Accept 헤더 값이 `text/html` 인 경우에는 `errorHtml()` 을 호출해서 view를 제공한다.

- `error()` : 그외 경우에 호출되고  `ResponseEntity`로 HTTP Body에 JSON 데이터를 반환한다.

### 스프링 부트의 예외 처리

앞서 학습했듯이 스프링 부트의 기본 설정은 오류 발생시 `/error` 를 오류 페이지로 요청한다.

`BasicErrorController` 는 이 경로를 기본으로 받는다. (`server.error.path` 로 수정 가능, 기본 경로 `/error`)

### Postman으로 실행

GET <http://localhost:8080/api/members/ex>

### 주의

`BasicErrorController` 를 사용하도록 `WebServerCustomizer` 의 `@Component` 를 주석처리 하자.

```
{
 "timestamp": "2021-04-28T00:00:00.000+00:00",
 "status": 500,
 "error": "Internal Server Error",
 "exception": "java.lang.RuntimeException",
 "trace": "java.lang.RuntimeException: 잘못된 사용자\\n\\tat
hello.exception.web.api.ApiExceptionController.getMember(ApiExceptionController
.java:19...",
 "message": "잘못된 사용자",
 "path": "/api/members/ex"
}
```

스프링 부트는 `BasicErrorController` 가 제공하는 기본 정보들을 활용해서 오류 API를 생성해준다.

다음 옵션들을 설정하면 더 자세한 오류 정보를 추가할 수 있다.

- `server.error.include-binding-errors=always`
- `server.error.include-exception=true`
- `server.error.include-message=always`
- `server.error.include-stacktrace=always`

물론 오류 메시지는 이렇게 막 추가하면 보안상 위험할 수 있다. 간결한 메시지만 노출하고, 로그를 통해서 확인하자.

## Html 페이지 vs API 오류

BasicErrorHandler 를 확장하면 JSON 메시지도 변경할 수 있다. 그런데 API 오류는 조금 뒤에 설명할 @ExceptionHandler 가 제공하는 기능을 사용하는 것이 더 나은 방법이므로 지금은 BasicErrorHandler 를 확장해서 JSON 오류 메시지를 변경할 수 있다 정도로만 이해해두자.

스프링 부트가 제공하는 BasicErrorHandler 는 HTML 페이지를 제공하는 경우에는 매우 편리하다. 4xx, 5xx 등등 모두 잘 처리해준다. 그런데 API 오류 처리는 다른 차원의 이야기이다. API마다, 각각의 컨트롤러나 예외마다 서로 다른 응답 결과를 출력해야 할 수도 있다. 예를 들어서 회원과 관련된 API에서 예외가 발생할 때 응답과, 상품과 관련된 API에서 발생하는 예외에 따라 그 결과가 달라질 수 있다. 결과적으로 매우 세밀하고 복잡하다. 따라서 이 방법은 HTML 화면을 처리할 때 사용하고, API는 오류 처리는 뒤에서 설명할 @ExceptionHandler 를 사용하자.

그렇다면 복잡한 API는 오류는 어떻게 처리해야하는지 지금부터 하나씩 알아보자.

## API 예외 처리 - HandlerExceptionResolver 시작

### 목표

예외가 발생해서 서블릿을 넘어 WAS까지 예외가 전달되면 HTTP 상태코드가 500으로 처리된다. 발생하는 예외에 따라서 400, 404 등등 다른 상태코드도 처리하고 싶다. 오류 메시지, 형식등을 API마다 다르게 처리하고 싶다.

### 상태코드 변환

예를 들어서 IllegalArgumentException 을 처리하지 못해서 컨트롤러 밖으로 넘어가는 일이 발생하면 HTTP 상태코드를 400으로 처리하고 싶다. 어떻게 해야할까?

### ApiExceptionController - 수정

```
@GetMapping("/api/members/{id}")
public MemberDto getMember(@PathVariable("id") String id) {

 if (id.equals("ex")) {
 throw new RuntimeException("잘못된 사용자");
 }
}
```

```
if (id.equals("bad")) {
 throw new IllegalArgumentException("잘못된 입력 값");
}

return new MemberDto(id, "hello " + id);
}
```

<http://localhost:8080/api/members/bad> 라고 호출하면 `IllegalArgumentException`이 발생하도록 했다.

실행해보면 상태 코드가 **500**인 것을 확인할 수 있다.

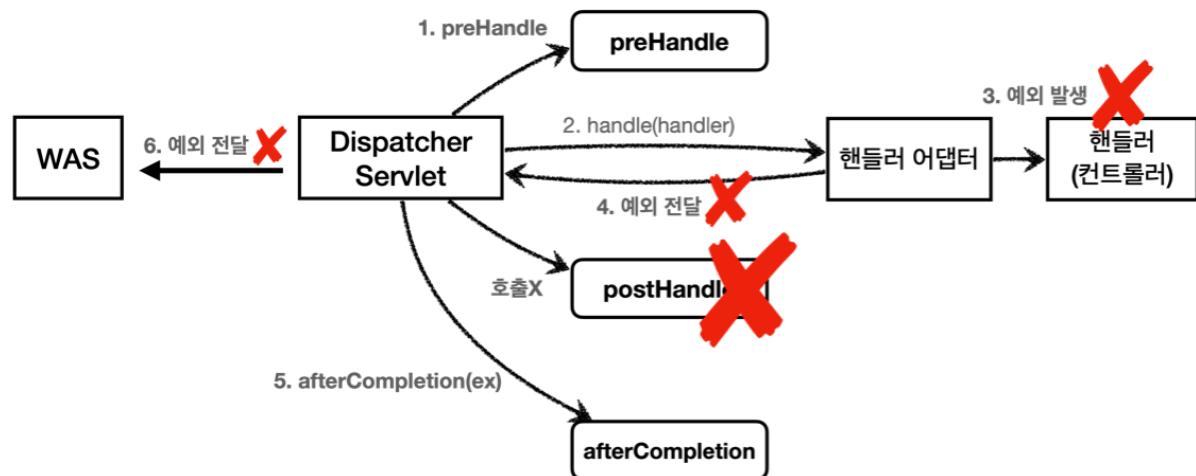
```
{
 "status": 500,
 "error": "Internal Server Error",
 "exception": "java.lang.IllegalArgumentException",
 "path": "/api/members/bad"
}
```

## HandlerExceptionResolver

스프링 MVC는 컨트롤러(핸들러) 밖으로 예외가 던져진 경우 예외를 해결하고, 동작을 새로 정의할 수 있는 방법을 제공한다. 컨트롤러 밖으로 던져진 예외를 해결하고, 동작 방식을 변경하고 싶으면 `HandlerExceptionResolver`를 사용하면 된다. 줄여서 `ExceptionResolver`라 한다.

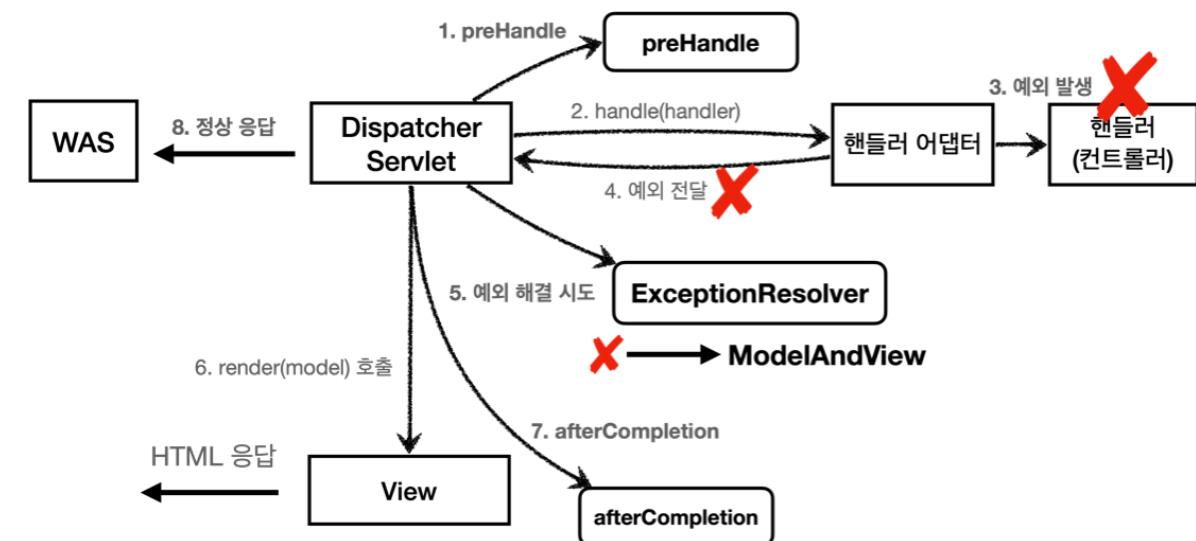
### ExceptionResolver 적용 전

### 예외 처리 - ExceptionResolver 적용 전



### ExceptionResolver 적용 후

#### 예외 처리 - ExceptionResolver 적용 후



참고: ExceptionResolver로 예외를 해결해도 postHandle()은 호출되지 않는다.

### HandlerExceptionResolver - 인터페이스

```
public interface HandlerExceptionResolver {
 ModelAndView resolveException(
 HttpServletRequest request, HttpServletResponse response,
 Object handler, Exception ex);
}
```

- handler : 핸들러(컨트롤러) 정보
- Exception ex : 핸들러(컨트롤러)에서 발생한 발생한 예외

## MyHandlerExceptionResolver

```

package hello.exception.resolver;

import lombok.extern.slf4j.Slf4j;
import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Slf4j
public class MyHandlerExceptionResolver implements HandlerExceptionResolver {

 @Override
 public ModelAndView resolveException(HttpServletRequest request,
 HttpServletResponse response, Object handler, Exception ex) {

 try {
 if (ex instanceof IllegalArgumentException) {
 log.info("IllegalArgumentException resolver to 400");
 response.sendError(HttpStatus.SC_BAD_REQUEST,
 ex.getMessage());
 return new ModelAndView();
 }
 } catch (IOException e) {
 log.error("resolver ex", e);
 }

 return null;
 }
}

```

- `ExceptionResolver` 가 `ModelAndView` 를 반환하는 이유는 마치 `try, catch` 를 하듯이, `Exception` 을 처리해서 정상 흐름처럼 변경하는 것이 목적이다. 이를 그대로 `Exception` 을 Resolver(해결)하는 것이 목적이다.

여기서는 `IllegalArgumentException` 이 발생하면 `response.sendError(400)` 를 호출해서 HTTP 상태 코드를 400으로 지정하고, 빈 `ModelAndView` 를 반환한다.

### 반환 값에 따른 동작 방식

`HandlerExceptionResolver` 의 반환 값에 따른 `DispatcherServlet` 의 동작 방식은 다음과 같다.

- **빈 ModelAndView:** `new ModelAndView()` 처럼 빈 `ModelAndView` 를 반환하면 뷰를 렌더링 하지 않고, 정상 흐름으로 서블릿이 리턴된다.
- **ModelAndView 지정:** `ModelAndView` 에 `View`, `Model` 등의 정보를 지정해서 반환하면 뷰를 렌더링 한다.
- **null:** `null` 을 반환하면, 다음 `ExceptionResolver` 를 찾아서 실행한다. 만약 처리할 수 있는 `ExceptionResolver` 가 없으면 예외 처리가 안되고, 기존에 발생한 예외를 서블릿 밖으로 던진다.

### ExceptionResolver 활용

- 예외 상태 코드 변환
  - 예외를 `response.sendError(xxx)` 호출로 변경해서 서블릿에서 상태 코드에 따른 오류를 처리하도록 위임
  - 이후 WAS는 서블릿 오류 페이지를 찾아서 내부 호출, 예를 들어서 스프링 부트가 기본으로 설정한 `/error` 가 호출됨
- 뷰 템플릿 처리
  - `ModelAndView` 에 값을 채워서 예외에 따른 새로운 오류화면 뷰 렌더링 해서 고객에게 제공
- API 응답 처리
  - `response.getWriter().println("hello")`; 처럼 HTTP 응답 바디에 직접 데이터를 넣어주는 것도 가능하다. 여기에 JSON 으로 응답하면 API 응답 처리를 할 수 있다.

### WebConfig - 수정

`WebMvcConfigurer` 를 통해 등록

```
/**
 * 기본 설정을 유지하면서 추가
 */
@Override
public void extendHandlerExceptionResolvers(List<HandlerExceptionResolver>
resolvers) {
```

```
resolvers.add(new MyHandlerExceptionResolver());
```

```
}
```

`configureHandlerExceptionResolvers(...)` 를 사용하면 스프링이 기본으로 등록하는 `ExceptionResolver` 가 제거되므로 주의, `extendHandlerExceptionResolvers` 를 사용하자.

### Postman으로 실행

- `http://localhost:8080/api/members/ex` → HTTP 상태 코드 500
- `http://localhost:8080/api/members/bad` → HTTP 상태 코드 400

## API 예외 처리 - HandlerExceptionResolver 활용

### 예외를 여기서 마무리하기

예외가 발생하면 WAS까지 예외가 던져지고, WAS에서 오류 페이지 정보를 찾아서 다시 `/error` 를 호출하는 과정은 생각해보면 너무 복잡하다. `ExceptionResolver` 를 활용하면 예외가 발생했을 때 이런 복잡한 과정 없이 여기에서 문제를 깔끔하게 해결할 수 있다.

예제로 알아보자.

먼저 사용자 정의 예외를 하나 추가하자.

### UserException

```
package hello.exception.exception;

public class UserException extends RuntimeException {

 public UserException() {
 super();
 }

 public UserException(String message) {
 super(message);
 }

 public UserException(String message, Throwable cause) {
 super(message, cause);
 }
}
```

```

 }

 public UserException(Throwable cause) {
 super(cause);
 }

 protected UserException(String message, Throwable cause, boolean
enableSuppression, boolean writableStackTrace) {
 super(message, cause, enableSuppression, writableStackTrace);
 }
}

```

## ApiExceptionController - 예외 추가

```

package hello.exception.api;

import hello.exception.exception.UserException;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@Slf4j
@RestController
public class ApiExceptionController {

 @GetMapping("/api/members/{id}")
 public MemberDto getMember(@PathVariable("id") String id) {

 if (id.equals("ex")) {
 throw new RuntimeException("잘못된 사용자");
 }
 if (id.equals("bad")) {
 throw new IllegalArgumentException("잘못된 입력 값");
 }
 }
}

```

```

 if (id.equals("user-ex")) {
 throw new UserException("사용자 오류");
 }

 return new MemberDto(id, "hello " + id);
}

@Data
@AllArgsConstructor
static class MemberDto {
 private String memberId;
 private String name;
}
}

```

<http://localhost:8080/api/members/user-ex> 호출시 `UserException`이 발생하도록 해두었다.

이제 이 예외를 처리하는 `UserHandlerExceptionResolver`를 만들어보자.

### UserHandlerExceptionResolver

```

package hello.exception.resolver;

import com.fasterxml.jackson.databind.ObjectMapper;
import hello.exception.exception.UserException;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

@Slf4j
public class UserHandlerExceptionResolver implements HandlerExceptionResolver {

```

```
private final ObjectMapper objectMapper = new ObjectMapper();

@Override
public ModelAndView resolveException(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) {

 try {
 if (ex instanceof UserException) {
 log.info("UserException resolver to 400");
 String acceptHeader = request.getHeader("accept");
 response.setStatus(HttpServletRequest.SC_BAD_REQUEST);

 if ("application/json".equals(acceptHeader)) {
 Map<String, Object> errorResult = new HashMap<>();
 errorResult.put("ex", ex.getClass());
 errorResult.put("message", ex.getMessage());
 String result =
objectMapper.writeValueAsString(errorResult);

 response.setContentType("application/json");
 response.setCharacterEncoding("utf-8");
 response.getWriter().write(result);
 return new ModelAndView();
 } else {
 //TEXT/HTML
 return new ModelAndView("error/500");
 }
 }
 } catch (IOException e) {
 log.error("resolver ex", e);
 }

 return null;
}
}
```

HTTP 요청 헤더의 `ACCEPT` 값이 `application/json` 이면 JSON으로 오류를 내려주고, 그 외 경우에는 `error/500`에 있는 HTML 오류 페이지를 보여준다.

## WebConfig에 UserHandlerExceptionResolver 추가

```
@Override
public void extendHandlerExceptionResolvers(List<HandlerExceptionResolver>
resolvers) {
 resolvers.add(new MyHandlerExceptionResolver());
 resolvers.add(new UserHandlerExceptionResolver());
}
```

### 실행

POSTMAN 실행,

```
http://localhost:8080/api/members/user-ex
```

ACCEPT : `application/json`

```
{
 "ex": "hello.exception.exception.UserException",
 "message": "사용자 오류"
}
```

ACCEPT : `text/html`

```
<!DOCTYPE HTML>
<html>
...
</html>
```

### 정리

`ExceptionResolver` 를 사용하면 컨트롤러에서 예외가 발생해도 `ExceptionResolver` 에서 예외를 처리해버린다.

따라서 예외가 발생해도 서블릿 컨테이너까지 예외가 전달되지 않고, 스프링 MVC에서 예외 처리는 끝이

난다.

결과적으로 WAS 입장에서는 정상 처리가 된 것이다. 이렇게 예외를 이곳에서 모두 처리할 수 있다는 것이 핵심이다.

서블릿 컨테이너까지 예외가 올라가면 복잡하고 지저분하게 추가 프로세스가 실행된다. 반면에 `ExceptionResolver`를 사용하면 예외처리가 상당히 깔끔해진다.

그런데 직접 `ExceptionResolver`를 구현하려고 하니 상당히 복잡하다. 지금부터 스프링이 제공하는 `ExceptionResolver`들을 알아보자.

## API 예외 처리 - 스프링이 제공하는 `ExceptionResolver`들

스프링 부트가 기본으로 제공하는 `ExceptionResolver`는 다음과 같다.

`HandlerExceptionResolverComposite`에 다음 순서로 등록

1. `ExceptionHandlerExceptionResolver`
2. `ResponseStatusExceptionResolver`
3. `DefaultHandlerExceptionResolver` → 우선 순위가 가장 낮다.

### **ExceptionHandlerExceptionResolver**

`@ExceptionHandler`을 처리한다. API 예외 처리는 대부분 이 기능으로 해결한다. 조금 뒤에 자세히 설명한다.

### **ResponseStatusExceptionResolver**

HTTP 상태 코드를 지정해준다.

예) `@ResponseStatus(value = HttpStatus.NOT_FOUND)`

### **DefaultHandlerExceptionResolver**

스프링 내부 기본 예외를 처리한다.

먼저 가장 쉬운 `ResponseStatusExceptionResolver`부터 알아보자.

### **ResponseStatusExceptionResolver**

`ResponseStatusExceptionResolver`는 예외에 따라서 HTTP 상태 코드를 지정해주는 역할을 한다.

다음 두 가지 경우를 처리한다.

- `@ResponseStatus` 가 달려있는 예외
- `ResponseStatusException` 예외

하나씩 확인해보자.

예외에 다음과 같이 `@ResponseStatus` 애노테이션을 적용하면 HTTP 상태 코드를 변경해준다.

```
package hello.exception.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(code = HttpStatus.BAD_REQUEST, reason = "잘못된 요청 오류")
public class BadRequestException extends RuntimeException {
}
```

`BadRequestException` 예외가 컨트롤러 밖으로 넘어가면 `ResponseStatusExceptionResolver` 예외가 해당 애노테이션을 확인해서 오류 코드를 `HttpStatus.BAD_REQUEST` (400)으로 변경하고, 메시지도 담는다.

`ResponseStatusExceptionResolver` 코드를 확인해보면 결국 `response.sendError(statusCode, resolvedReason)` 를 호출하는 것을 확인할 수 있다.

`sendError(400)` 를 호출했기 때문에 WAS에서 다시 오류 페이지(`/error`)를 내부 요청한다.

## ApiExceptionController - 추가

```
@GetMapping("/api/response-status-ex1")
public String responseStatusEx1() {
 throw new BadRequestException();
}
```

### 실행

`http://localhost:8080/api/response-status-ex1?message=`

```
{
 "status": 400,
```

```
 "error": "Bad Request",
 "exception": "hello.exception.exception.BadRequestException",
 "message": "잘못된 요청 오류",
 "path": "/api/response-status-ex1"
 }
```

## 메시지 기능

reason 을 MessageSource 에서 찾는 기능도 제공한다. reason = "error.bad"

```
package hello.exception.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

//@ResponseStatus(code = HttpStatus.BAD_REQUEST, reason = "잘못된 요청 오류")
@ResponseStatus(code = HttpStatus.BAD_REQUEST, reason = "error.bad")
public class BadRequestException extends RuntimeException {
}
```

## messages.properties

error.bad=잘못된 요청 오류입니다. 메시지 사용

## 메시지 사용 결과

```
{
 "status": 400,
 "error": "Bad Request",
 "exception": "hello.exception.exception.BadRequestException",
 "message": "잘못된 요청 오류입니다. 메시지 사용",
 "path": "/api/response-status-ex1"
}
```

## ResponseStatusException

`@ResponseStatus` 는 개발자가 직접 변경할 수 없는 예외에는 적용할 수 없다. (애노테이션을 직접 넣어야 하는데, 내가 코드를 수정할 수 없는 라이브러리의 예외 코드 같은 곳에는 적용할 수 없다.)  
추가로 애노테이션을 사용하기 때문에 조건에 따라 동적으로 변경하는 것도 어렵다. 이때는 `ResponseStatusException` 예외를 사용하면 된다.

## ApiExceptionController - 추가

```
@GetMapping("/api/response-status-ex2")
public String responseStatusEx2() {
 throw new ResponseStatusException(HttpStatus.NOT_FOUND, "error.bad", new
IllegalArgumentException());
}
```

<http://localhost:8080/api/response-status-ex2>

```
{
 "status": 404,
 "error": "Not Found",
 "exception": "org.springframework.web.server.ResponseStatusException",
 "message": "잘못된 요청 오류입니다. 메시지 사용",
 "path": "/api/response-status-ex2"
}
```

## API 예외 처리 - 스프링이 제공하는 ExceptionResolver2

이번에는 `DefaultHandlerExceptionResolver` 를 살펴보자.

`DefaultHandlerExceptionResolver` 는 스프링 내부에서 발생하는 스프링 예외를 해결한다.  
대표적으로 파라미터 바인딩 시점에 타입이 맞지 않으면 내부에서 `TypeMismatchException`이 발생하는데, 이 경우 예외가 발생했기 때문에 그냥 두면 서블릿 컨테이너까지 오류가 올라가고, 결과적으로 500 오류가 발생한다.

그런데 파라미터 바인딩은 대부분 클라이언트가 HTTP 요청 정보를 잘못 호출해서 발생하는 문제이다.

HTTP에서는 이런 경우 HTTP 상태 코드 400을 사용하도록 되어 있다.

`DefaultHandlerExceptionResolver` 는 이것을 500 오류가 아니라 HTTP 상태 코드 400 오류로

변경한다.

스프링 내부 오류를 어떻게 처리할지 수 많은 내용이 정의되어 있다.

### 코드 확인

`DefaultHandlerExceptionResolver.handleTypeMismatch` 를 보면 다음과 같은 코드를 확인할 수 있다.

`response.sendError(HttpServletRequest.SC_BAD_REQUEST) (400)`

결국 `response.sendError()` 를 통해서 문제를 해결한다.

`sendError(400)` 를 호출했기 때문에 WAS에서 다시 오류 페이지(`/error`)를 내부 요청한다.

### ApiExceptionController - 추가

```
@GetMapping("/api/default-handler-ex")
public String defaultException(@RequestParam Integer data) {
 return "ok";
}
```

`Integer data`에 문자를 입력하면 내부에서 `TypeMismatchException`이 발생한다.

### 실행

`http://localhost:8080/api/default-handler-ex?data=hello&message=`

```
{
 "status": 400,
 "error": "Bad Request",
 "exception":
"org.springframework.web.method.annotation.MethodArgumentTypeMismatchException"
,
 "message": "Failed to convert value of type 'java.lang.String' to required
type 'java.lang.Integer'; nested exception is java.lang.NumberFormatException:
For input string: \"hello\"",
 "path": "/api/default-handler-ex"
}
```

실행 결과를 보면 HTTP 상태 코드가 400인 것을 확인할 수 있다.

## 정리

지금까지 다음 `ExceptionResolver` 들에 대해 알아보았다.

1. `ExceptionHandlerExceptionResolver` → 다음 시간에
2. `ResponseStatusExceptionResolver` → HTTP 응답 코드 변경
3. `DefaultHandlerExceptionResolver` → 스프링 내부 예외 처리

지금까지 HTTP 상태 코드를 변경하고, 스프링 내부 예외의 상태코드를 변경하는 기능도 알아보았다.

그런데 `HandlerExceptionResolver` 를 직접 사용하기는 복잡하다. API 오류 응답의 경우 `response` 에 직접 데이터를 넣어야 해서 매우 불편하고 번거롭다. `ModelAndView` 를 반환해야 하는 것도 API에는 잘 맞지 않는다.

스프링은 이 문제를 해결하기 위해 `@ExceptionHandler` 라는 매우 혁신적인 예외 처리 기능을 제공한다.

그것이 아직 소개하지 않은 `ExceptionHandlerExceptionResolver` 이다.

## API 예외 처리 - `@ExceptionHandler`

### HTML 화면 오류 vs API 오류

웹 브라우저에 HTML 화면을 제공할 때는 오류가 발생하면 `BasicErrorController` 를 사용하는게 편하다.

이때는 단순히 5xx, 4xx 관련된 오류 화면을 보여주면 된다. `BasicErrorController` 는 이런 메커니즘을 모두 구현해두었다.

그런데 API는 각 시스템마다 응답의 모양도 다르고, 스펙도 모두 다르다. 예외 상황에 단순히 오류 화면을 보여주는 것이 아니라, 예외에 따라서 각각 다른 데이터를 출력해야 할 수도 있다. 그리고 같은 예외라고 해도 어떤 컨트롤러에서 발생했는가에 따라서 다른 예외 응답을 내려주어야 할 수 있다. 한마디로 매우 세밀한 제어가 필요하다.

앞서 이야기했지만, 예를 들어서 상품 API와 주문 API는 오류가 발생했을 때 응답의 모양이 완전히 다를 수 있다.

결국 지금까지 살펴본 `BasicErrorController` 를 사용하거나 `HandlerExceptionResolver` 를 직접 구현하는 방식으로 API 예외를 다루기는 쉽지 않다.

### API 예외처리의 어려운 점

- `HandlerExceptionResolver` 를 떠올려 보면 `ModelAndView` 를 반환해야 했다. 이것은 API 응답에는 필요하지 않다.
- API 응답을 위해서 `HttpServletResponse` 에 직접 응답 데이터를 넣어주었다. 이것은 매우 불편하다. 스프링 컨트롤러에 비유하면 마치 과거 서블릿을 사용하던 시절로 돌아간 것 같다.

- 특정 컨트롤러에서만 발생하는 예외를 별도로 처리하기 어렵다. 예를 들어서 회원을 처리하는 컨트롤러에서 발생하는 `RuntimeException` 예외와 상품을 관리하는 컨트롤러에서 발생하는 동일한 `RuntimeException` 예외를 서로 다른 방식으로 처리하고 싶다면 어떻게 해야 할까?

### **@ExceptionHandler**

스프링은 API 예외 처리 문제를 해결하기 위해 `@ExceptionHandler`라는 애노테이션을 사용하는 매우 편리한 예외 처리 기능을 제공하는데, 이것이 바로 `ExceptionHandlerExceptionResolver`이다.

스프링은 `ExceptionHandlerExceptionResolver`를 기본으로 제공하고, 기본으로 제공하는 `ExceptionResolver` 중에 우선순위도 가장 높다. 실무에서 API 예외 처리는 대부분 이 기능을 사용한다.

먼저 예제로 알아보자.

### **ErrorResult**

```
package hello.exception.exhandler;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class ErrorResult {
 private String code;
 private String message;
}
```

예외가 발생했을 때 API 응답으로 사용하는 객체를 정의했다.

### **ApiExceptionV2Controller**

```
package hello.exception.exhandler;

import hello.exception.exception.UserException;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
```

```
@Slf4j
@RestController
public class ApiExceptionV2Controller {

 @ResponseStatus(HttpStatus.BAD_REQUEST)
 @ExceptionHandler(IllegalArgumentException.class)
 public ErrorResult illegalExHandle(IllegalArgumentException e) {
 log.error("[exceptionHandle] ex", e);
 return new ErrorResult("BAD", e.getMessage());
 }

 @ExceptionHandler
 public ResponseEntity<ErrorResult> userExHandle(UserException e) {
 log.error("[exceptionHandle] ex", e);
 ErrorResult errorResult = new ErrorResult("USER-EX", e.getMessage());
 return new ResponseEntity<>(errorResult, HttpStatus.BAD_REQUEST);
 }

 @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
 @ExceptionHandler
 public ErrorResult exHandle(Exception e) {
 log.error("[exceptionHandle] ex", e);
 return new ErrorResult("EX", "내부 오류");
 }

 @GetMapping("/api2/members/{id}")
 public MemberDto getMember(@PathVariable("id") String id) {

 if (id.equals("ex")) {
 throw new RuntimeException("잘못된 사용자");
 }
 if (id.equals("bad")) {
 throw new IllegalArgumentException("잘못된 입력 값");
 }
 if (id.equals("user-ex")) {
 throw new UserException("사용자 오류");
 }
 }
}
```

```

 return new MemberDto(id, "hello " + id);
 }

 @Data
 @AllArgsConstructor
 static class MemberDto {
 private String memberId;
 private String name;
 }

}

```

### @ExceptionHandler 예외 처리 방법

`@ExceptionHandler` 애노테이션을 선언하고, 해당 컨트롤러에서 처리하고 싶은 예외를 지정해주면 된다. 해당 컨트롤러에서 예외가 발생하면 이 메서드가 호출된다. 참고로 지정한 예외 또는 그 예외의 자식 클래스는 모두 잡을 수 있다.

다음 예제는 `IllegalArgumentException` 또는 그 하위 자식 클래스를 모두 처리할 수 있다.

```

@ExceptionHandler(IllegalArgumentException.class)
public ErrorResult illegalExHandle(IllegalArgumentException e) {
 log.error("[exceptionHandle] ex", e);
 return new ErrorResult("BAD", e.getMessage());
}

```

### 우선순위

스프링의 우선순위는 항상 자세한 것이 우선권을 가진다. 예를 들어서 부모, 자식 클래스가 있고 다음과 같이 예외가 처리된다.

```

@ExceptionHandler(부모예외.class)
public String 부모예외처리()(부모예외 e) {}

@ExceptionHandler(자식예외.class)
public String 자식예외처리()(자식예외 e) {}

```

`@ExceptionHandler`에 지정한 부모 클래스는 자식 클래스까지 처리할 수 있다. 따라서 `자식예외` 가

발생하면 `부모예외처리()`, `자식예외처리()` 둘다 호출 대상이 된다. 그런데 둘 중 더 자세한 것이 우선권을 가지므로 `자식예외처리()` 가 호출된다. 물론 `부모예외` 가 호출되면 `부모예외처리()` 만 호출 대상이 되므로 `부모예외처리()` 가 호출된다.

## 다양한 예외

다음과 같이 다양한 예외를 한번에 처리할 수 있다.

```
@ExceptionHandler({AException.class, BException.class})
public String ex(Exception e) {
 log.info("exception e", e);
}
```

## 예외 생략

`@ExceptionHandler` 에 예외를 생략할 수 있다. 생략하면 메서드 파라미터의 예외가 지정된다.

```
@ExceptionHandler
public ResponseEntity<ErrorResult> userExHandle(UserException e) {}
```

## 파라미터와 응답

`@ExceptionHandler` 에는 마치 스프링의 컨트롤러의 파라미터 응답처럼 다양한 파라미터와 응답을 지정할 수 있다.

자세한 파라미터와 응답은 다음 공식 매뉴얼을 참고하자.

<https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-exceptionhandler-args>

## Postman 실행

<http://localhost:8080/api2/members/bad>

## IllegalArgumentException 처리

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(IllegalArgumentException.class)
public ErrorResult illegalExHandle(IllegalArgumentException e) {
 log.error("[exceptionHandle] ex", e);
```

```
 return new ErrorResult("BAD", e.getMessage());
```

```
}
```

### 실행 흐름

- 컨트롤러를 호출한 결과 `IllegalArgumentException` 예외가 컨트롤러 밖으로 던져진다.
- 예외가 발생했으므로 `ExceptionResolver` 가 작동한다. 가장 우선순위가 높은 `ExceptionHandlerExceptionResolver` 가 실행된다.
- `ExceptionHandlerExceptionResolver` 는 해당 컨트롤러에 `IllegalArgumentException` 을 처리할 수 있는 `@ExceptionHandler` 가 있는지 확인한다.
- `illegalExHandle()` 를 실행한다. `@RestController` 이므로 `illegalExHandle()` 에도 `@ResponseBody` 가 적용된다. 따라서 HTTP 컨버터가 사용되고, 응답이 다음과 같은 JSON으로 반환된다.
- `@ResponseStatus(HttpStatus.BAD_REQUEST)` 를 지정했으므로 HTTP 상태 코드 400으로 응답한다.

### 결과

```
{
 "code": "BAD",
 "message": "잘못된 입력 값"
}
```

### Postman 실행

<http://localhost:8080/api2/members/user-ex>

### UserException 처리

```
@ExceptionHandler
public ResponseEntity<ErrorResult> userExHandle(UserException e) {
 log.error("[exceptionHandle] ex", e);
 ErrorResult errorResult = new ErrorResult("USER-EX", e.getMessage());
 return new ResponseEntity<>(errorResult, HttpStatus.BAD_REQUEST);
}
```

- `@ExceptionHandler` 에 예외를 지정하지 않으면 해당 메서드 파라미터 예외를 사용한다. 여기서는 `UserException` 을 사용한다.
- `ResponseEntity` 를 사용해서 HTTP 메시지 바디에 직접 응답한다. 물론 HTTP 컨버터가 사용된다.

`ResponseEntity` 를 사용하면 HTTP 응답 코드를 프로그래밍해서 동적으로 변경할 수 있다. 앞서 살펴본 `@ResponseStatus` 는 애노테이션이므로 HTTP 응답 코드를 동적으로 변경할 수 없다.

### Postman 실행

- <http://localhost:8080/api2/members/ex>

### Exception

```
@ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
@ExceptionHandler
public ErrorResult exHandle(Exception e) {
 log.error("[exceptionHandle] ex", e);
 return new ErrorResult("EX", "내부 오류");
}
```

- `throw new RuntimeException("잘못된 사용자")` 이 코드가 실행되면서, 컨트롤러 밖으로 `RuntimeException` 이 던져진다.
- `RuntimeException` 은 `Exception` 의 자식 클래스이다. 따라서 이 메서드가 호출된다.
- `@ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)` 로 HTTP 상태 코드를 500으로 응답한다.

### 기타

#### HTML 오류 화면

다음과 같이 `ModelAndView` 를 사용해서 오류 화면(HTML)을 응답하는데 사용할 수도 있다.

```
@ExceptionHandler(ViewException.class)
public ModelAndView ex(ViewException e) {
 log.info("exception e", e);
 return new ModelAndView("error");
}
```

### API 예외 처리 - `@ControllerAdvice`

`@ExceptionHandler` 를 사용해서 예외를 깔끔하게 처리할 수 있게 되었지만, 정상 코드와 예외 처리 코드가 하나의 컨트롤러에 섞여 있다. `@ControllerAdvice` 또는 `@RestControllerAdvice` 를 사용하면 둘을 분리할 수 있다.

## ExControllerAdvice

```
package hello.exception.exhandler.advice;

import hello.exception.exception.UserException;
import hello.exception.exhandler.ErrorResult;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@Slf4j
@RestControllerAdvice
public class ExControllerAdvice {

 @ResponseStatus(HttpStatus.BAD_REQUEST)
 @ExceptionHandler(IllegalArgumentException.class)
 public ErrorResult illegalExHandle(IllegalArgumentException e) {
 log.error("[exceptionHandle] ex", e);
 return new ErrorResult("BAD", e.getMessage());
 }

 @ExceptionHandler
 public ResponseEntity<ErrorResult> userExHandle(UserException e) {
 log.error("[exceptionHandle] ex", e);
 ErrorResult errorResult = new ErrorResult("USER-EX", e.getMessage());
 return new ResponseEntity<>(errorResult, HttpStatus.BAD_REQUEST);
 }

 @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
 @ExceptionHandler
 public ErrorResult exHandle(Exception e) {
 log.error("[exceptionHandle] ex", e);
 }
}
```

```

 return new ErrorResult("EX", "내부 오류");
 }

}

```

## ApiExceptionV2Controller 코드에 있는 @ExceptionHandler 모두 제거

```

package hello.exception.exhandler;

import hello.exception.exception.UserException;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.*;

@Slf4j
@RestController
public class ApiExceptionV2Controller {

 @GetMapping("/api2/members/{id}")
 public MemberDto getMember(@PathVariable("id") String id) {

 if (id.equals("ex")) {
 throw new RuntimeException("잘못된 사용자");
 }
 if (id.equals("bad")) {
 throw new IllegalArgumentException("잘못된 입력 값");
 }
 if (id.equals("user-ex")) {
 throw new UserException("사용자 오류");
 }

 return new MemberDto(id, "hello " + id);
 }

 @Data
 @AllArgsConstructor
 static class MemberDto {

```

```
 private String memberId;
 private String name;
}

}
```

## Postman 실행

- <http://localhost:8080/api2/members/bad>
- <http://localhost:8080/api2/members/user-ex>
- <http://localhost:8080/api2/members/ex>

## @ControllerAdvice

- `@ControllerAdvice`는 대상으로 지정한 여러 컨트롤러에 `@ExceptionHandler`, `@InitBinder` 기능을 부여해주는 역할을 한다.
- `@ControllerAdvice`에 대상을 지정하지 않으면 모든 컨트롤러에 적용된다. (글로벌 적용)
- `@RestControllerAdvice`는 `@ControllerAdvice`와 같고, `@ResponseBody`가 추가되어 있다.  
`@Controller`, `@RestController`의 차이와 같다.

## 대상 컨트롤러 지정 방법

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class ExampleAdvice1 {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = {ControllerInterface.class,
AbstractController.class})
public class ExampleAdvice3 {}
```

<https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-controller-advice> (스프링 공식 문서 참고)

스프링 공식 문서 예제에서 보는 것처럼 특정 애노테이션이 있는 컨트롤러를 지정할 수 있고, 특정 패키지를 직접 지정할 수도 있다. 패키지 지정의 경우 해당 패키지와 그 하위에 있는 컨트롤러가 대상이 된다. 그리고

특정 클래스를 지정할 수도 있다.

대상 컨트롤러 지정을 생략하면 모든 컨트롤러에 적용된다.

### 정리

`@ExceptionHandler` 와 `@ControllerAdvice` 를 조합하면 예외를 깔끔하게 해결할 수 있다.

### 정리

# 10. 스프링 타입 컨버터

#인강/5. 스프링 MVC 2/강의#

## 목차

- 10. 스프링 타입 컨버터 - 프로젝트 생성
- 10. 스프링 타입 컨버터 - 스프링 타입 컨버터 소개
- 10. 스프링 타입 컨버터 - 타입 컨버터 - Converter
- 10. 스프링 타입 컨버터 - 컨버전 서비스 - ConversionService
- 10. 스프링 타입 컨버터 - 스프링에 Converter 적용하기
- 10. 스프링 타입 컨버터 - 뷰 템플릿에 컨버터 적용하기
- 10. 스프링 타입 컨버터 - 포맷터 - Formatter
- 10. 스프링 타입 컨버터 - 포맷터를 지원하는 컨버전 서비스
- 10. 스프링 타입 컨버터 - 포맷터 적용하기
- 10. 스프링 타입 컨버터 - 스프링이 제공하는 기본 포맷터
- 10. 스프링 타입 컨버터 - 정리

## 프로젝트 생성

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

<https://start.spring.io>

- 프로젝트 선택
  - Project: Gradle Project
  - Language: Java
  - Spring Boot: 2.4.x
- Project Metadata
  - Group: hello
  - Artifact: typeconverter
  - Name: typeconverter
  - Package name: **hello.typeconverter**
  - Packaging: **Jar**
  - Java: 11
- Dependencies: **Spring Web, Lombok , Thymeleaf**

## build.gradle

```

plugins {
 id 'org.springframework.boot' version '2.4.5'
 id 'io.spring.dependency-management' version '1.0.11.RELEASE'
 id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
 compileOnly {
 extendsFrom annotationProcessor
 }
}

repositories {
 mavenCentral()
}

dependencies {
 implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
 implementation 'org.springframework.boot:spring-boot-starter-web'
 compileOnly 'org.projectlombok:lombok'
 annotationProcessor 'org.projectlombok:lombok'
 testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
 useJUnitPlatform()
}

```

- 동작 확인
  - 기본 메인 클래스 실행( `TypeconverterApplication.main()` )
  - `http://localhost:8080` 호출해서 Whitelabel Error Page가 나오면 정상 동작

## 스프링 타입 컨버터 소개

문자를 숫자로 변환하거나, 반대로 숫자를 문자로 변환해야 하는 것처럼 애플리케이션을 개발하다 보면 타입을 변환해야 하는 경우가 상당히 많다.

다음 예를 보자.

### HelloController - 문자 타입을 숫자 타입으로 변경

```
package hello.typeconverter.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpServletRequest;

@RestController
public class HelloController {

 @GetMapping("/hello-v1")
 public String helloV1(HttpServletRequest request) {
 String data = request.getParameter("data"); //문자 타입 조회
 Integer intValue = Integer.valueOf(data); //숫자 타입으로 변경
 System.out.println("intValue = " + intValue);
 return "ok";
 }
}
```

### 실행

<http://localhost:8080/hello-v1?data=10>

### 분석

```
String data = request.getParameter("data")
```

HTTP 요청 파라미터는 모두 문자로 처리된다. 따라서 요청 파라미터를 자바에서 다른 타입으로 변환해서 사용하고 싶으면 다음과 같이 숫자 타입으로 변환하는 과정을 거쳐야 한다.

```
Integer intValue = Integer.valueOf(data)
```

이번에는 스프링 MVC가 제공하는 `@RequestParam`을 사용해보자.

## HelloController - 추가

```
@GetMapping("/hello-v2")
public String helloV2(@RequestParam Integer data) {
 System.out.println("data = " + data);
 return "ok";
}
```

### 실행

<http://localhost:8080/hello-v2?data=10>

앞서 보았듯이 HTTP 쿼리 스트링으로 전달하는 `data=10` 부분에서 10은 숫자 10이 아니라 문자 10이다.

스프링이 제공하는 `@RequestParam`을 사용하면 이 문자 10을 `Integer` 타입의 숫자 10으로 편리하게 받을 수 있다.

이것은 스프링이 중간에서 타입을 변환해주었기 때문이다.

이러한 예는 `@ModelAttribute`, `@PathVariable`에서도 확인할 수 있다.

### @ModelAttribute 타입 변환 예시

```
@ModelAttribute UserData data

class UserData {
 Integer data;
}
```

`@RequestParam` 와 같이, 문자 `data=10` 을 숫자 10으로 받을 수 있다.

### @PathVariable 타입 변환 예시

```
/users/{userId}
@PathVariable("data") Integer data
```

URL 경로는 문자다. `/users/10` → 여기서 10도 숫자 10이 아니라 그냥 문자 "10"이다. `data`를 `Integer` 타입으로 받을 수 있는 것도 스프링이 타입 변환을 해주기 때문이다.

## 스프링의 타입 변환 적용 예

- 스프링 MVC 요청 파라미터
  - `@RequestParam`, `@ModelAttribute`, `@PathVariable`
- `@Value` 등으로 YML 정보 읽기
- XML에 넣은 스프링 빈 정보를 변환
- 뷰를 렌더링 할 때

## 스프링과 타입 변환

이렇게 타입을 변환해야 하는 경우는 상당히 많다. 개발자가 직접 하나하나 타입 변환을 해야 한다면, 생각만 해도 괴로울 것이다.

스프링이 중간에 타입 변환기를 사용해서 타입을 `String` → `Integer`로 변환해주었기 때문에 개발자는 편리하게 해당 타입을 바로 받을 수 있다. 앞에서는 문자를 숫자로 변경하는 예시를 들었지만, 반대로 숫자를 문자로 변경하는 것도 가능하고, `Boolean` 타입을 숫자로 변경하는 것도 가능하다. 만약 개발자가 새로운 타입을 만들어서 변환하고 싶으면 어떻게 하면 될까?

## 컨버터 인터페이스

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {
 T convert(S source);
}
```

스프링은 확장 가능한 컨버터 인터페이스를 제공한다.

개발자는 스프링에 추가적인 타입 변환이 필요하면 이 컨버터 인터페이스를 구현해서 등록하면 된다.

이 컨버터 인터페이스는 모든 타입에 적용할 수 있다. 필요하면  $X \rightarrow Y$  타입으로 변환하는 컨버터 인터페이스를 만들고, 또  $Y \rightarrow X$  타입으로 변환하는 컨버터 인터페이스를 만들어서 등록하면 된다.

예를 들어서 문자로 "true" 가 오면 `Boolean` 타입으로 받고 싶으면 `String` → `Boolean` 타입으로 변환되도록 컨버터 인터페이스를 만들어서 등록하고, 반대로 적용하고 싶으면 `Boolean` → `String` 타입으로 변환되도록 컨버터를 추가로 만들어서 등록하면 된다.

## 참고

과거에는 `PropertyEditor`라는 것으로 타입을 변환했다. `PropertyEditor`는 동시성 문제가 있어서 타입을 변환할 때마다 객체를 계속 생성해야 하는 단점이 있다. 지금은 `Converter`의 등장으로 해당 문제들이 해결되었고, 가능 확장이 필요하면 `Converter`를 사용하면 된다.

실제 코드를 통해서 타입 컨버터를 이해해보자.

## 타입 컨버터 - Converter

타입 컨버터를 어떻게 사용하는지 코드로 알아보자.

타입 컨버터를 사용하려면 `org.springframework.core.convert.converter.Converter` 인터페이스를 구현하면 된다.

### 주의

`Converter`라는 이름의 인터페이스가 많으니 조심해야 한다.

`org.springframework.core.convert.converter.Converter`를 사용해야 한다.

### 컨버터 인터페이스

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {
 T convert(S source);
}
```

먼저 가장 단순한 형태인 문자를 숫자로 바꾸는 타입 컨버터를 만들어보자.

### StringToIntegerConverter - 문자를 숫자로 변환하는 타입 컨버터

```
package hello.typeconverter.converter;

import lombok.extern.slf4j.Slf4j;
import org.springframework.core.convert.converter.Converter;

@Slf4j
public class StringToIntegerConverter implements Converter<String, Integer> {

 @Override
```

```
public Integer convert(String source) {
 log.info("convert source={}, source");
 return Integer.valueOf(source);
}
}
```

`String` → `Integer`로 변환하기 때문에 소스가 `String`이 된다. 이 문자를 `Integer.valueOf(source)`를 사용해서 숫자로 변경한 다음에 변경된 숫자를 반환하면 된다.

### **IntegerToStringConverter - 숫자를 문자로 변환하는 타입 컨버터**

```
package hello.typeconverter.converter;

import lombok.extern.slf4j.Slf4j;
import org.springframework.core.convert.converter.Converter;

@Slf4j
public class IntegerToStringConverter implements Converter<Integer, String> {
 @Override
 public String convert(Integer source) {
 log.info("convert source={}, source");
 return String.valueOf(source);
 }
}
```

이번에는 숫자를 문자로 변환하는 타입 컨버터이다. 앞의 컨버터와 반대의 일을 한다. 이번에는 숫자가 입력되기 때문에 소스가 `Integer`가 된다. `String.valueOf(source)`를 사용해서 문자로 변경한 다음 변경된 문자를 반환하면 된다.

테스트 코드를 통해서 타입 컨버터가 어떻게 동작하는지 확인해보자.

### **ConverterTest - 타입 컨버터 테스트 코드**

```
package hello.typeconverter.converter;
```

```

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.*;

class ConverterTest {

 @Test
 void stringToInteger() {
 StringToIntegerConverter converter = new StringToIntegerConverter();
 Integer result = converter.convert("10");
 assertThat(result).isEqualTo(10);
 }

 @Test
 void integerToString() {
 IntegerToStringConverter converter = new IntegerToStringConverter();
 String result = converter.convert(10);
 assertThat(result).isEqualTo("10");
 }
}

```

## 사용자 정의 타입 컨버터

타입 컨버터 이해를 돋기 위해 조금 다른 컨버터를 준비해보았다.

127.0.0.1:8080 과 같은 IP, PORT를 입력하면 IpPort 객체로 변환하는 컨버터를 만들어보자.

## IpPort

```

package hello.typeconverter.type;

import lombok.EqualsAndHashCode;
import lombok.Getter;

@Getter
@EqualsAndHashCode

```

```

public class IpPort {

 private String ip;
 private int port;

 public IpPort(String ip, int port) {
 this.ip = ip;
 this.port = port;
 }

}

```

롬복의 `@EqualsAndHashCode` 를 넣으면 모든 필드를 사용해서 `equals()`, `hashcode()` 를 생성한다.  
따라서 모든 필드의 값이 같다면 `a.equals(b)` 의 결과가 참이 된다.

## StringToIpPortConverter - 컨버터

```

package hello.typeconverter.converter;

import hello.typeconverter.type.IpPort;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.convert.converter.Converter;

@Slf4j
public class StringToIpPortConverter implements Converter<String, IpPort> {

 @Override
 public IpPort convert(String source) {
 log.info("convert source={}", source);
 String[] split = source.split(":");
 String ip = split[0];
 int port = Integer.parseInt(split[1]);

 return new IpPort(ip, port);
 }
}

```

127.0.0.1:8080 같은 문자를 입력하면 `IpPort` 객체를 만들어 반환한다.

## IpPortToStringConverter

```
package hello.typeconverter.converter;

import hello.typeconverter.type.IpPort;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.convert.converter.Converter;

@Slf4j
public class IpPortToStringConverter implements Converter<IpPort, String> {

 @Override
 public String convert(IpPort source) {
 log.info("convert source={}", source);
 return source.getIp() + ":" + source.getPort();
 }
}
```

`IpPort` 객체를 입력하면 127.0.0.1:8080 같은 문자를 반환한다.

## ConverterTest - `IpPort` 컨버터 테스트 추가

```
@Test
void stringToIpPort() {
 StringToIpPortConverter converter = new StringToIpPortConverter();
 String source = "127.0.0.1:8080";
 IpPort result = converter.convert(source);
 assertThat(result).isEqualTo(new IpPort("127.0.0.1", 8080));
}

@Test
void ipPortToString() {
 IpPortToStringConverter converter = new IpPortToStringConverter();
 IpPort source = new IpPort("127.0.0.1", 8080);
 String result = converter.convert(source);
 assertThat(result).isEqualTo("127.0.0.1:8080");
}
```

```
}
```

타입 컨버터 인터페이스가 단순해서 이해하기 어렵지 않을 것이다.

그런데 이렇게 타입 컨버터를 하나하나 직접 사용하면, 개발자가 직접 컨버팅 하는 것과 큰 차이가 없다.

타입 컨버터를 등록하고 관리하면서 편리하게 변환 기능을 제공하는 역할을 하는 무언가가 필요하다.

## 참고

스프링은 용도에 따라 다양한 방식의 타입 컨버터를 제공한다.

`Converter` → 기본 타입 컨버터

`ConverterFactory` → 전체 클래스 계층 구조가 필요할 때

`GenericConverter` → 정교한 구현, 대상 필드의 애노테이션 정보 사용 가능

`ConditionalGenericConverter` → 특정 조건이 참인 경우에만 실행

자세한 내용은 공식 문서를 참고하자.

<https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#core-convert>

## 참고

스프링은 문자, 숫자, 불린, Enum등 일반적인 타입에 대한 대부분의 컨버터를 기본으로 제공한다. IDE에서

`Converter`, `ConverterFactory`, `GenericConverter`의 구현체를 찾아보면 수 많은 컨버터를 확인할 수 있다.

## 컨버전 서비스 - `ConversionService`

이렇게 타입 컨버터를 하나하나 직접 찾아서 타입 변환에 사용하는 것은 매우 불편하다. 그래서 스프링은 개별 컨버터를 모아두고 그것들을 묶어서 편리하게 사용할 수 있는 기능을 제공하는데, 이것이 바로 컨버전 서비스(`ConversionService`)이다.

### `ConversionService` 인터페이스

```
package org.springframework.core.convert;

import org.springframework.lang.Nullable;
```

```

public interface ConversionService {

 boolean canConvert(@Nullable Class<?> sourceType, Class<?> targetType);
 boolean canConvert(@Nullable TypeDescriptor sourceType, TypeDescriptor
targetType);

 <T> T convert(@Nullable Object source, Class<T> targetType);
 Object convert(@Nullable Object source, @Nullable TypeDescriptor sourceType,
TypeDescriptor targetType);

}

```

컨버전 서비스 인터페이스는 단순히 컨버팅이 가능한가? 확인하는 기능과, 컨버팅 기능을 제공한다.

사용 예를 확인해보자.

### **ConversionServiceTest - 컨버전 서비스 테스트 코드**

```

package hello.typeconverter.converter;

import hello.typeconverter.type.IpPort;
import org.junit.jupiter.api.Test;
import org.springframework.core.convert.support.DefaultConversionService;

import static org.assertj.core.api.Assertions.*;

public class ConversionServiceTest {

 @Test
 void conversionService() {
 //등록
 DefaultConversionService conversionService = new
DefaultConversionService();
 conversionService.addConverter(new StringToIntegerConverter());
 conversionService.addConverter(new IntegerToStringConverter());
 conversionService.addConverter(new StringToIpPortConverter());
 conversionService.addConverter(new IpPortToStringConverter());

 //사용
 }
}

```

```

 assertThat(conversionService.convert("10",
Integer.class)).isEqualTo(10);
 assertThat(conversionService.convert(10,
String.class)).isEqualTo("10");

 IpPort ipPort = conversionService.convert("127.0.0.1:8080",
IpPort.class);
 assertThat(ipPort).isEqualTo(new IpPort("127.0.0.1", 8080));

 String ipPortString = conversionService.convert(new IpPort("127.0.0.1",
8080), String.class);
 assertThat(ipPortString).isEqualTo("127.0.0.1:8080");

 }
}

```

`DefaultConversionService`는 `ConversionService` 인터페이스를 구현했는데, 추가로 컨버터를 등록하는 기능도 제공한다.

### 등록과 사용 분리

컨버터를 등록할 때는 `StringToIntegerConverter` 같은 타입 컨버터를 명확하게 알아야 한다. 반면에 컨버터를 사용하는 입장에서는 타입 컨버터를 전혀 몰라도 된다. 타입 컨버터들은 모두 컨버전 서비스 내부에 숨어서 제공된다. 따라서 타입을 변환을 원하는 사용자는 컨버전 서비스 인터페이스에만 의존하면 된다. 물론 컨버전 서비스를 등록하는 부분과 사용하는 부분을 분리하고 의존관계 주입을 사용해야 한다.

### 컨버전 서비스 사용

```
Integer value = conversionService.convert("10", Integer.class)
```

### 인터페이스 분리 원칙 - ISP(Interface Segregation Principle)

인터페이스 분리 원칙은 클라이언트가 자신이 이용하지 않는 메서드에 의존하지 않아야 한다.

`DefaultConversionService`는 다음 두 인터페이스를 구현했다.

- `ConversionService` : 컨버터 사용에 초점
- `ConverterRegistry` : 컨버터 등록에 초점

이렇게 인터페이스를 분리하면 컨버터를 사용하는 클라이언트와 컨버터를 등록하고 관리하는 클라이언트의 관심사를 명확하게 분리할 수 있다. 특히 컨버터를 사용하는 클라이언트는 `ConversionService`만

의존하면 되므로, 컨버터를 어떻게 등록하고 관리하는지는 전혀 몰라도 된다. 결과적으로 컨버터를 사용하는 클라이언트는 꼭 필요한 메서드만 알게된다. 이렇게 인터페이스를 분리하는 것을 ISP 라 한다.

ISP 참고: <https://ko.wikipedia.org/wiki/>

%EC%9D%B8%ED%84%B0%ED%8E%98%EC%9D%B4%EC%8A%A4\_%EB%B6%84%EB%  
A6%AC\_%EC%9B%90%EC%B9%99

스프링은 내부에서 `ConversionService` 를 사용해서 타입을 변환한다. 예를 들어서 앞서 살펴본 `@RequestParam` 같은 곳에서 이 기능을 사용해서 타입을 변환한다.

이제 컨버전 서비스를 스프링에 적용해보자.

## 스프링에 Converter 적용하기

웹 애플리케이션에 `Converter` 를 적용해보자.

### WebConfig - 컨버터 등록

```
package hello.typeconverter;

import hello.typeconverter.converter.IntegerToStringConverter;
import hello.typeconverter.converter.IpPortToStringConverter;
import hello.typeconverter.converter.StringToIntegerConverter;
import hello.typeconverter.converter.StringToIpPortConverter;
import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

 @Override
 public void addFormatters(FormatterRegistry registry) {
 registry.addConverter(new StringToIntegerConverter());
 registry.addConverter(new IntegerToStringConverter());
 registry.addConverter(new StringToIpPortConverter());
 registry.addConverter(new IpPortToStringConverter());
 }
}
```

```
}
```

스프링은 내부에서 `ConversionService`를 제공한다. 우리는 `WebMvcConfigurer`가 제공하는 `addFormatters()`를 사용해서 추가하고 싶은 컨버터를 등록하면 된다. 이렇게 하면 스프링은 내부에서 사용하는 `ConversionService`에 컨버터를 추가해준다.

등록한 컨버터가 잘 동작하는지 확인해보자.

### HelloController - 기존 코드

```
@GetMapping("/hello-v2")
public String helloV2(@RequestParam Integer data) {
 System.out.println("data = " + data);
 return "ok";
}
```

#### 실행

<http://localhost:8080/hello-v2?data=10>

#### 실행 로그

```
StringToIntegerConverter : convert source=10
data = 10
```

?data=10의 쿼리 파라미터는 문자이고 이것을 `Integer data`로 변환하는 과정이 필요하다.

실행해보면 직접 등록한 `StringToIntegerConverter`가 작동하는 로그를 확인할 수 있다.

그런데 생각해보면 `StringToIntegerConverter`를 등록하기 전에도 이 코드는 잘 수행되었다. 그것은 스프링이 내부에서 수 많은 기본 컨버터들을 제공하기 때문이다. 컨버터를 추가하면 추가한 컨버터가 기본 컨버터 보다 높은 우선순위를 가진다.

이번에는 직접 정의한 타입인 `IpPort`를 사용해보자.

### HelloController - 추가

```
@GetMapping("/ip-port")
public String ipPort(@RequestParam IpPort ipPort) {
 System.out.println("ipPort IP = " + ipPort.getIp());
 System.out.println("ipPort PORT = " + ipPort.getPort());
 return "ok";
}
```

## 실행

<http://localhost:8080/ip-port?ipPort=127.0.0.1:8080>

## 실행 로그

```
StringToIpPortConverter : convert source=127.0.0.1:8080
ipPort IP = 127.0.0.1
ipPort PORT = 8080
```

?ipPort=127.0.0.1:8080 쿼리 스트링이 @RequestParam IpPort ipPort에서 객체 타입으로 잘 변환 된 것을 확인할 수 있다.

## 처리 과정

@RequestParam 은 @RequestParam 을 처리하는 ArgumentResolver 인 RequestParamMethodArgumentResolver에서 ConversionService 를 사용해서 타입을 변환한다. 부모 클래스와 다양한 외부 클래스를 호출하는 등 복잡한 내부 과정을 거치기 때문에 대략 이렇게 처리되는 것으로 이해해도 충분하다. 만약 더 깊이있게 확인하고 싶으면 IpPortConverter 에 디버그 브레이크 포인트를 걸어서 확인해보자.

## 뷰 템플릿에 컨버터 적용하기

이번에는 뷰 템플릿에 컨버터를 적용하는 방법을 알아보자.

타임리프는 렌더링 시에 컨버터를 적용해서 렌더링 하는 방법을 편리하게 지원한다.

이전까지는 문자를 객체로 변환했다면, 이번에는 그 반대로 객체를 문자로 변환하는 작업을 확인할 수 있다.

## ConverterController

```
package hello.typeconverter.controller;
```

```

import hello.typeconverter.type.IpPort;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class ConverterController {

 @GetMapping("/converter-view")
 public String converterView(Model model) {
 model.addAttribute("number", 10000);
 model.addAttribute("ipPort", new IpPort("127.0.0.1", 8080));
 return "converter-view";
 }
}

```

Model에 숫자 10000 와 ipPort 객체를 담아서 뷰 템플릿에 전달한다.

resources/templates/converter-view.html

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="UTF-8">
 <title>Title</title>
</head>
<body>

 ${number}:
 ${{number}}:
 ${ipPort}:
 ${{ipPort}}:

</body>

```

```
</html>
```

타임리프는  `${...}`  를 사용하면 자동으로 컨버전 서비스를 사용해서 변환된 결과를 출력해준다. 물론 스프링과 통합 되어서 스프링이 제공하는 컨버전 서비스를 사용하므로, 우리가 등록한 컨버터들을 사용할 수 있다.

변수 표현식 :  `${...}`

컨버전 서비스 적용 :  `${...}`

## 실행

<http://localhost:8080/converter-view>

## 실행 결과

- `${number}: 10000`
- `${{number}}: 10000`
- `${ipPort}: hello.typeconverter.type.IpPort@59cb0946`
- `${{ipPort}}: 127.0.0.1:8080`

## 실행 결과 로그

```
IntegerToStringConverter : convert source=10000
IpPortToStringConverter : convert
source=hello.typeconverter.type.IpPort@59cb0946
```

- `${{number}}`  : 뷰 템플릿은 데이터를 문자로 출력한다. 따라서 컨버터를 적용하게 되면  `Integer`  타입인  `10000`  을  `String`  타입으로 변환하는 컨버터인  `IntegerToStringConverter`  를 실행하게 된다. 이 부분은 컨버터를 실행하지 않아도 타임리프가 숫자를 문자로 자동으로 변환하기 때문에 컨버터를 적용할 때와 하지 않을 때가 같다.
- `${{ipPort}}`  : 뷰 템플릿은 데이터를 문자로 출력한다. 따라서 컨버터를 적용하게 되면  `IpPort`  타입을  `String`  타입으로 변환해야 하므로  `IpPortToStringConverter`  가 적용된다. 그 결과  `127.0.0.1:8080`  가 출력된다.

## 폼에 적용하기

이번에는 컨버터를 폼에 적용해보자.

## ConverterController - 코드 추가

```
package hello.typeconverter.controller;

import hello.typeconverter.type.IpPort;
import lombok.Data;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

@Controller
public class ConverterController {

 @GetMapping("/converter-view")
 public String converterView(Model model) {
 model.addAttribute("number", 10000);
 model.addAttribute("ipPort", new IpPort("127.0.0.1", 8080));
 return "converter-view";
 }

 @GetMapping("/converter/edit")
 public String converterForm(Model model) {

 IpPort ipPort = new IpPort("127.0.0.1", 8080);
 Form form = new Form(ipPort);

 model.addAttribute("form", form);
 return "converter-form";
 }

 @PostMapping("/converter/edit")
 public String converterEdit(@ModelAttribute Form form, Model model) {
 IpPort ipPort = form.getIpPort();
 model.addAttribute("ipPort", ipPort);
 return "converter-view";
 }
}
```

```

}

@Data
static class Form {
 private IpPort ipPort;

 public Form(IpPort ipPort) {
 this.ipPort = ipPort;
 }
}

```

`Form` 객체를 데이터를 전달하는 품 객체로 사용한다.

- GET /converter/edit : `IpPort` 를 뷰 템플릿 품에 출력한다.
- POST /converter/edit : 뷰 템플릿 품의 `IpPort` 정보를 받아서 출력한다.

`resources/templates/converter-form.html`

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="UTF-8">
 <title>Title</title>
</head>
<body>

<form th:object="${form}" th:method="post">
 th:field <input type="text" th:field="*{ipPort}">

 th:value <input type="text" th:value="*{ipPort}">(보여주기 용도)

 <input type="submit"/>
</form>

</body>
</html>

```

타임리프의 `th:field` 는 앞서 설명했듯이 `id`, `name` 를 출력하는 등 다양한 기능이 있는데, 여기에 컨버전 서비스도 함께 적용된다.

## 실행

<http://localhost:8080/converter/edit>

- GET /converter/edit
  - `th:field` 가 자동으로 컨버전 서비스를 적용해주어서 `${{ipPort}}` 처럼 적용이 되었다. 따라서 `IpPort`  $\rightarrow$  `String` 으로 변환된다.
- POST /converter/edit
  - `@ModelAttribute` 를 사용해서 `String`  $\rightarrow$  `IpPort` 로 변환된다.

## 포맷터 - Formatter

`Converter` 는 입력과 출력 타입에 제한이 없는, 범용 타입 변환 기능을 제공한다.

이번에는 일반적인 웹 애플리케이션 환경을 생각해보자. 불린 타입을 숫자로 바꾸는 것 같은 범용 기능 보다는 개발자 입장에서는

문자를 다른 타입으로 변환하거나, 다른 타입을 문자로 변환하는 상황이 대부분이다.

앞서 살펴본 예제들을 떠올려 보면 문자를 다른 객체로 변환하거나 객체를 문자로 변환하는 일이 대부분이다.

### 웹 애플리케이션에서 객체를 문자로, 문자를 객체로 변환하는 예

- 화면에 숫자를 출력해야 하는데, `Integer`  $\rightarrow$  `String` 출력 시점에 숫자 `1000`  $\rightarrow$  문자 `"1,000"` 이렇게 1000 단위에 쉼표를 넣어서 출력하거나, 또는 `"1,000"` 라는 문자를 `1000` 이라는 숫자로 변경해야 한다.
- 날짜 객체를 문자인 `"2021-01-01 10:50:11"` 와 같이 출력하거나 또는 그 반대의 상황

## Locale

여기에 추가로 날짜 숫자의 표현 방법은 `Locale` 현지화 정보가 사용될 수 있다.

이렇게 객체를 특정한 포맷에 맞추어 문자로 출력하거나 또는 그 반대의 역할을 하는 것에 특화된 기능이 바로 포맷터(`Formatter`)이다. 포맷터는 컨버터의 특별한 버전으로 이해하면 된다.

## Converter vs Formatter

- `Converter` 는 범용(객체  $\rightarrow$  객체)
- `Formatter` 는 문자에 특화(객체  $\rightarrow$  문자, 문자  $\rightarrow$  객체) + 현지화(Locale)
  - `Converter` 의 특별한 버전

## 포맷터 - Formatter 만들기

포맷터(Formatter)는 객체를 문자로 변경하고, 문자를 객체로 변경하는 두 가지 기능을 모두 수행한다.

- String print(T object, Locale locale) : 객체를 문자로 변경한다.
- T parse(String text, Locale locale) : 문자를 객체로 변경한다.

## Formatter 인터페이스

```
public interface Printer<T> {
 String print(T object, Locale locale);
}

public interface Parser<T> {
 T parse(String text, Locale locale) throws ParseException;
}

public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

숫자 1000 을 문자 "1,000" 으로 그러니까, 1000 단위로 쉼표가 들어가는 포맷을 적용해보자. 그리고 그 반대도 처리해주는 포맷터를 만들어보자.

## MyNumberFormatter

```
package hello.typeconverter.formatter;

import lombok.extern.slf4j.Slf4j;
import org.springframework.format.Formatter;

import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

@Slf4j
public class MyNumberFormatter implements Formatter<Number> {
```

```

@Override
public Number parse(String text, Locale locale) throws ParseException {
 log.info("text={}, locale={}", text, locale);
 NumberFormat format = NumberFormat.getInstance(locale);
 return format.parse(text);
}

@Override
public String print(Number object, Locale locale) {
 log.info("object={}, locale={}", object, locale);
 return NumberFormat.getInstance(locale).format(object);
}

```

"1,000"처럼 숫자 중간의 쉼표를 적용하려면 자바가 기본으로 제공하는 `NumberFormat` 객체를 사용하면 된다. 이 객체는 `Locale` 정보를 활용해서 나라별로 다른 숫자 포맷을 만들어준다.

`parse()`를 사용해서 문자를 숫자로 변환한다. 참고로 `Number` 타입은 `Integer`, `Long`과 같은 숫자 타입의 부모 클래스이다.

`print()`를 사용해서 객체를 문자로 변환한다.

잘 동작하는지 테스트 코드를 만들어보자.

## MyNumberFormatterTest

```

package hello.typeconverter.formatter;

import org.junit.jupiter.api.Test;

import java.text.ParseException;
import java.util.Locale;

import static org.assertj.core.api.Assertions.*;

class MyNumberFormatterTest {

```

```

MyNumberFormatter formatter = new MyNumberFormatter();

@Test
void parse() throws ParseException {
 Number result = formatter.parse("1,000", Locale.KOREA);
 assertThat(result).isEqualTo(1000L); //Long 타입 주의
}

@Test
void print() {
 String result = formatter.print(1000, Locale.KOREA);
 assertThat(result).isEqualTo("1,000");
}

```

`parse()`의 결과가 `Long`이기 때문에 `isEqualTo(1000L)`을 통해 비교할 때 마지막에 `L`을 넣어주어야 한다.

## 실행 결과 로그

```

MyNumberFormatter - text=1,000, locale=ko_KR
MyNumberFormatter - object=1000, locale=ko_KR

```

## 참고

스프링은 용도에 따라 다양한 방식의 포맷터를 제공한다.

`Formatter` 포맷터

`AnnotationFormatterFactory` 필드의 타입이나 애노테이션 정보를 활용할 수 있는 포맷터

자세한 내용은 공식 문서를 참고하자.

<https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#format>

## 포맷터를 지원하는 컨버전 서비스

컨버전 서비스에는 컨버터만 등록할 수 있고, 포맷터를 등록할 수는 없다. 그런데 생각해보면 포맷터는 객체

→ 문자, 문자 → 객체로 변환하는 특별한 컨버터일 뿐이다.

포맷터를 지원하는 컨버전 서비스를 사용하면 컨버전 서비스에 포맷터를 추가할 수 있다. 내부에서 어댑터 패턴을 사용해서 `Formatter` 가 `Converter` 처럼 동작하도록 지원한다.

`FormattingConversionService` 는 포맷터를 지원하는 컨버전 서비스이다.

`DefaultFormattingConversionService` 는 `FormattingConversionService` 에 기본적인 통화, 숫자 관련 몇가지 기본 포맷터를 추가해서 제공한다.

```
package hello.typeconverter.formatter;

import hello.typeconverter.converter.IpPortToStringConverter;
import hello.typeconverter.converter.StringToIpPortConverter;
import hello.typeconverter.type.IpPort;
import org.junit.jupiter.api.Test;
import org.springframework.format.support.FormattingConversionService;

import static org.assertj.core.api.Assertions.assertThat;

public class FormattingConversionServiceTest {

 @Test
 void formattingConversionService() {
 DefaultFormattingConversionService conversionService = new
DefaultFormattingConversionService();
 //컨버터 등록
 conversionService.addConverter(new StringToIpPortConverter());
 conversionService.addConverter(new IpPortToStringConverter());
 //포맷터 등록
 conversionService.addFormatter(new MyNumberFormatter());

 //컨버터 사용
 IpPort ipPort = conversionService.convert("127.0.0.1:8080",
IpPort.class);
 assertThat(ipPort).isEqualTo(new IpPort("127.0.0.1", 8080));
 //포맷터 사용
 assertThat(conversionService.convert(1000,
```

```
String.class)).isEqualTo("1,000");
 assertThat(conversionService.convert("1,000",
Long.class)).isEqualTo(1000L);
 }
}
```

## DefaultFormattingConversionService 상속 관계

FormattingConversionService 는 ConversionService 관련 기능을 상속받기 때문에 결과적으로 컨버터도 포맷터도 모두 등록할 수 있다. 그리고 사용할 때는 ConversionService 가 제공하는 convert 를 사용하면 된다.

추가로 스프링 부트는 DefaultFormattingConversionService 를 상속 받은 WebConversionService 를 내부에서 사용한다.

## 포맷터 적용하기

포맷터를 웹 애플리케이션에 적용해보자.

### WebConfig - 수정

```
package hello.typeconverter;

import hello.typeconverter.converter.IntegerToStringConverter;
import hello.typeconverter.converter.IpPortToStringConverter;
import hello.typeconverter.converter.StringToIntegerConverter;
import hello.typeconverter.converter.StringToIpPortConverter;
import hello.typeconverter.formatter.MyNumberFormatter;
import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

 @Override
 public void addFormatters(FormatterRegistry registry) {
```

```

//주석처리 우선순위
//registry.addConverter(new StringToIntegerConverter());
//registry.addConverter(new IntegerToStringConverter());
registry.addConverter(new StringToIpPortConverter());
registry.addConverter(new IpPortToStringConverter());

//추가
registry.addFormatter(new MyNumberFormatter());
}

}

```

**주의** `StringToIntegerConverter`, `IntegerToStringConverter`를 꼭 주석처리 하자.

`MyNumberFormatter`도 숫자 → 문자, 문자 → 숫자로 변경하기 때문에 둘의 기능이 겹친다. 우선순위는 컨버터가 우선하므로 포맷터가 적용되지 않고, 컨버터가 적용된다.

### 실행 - 객체 → 문자

<http://localhost:8080/converter-view>

- `${number}: 10000`
- `${{number}}: 10,000`

컨버전 서비스를 적용한 결과 `MyNumberFormatter` 가 적용되어서 `10,000` 문자가 출력된 것을 확인할 수 있다.

### 실행 - 문자 → 객체

<http://localhost:8080/hello-v2?data=10,000>

### 실행 로그

```

MyNumberFormatter : text=10,000, locale=ko_KR
data = 10000

```

"10,000" 이라는 포맷팅 된 문자가 `Integer` 타입의 숫자 10000으로 정상 변환 된 것을 확인할 수 있다.

## 스프링이 제공하는 기본 포맷터

스프링은 자바에서 기본으로 제공하는 타입들에 대해 수 많은 포맷터를 기본으로 제공한다.

IDE에서 `Formatter` 인터페이스의 구현 클래스를 찾아보면 수 많은 날짜나 시간 관련 포맷터가 제공되는 것을 확인할 수 있다.

그런데 포맷터는 기본 형식이 지정되어 있기 때문에, 객체의 각 필드마다 다른 형식으로 포맷을 지정하기는 어렵다.

스프링은 이런 문제를 해결하기 위해 애노테이션 기반으로 원하는 형식을 지정해서 사용할 수 있는 매우 유용한 포맷터 두 가지를 기본으로 제공한다.

- `@NumberFormat` : 숫자 관련 형식 지정 포맷터 사용, `NumberFormatAnnotationFormatterFactory`
- `@DateTimeFormat` : 날짜 관련 형식 지정 포맷터 사용,  
`Jsr310DateFormatAnnotationFormatterFactory`

예제를 통해서 알아보자.

### FormatterController

```
package hello.typeconverter.controller;

import lombok.Data;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.format.annotation.NumberFormat;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

import java.time.LocalDateTime;

@Controller
public class FormatterController {

 @GetMapping("/formatter/edit")
 public String formatterForm(Model model) {
```

```

 Form form = new Form();
 form.setNumber(10000);
 form.setLocalDateTime(LocalDateTime.now());

 model.addAttribute("form", form);
 return "formatter-form";
 }

 @PostMapping("/formatter/edit")
 public String formatterEdit(@ModelAttribute Form form) {
 return "formatter-view";
 }

 @Data
 static class Form {

 @NumberFormat(pattern = "###,###")
 private Integer number;

 @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
 private LocalDateTime localDateTime;
 }
}

```

templates/formatter-form.html

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="UTF-8">
 <title>Title</title>
</head>
<body>

<form th:object="${form}" th:method="post">
 number <input type="text" th:field="*{number}">

 localDateTime <input type="text" th:field="*{localDateTime}">


```

```
<input type="submit"/>
</form>

</body>
</html>
```

templates/formatter-view.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="UTF-8">
 <title>Title</title>
</head>
<body>

 ${form.number}:
 ${{form.number}}:
 ${form.localDateTime}:
 ${{form.localDateTime}}:

</body>
</html>
```

## 실행

<http://localhost:8080/formatter/edit>

실행해보면 지정한 포맷으로 출력된 것을 확인할 수 있다.

## 결과

- \${form.number}: 10000

- \${form.number}: 10,000
- \${form.localDateTime}: 2021-01-01T00:00:00
- \${form.localDate}: 2021-01-01 00:00:00

## 참고

@NumberFormat, @DateTimeFormat의 자세한 사용법이 궁금한 분들은 다음 링크를 참고하거나 관련 애노테이션을 검색해보자.

<https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#format-CustomFormatAnnotations>

## 정리

컨버터를 사용하든, 포맷터를 사용하든 등록 방법은 다르지만, 사용할 때는 컨버전 서비스를 통해서 일관성 있게 사용할 수 있다.

### 주의!

메시지 컨버터(`HttpMessageConverter`)에는 컨버전 서비스가 적용되지 않는다.

특히 객체를 JSON으로 변환할 때 메시지 컨버터를 사용하면서 이 부분을 많이 오해하는데,

`HttpMessageConverter`의 역할은 HTTP 메시지 바디의 내용을 객체로 변환하거나 객체를 HTTP 메시지 바디에 입력하는 것이다. 예를 들어서 JSON을 객체로 변환하는 메시지 컨버터는 내부에서 Jackson 같은 라이브러리를 사용한다. 객체를 JSON으로 변환한다면 그 결과는 이 라이브러리에 달린 것이다. 따라서 JSON 결과로 만들어지는 숫자나 날짜 포맷을 변경하고 싶으면 해당 라이브러리가 제공하는 설정을 통해서 포맷을 지정해야 한다. 결과적으로 이것은 컨버전 서비스와 전혀 관계가 없다.

컨버전 서비스는 `@RequestParam`, `@ModelAttribute`, `@PathVariable`, 뷰 템플릿 등에서 사용할 수 있다.

# 11. 파일 업로드

#인강/5. 스프링 MVC 2/강의#

## 목차

- 11. 파일 업로드 - 파일 업로드 소개
- 11. 파일 업로드 - 프로젝트 생성
- 11. 파일 업로드 - 서블릿과 파일 업로드1
- 11. 파일 업로드 - 서블릿과 파일 업로드2
- 11. 파일 업로드 - 스프링과 파일 업로드
- 11. 파일 업로드 - 예제로 구현하는 파일 업로드, 다운로드
- 11. 파일 업로드 - 정리

## 파일 업로드 소개

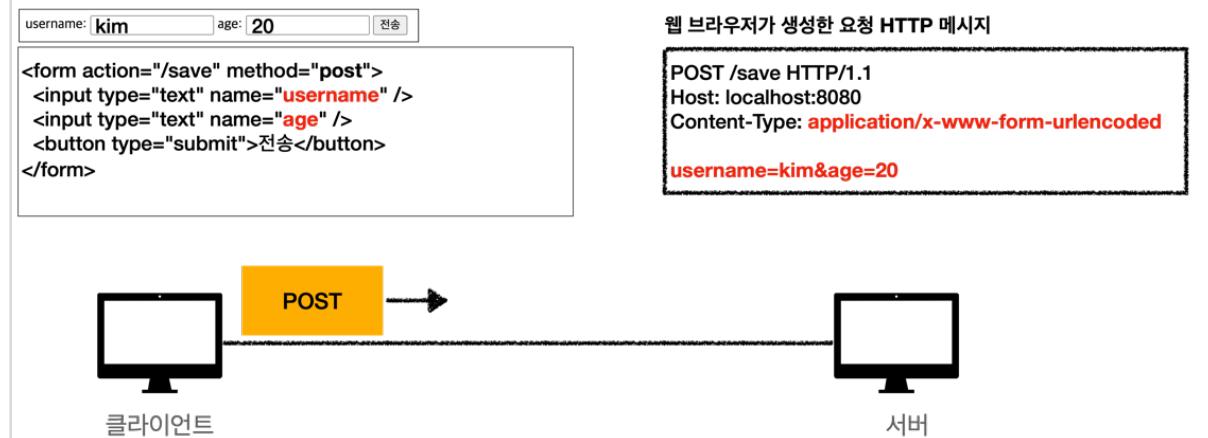
일반적으로 사용하는 HTML Form을 통한 파일 업로드를 이해하려면 먼저 폼을 전송하는 다음 두 가지 방식의 차이를 이해해야 한다.

### HTML 폼 전송 방식

- application/x-www-form-urlencoded
- multipart/form-data

#### application/x-www-form-urlencoded 방식

### HTML Form 데이터 전송 POST 전송 - 저장



내용을 추가한다.

Content-Type: application/x-www-form-urlencoded

그리고 폼에 입력한 전송할 항목을 HTTP Body에 문자로 username=kim&age=20 와 같이 & 로 구분해서 전송한다.

파일을 업로드 하려면 파일은 문자가 아니라 바이너리 데이터를 전송해야 한다. 문자를 전송하는 이 방식으로 파일을 전송하기는 어렵다. 그리고 또 한가지 문제가 더 있는데, 보통 폼을 전송할 때 파일만 전송하는 것이 아니라는 점이다.

다음 예를 보자.

- 이름
- 나이
- 첨부파일

여기에서 이름과 나이도 전송해야 하고, 첨부파일도 함께 전송해야 한다. 문제는 이름과 나이는 문자로 전송하고, 첨부파일은 바이너리로 전송해야 한다는 점이다. 여기에서 문제가 발생한다. **문자와 바이너리를 동시에 전송해야 하는 상황이다.**

이 문제를 해결하기 위해 HTTP는 multipart/form-data 라는 전송 방식을 제공한다.

### **multipart/form-data** 방식

## **HTML Form 데이터 전송**

### **multipart/form-data**

웹 브라우저가 생성한 요청 HTTP 메시지

username: kim  
age: 20  
file: 파일 선택 intro.png  
전송

```
<form action="/save" method="post" enctype="multipart/form-data">
<input type="text" name="username" />
<input type="text" name="age" />
<input type="file" name="file1" />
<button type="submit">전송</button>
</form>
```

POST /save HTTP/1.1  
Host: localhost:8080  
Content-Type: multipart/form-data; boundary=-----XXX  
Content-Length: 10457

-----XXX  
Content-Disposition: form-data; name="username"  
kim  
-----XXX  
Content-Disposition: form-data; name="age"  
20  
-----XXX  
Content-Disposition: form-data; name="file1"; filename="intro.png"  
Content-Type: image/png  
109238a9o0p3eqwokjasd09ou3oirjwoe9u34ouief...  
-----XXX--

끝에는 -- 추가

이 방식을 사용하려면 Form 태그에 별도의 enctype="multipart/form-data" 를 지정해야 한다.

`multipart/form-data` 방식은 다른 종류의 여러 파일과 폼의 내용 함께 전송할 수 있다. (그래서 이름이 `multipart`이다.)

폼의 입력 결과로 생성된 HTTP 메시지를 보면 각각의 전송 항목이 구분이 되어있다. `Content-Disposition`이라는 항목별 헤더가 추가되어 있고 여기에 부가 정보가 있다. 예제에서는 `username`, `age`, `file1`이 각각 분리되어 있고, 폼의 일반 데이터는 각 항목별로 문자가 전송되고, 파일의 경우 파일 이름과 Content-Type이 추가되고 바이너리 데이터가 전송된다.

`multipart/form-data`는 이렇게 각각의 항목을 구분해서, 한번에 전송하는 것이다.

## Part

`multipart/form-data`는 `application/x-www-form-urlencoded`와 비교해서 매우 복잡하고 각각의 부분(`Part`)로 나누어져 있다. 그렇다면 이렇게 복잡한 HTTP 메시지를 서버에서 어떻게 사용할 수 있을까?

## 참고

`multipart/form-data`와 폼 데이터 전송에 대한 더 자세한 내용은 모든 개발자를 위한 HTTP 웹 기본 지식 강의를 참고하자.

## 프로젝트 생성

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

<https://start.spring.io>

- 프로젝트 선택
  - Project: Gradle Project
  - Language: Java
  - Spring Boot: 2.4.x
- Project Metadata
  - Group: hello
  - Artifact: upload
  - Name: upload
  - Package name: **hello.upload**
  - Packaging: **Jar**
  - Java: 11

- Dependencies: **Spring Web, Lombok , Thymeleaf**

### **build.gradle**

```

plugins {
 id 'org.springframework.boot' version '2.4.5'
 id 'io.spring.dependency-management' version '1.0.11.RELEASE'
 id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
 compileOnly {
 extendsFrom annotationProcessor
 }
}

repositories {
 mavenCentral()
}

dependencies {
 implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
 implementation 'org.springframework.boot:spring-boot-starter-web'
 compileOnly 'org.projectlombok:lombok'
 annotationProcessor 'org.projectlombok:lombok'
 testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
 useJUnitPlatform()
}

```

- 동작 확인
  - 기본 메인 클래스 실행( `UploadApplication.main()` )
  - `http://localhost:8080` 호출해서 Whitelabel Error Page가 나오면 정상 동작

편의상 `index.html` 을 추가해두자.

`resources/static/index.html`

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="UTF-8">
 <title>Title</title>
</head>
<body>

 상품 관리

 서블릿 파일 업로드1
 서블릿 파일 업로드2
 스프링 파일 업로드
 상품 - 파일, 이미지 업로드

</body>
</html>
```

## 서블릿과 파일 업로드1

먼저 서블릿을 통한 파일 업로드를 코드와 함께 알아보자.

### ServletUploadControllerV1

```
package hello.upload.controller;
```

```
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.Part;
import java.io.IOException;
import java.util.Collection;

@Slf4j
@Controller
@RequestMapping("/servlet/v1")
public class ServletUploadControllerV1 {

 @GetMapping("/upload")
 public String newFile() {
 return "upload-form";
 }

 @PostMapping("/upload")
 public String saveFileV1(HttpServletRequest request) throws
ServletException, IOException {
 log.info("request={}", request);

 String itemName = request.getParameter("itemName");
 log.info("itemName={}", itemName);

 Collection<Part> parts = request.getParts();
 log.info("parts={}", parts);

 return "upload-form";
 }
}
```

`request.getParts()` : `multipart/form-data` 전송 방식에서 각각 나누어진 부분을 받아서 확인할 수 있다.

`resources/templates/upload-form.html`

```
<!DOCTYPE HTML>

<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
</head>
<body>

<div class="container">

 <div class="py-5 text-center">
 <h2>상품 등록 폼</h2>
 </div>

 <h4 class="mb-3">상품 입력</h4>

 <form th:action method="post" enctype="multipart/form-data">

 상품명 <input type="text" name="itemName">
 파일<input type="file" name="file" >

 <input type="submit"/>
 </form>

</div> <!-- /container -->
</body>
</html>
```

테스트를 진행하기 전에 먼저 다음 옵션들을 추가하자.

`application.properties`

```
logging.level.org.apache.coyote.http11=debug
```

이 옵션을 사용하면 HTTP 요청 메시지를 확인할 수 있다.

## 실행

<http://localhost:8080/servlet/v1/upload>

실행해보면 `logging.level.org.apache.coyote.http11` 옵션을 통한 로그에서 `multipart/form-data` 방식으로 전송된 것을 확인할 수 있다.

## 결과 로그

```
Content-Type: multipart/form-data; boundary=----xxxx

----xxx
Content-Disposition: form-data; name="itemName"

Spring
----xxx
Content-Disposition: form-data; name="file"; filename="test.data"
Content-Type: application/octet-stream

sdklajkljdf...
```

## 멀티파트 사용 옵션

### 업로드 사이즈 제한

```
spring.servlet.multipart.max-file-size=1MB
spring.servlet.multipart.max-request-size=10MB
```

큰 파일을 무제한 업로드하게 둘 수는 없으므로 업로드 사이즈를 제한할 수 있다.

사이즈를 넘으면 예외(`SizeLimitExceededException`)가 발생한다.

`max-file-size`: 파일 하나의 최대 사이즈, 기본 1MB

```
max-request-size : 멀티파트 요청 하나에 여러 파일을 업로드 할 수 있는데, 그 전체 합이다. 기본 10MB
```

### spring.servlet.multipart.enabled 끄기

```
spring.servlet.multipart.enabled=false
```

#### 결과 로그

```
request=org.apache.catalina.connector.RequestFacade@xxx
itemName=null
parts=[]
```

멀티파트는 일반적인 폼 요청인 `application/x-www-form-urlencoded` 보다 훨씬 복잡하다.

`spring.servlet.multipart.enabled` 옵션을 끄면 서블릿 컨테이너는 멀티파트와 관련된 처리를 하지 않는다.

그래서 결과 로그를 보면 `request.getParameter("itemName")`, `request.getParts()`의 결과가 비어있다.

### spring.servlet.multipart.enabled 켜기

```
spring.servlet.multipart.enabled=true (기본 true)
```

이 옵션을 켜면 스프링 부트는 서블릿 컨테이너에게 멀티파트 데이터를 처리하라고 설정한다. 참고로 기본 값은 `true` 이다.

```
request=org.springframework.web.multipart.support.StandardMultipartHttpServletRequest
itemName=Spring
parts=[ApplicationPart1, ApplicationPart2]
```

`request.getParameter("itemName")`의 결과도 잘 출력되고, `request.getParts()`에도 요청한 두 가지 멀티파트의 부분 데이터가 포함된 것을 확인할 수 있다. 이 옵션을 켜면 복잡한 멀티파트 요청을 처리해서 사용할 수 있게 제공한다.

로그를 보면 `HttpServletRequest` 객체가 `RequestFacade` → `StandardMultipartHttpServletRequest`로 변한 것을 확인할 수 있다.

## 참고

`spring.servlet.multipart.enabled` 옵션을 켜면 스프링의 `DispatcherServlet`에서 멀티파트 리졸버(`MultipartResolver`)를 실행한다.

멀티파트 리졸버는 멀티파트 요청인 경우 서블릿 컨테이너가 전달하는 일반적인 `HttpServletRequest`를 `MultipartHttpServletRequest`로 변환해서 반환한다.

`MultipartHttpServletRequest`는 `HttpServletRequest`의 자식 인터페이스이고, 멀티파트와 관련된 추가 기능을 제공한다.

스프링이 제공하는 기본 멀티파트 리졸버는 `MultipartHttpServletRequest` 인터페이스를 구현한 `StandardMultipartHttpServletRequest`를 반환한다.

이제 컨트롤러에서 `HttpServletRequest` 대신에 `MultipartHttpServletRequest`를 주입받을 수 있는데, 이것을 사용하면 멀티파트와 관련된 여러가지 처리를 편리하게 할 수 있다. 그런데 이후 강의에서 설명할 `MultipartFile`이라는 것을 사용하는 것이 더 편하기 때문에 `MultipartHttpServletRequest`를 잘 사용하지는 않는다. 더 자세한 내용은 `MultipartResolver`를 검색해보자.

## 서블릿과 파일 업로드2

서블릿이 제공하는 `Part`에 대해 알아보고 실제 파일도 서버에 업로드 해보자.

먼저 파일을 업로드를 하려면 실제 파일이 저장되는 경로가 필요하다.

해당 경로에 실제 폴더를 만들어두자.

그리고 다음에 만들어진 경로를 입력해두자.

### application.properties

```
file.dir=파일 업로드 경로 설정(예): /Users/kimyounghan/study/file/
```

## 주의

1. 꼭 해당 경로에 실제 폴더를 미리 만들어두자.
2. `application.properties`에서 설정할 때 마지막에 `/`(슬래시)가 포함된 것에 주의하자.

## ServletUploadControllerV2

```
package hello.upload.controller;
```

```
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Controller;
import org.springframework.util.StreamUtils;
import org.springframework.util.StringUtils;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.Part;
import java.io.IOException;
import java.io.InputStream;
import java.nio.charset.StandardCharsets;
import java.util.Collection;

@Slf4j
@Controller
@RequestMapping("/servlet/v2")
public class ServletUploadControllerV2 {

 @Value("${file.dir}")
 private String fileDir;

 @GetMapping("/upload")
 public String newFile() {
 return "upload-form";
 }

 @PostMapping("/upload")
 public String saveFileV1(HttpServletRequest request) throws
 ServletException, IOException {
 log.info("request={}", request);

 String itemName = request.getParameter("itemName");
 log.info("itemName={}, itemName", itemName);
```

```
Collection<Part> parts = request.getParts();
log.info("parts={}", parts);
for (Part part : parts) {

 log.info("==== PART ====");
 log.info("name={}, part.getName()");
 Collection<String> headerNames = part.getHeaderNames();
 for (String headerName : headerNames) {
 log.info("header {}: {}", headerName,
part.getHeader(headerName));
 }
 //편의 메서드

 //content-disposition; filename
 log.info("submittedFileName={}, part.getSubmittedFileName()");
 log.info("size={}, part.getSize()); //part body size

 //데이터 읽기
 InputStream inputStream = part.getInputStream();
 String body = StreamUtils.copyToString(inputStream,
StandardCharsets.UTF_8);
 log.info("body={}, body");

 //파일에 저장하기
 if (StringUtils.hasText(part.getSubmittedFileName())) {
 String fullPath = fileDir + part.getSubmittedFileName();
 log.info("파일 저장 fullPath={}, fullPath");
 part.write(fullPath);
 }
}

return "upload-form";
}

}
```

```
@Value("${file.dir}")
private String fileDir;
```

application.properties에서 설정한 file.dir의 값을 주입한다.

멀티파트 형식은 전송 데이터를 하나하나 각각 부분(Part)으로 나누어 전송한다. parts에는 이렇게 나누어진 데이터가 각각 담긴다.

서블릿이 제공하는 Part는 멀티파트 형식을 편리하게 읽을 수 있는 다양한 메서드를 제공한다.

### Part 주요 메서드

part.getSubmittedFileName(): 클라이언트가 전달한 파일명

part.getInputStream(): Part의 전송 데이터를 읽을 수 있다.

part.write(...): Part를 통해 전송된 데이터를 저장할 수 있다.

### 실행

<http://localhost:8080/servlet/v2/upload>

다음 내용을 전송했다.

- itemName : 상품A
- file : 스크린샷.png

### 결과 로그

```
===== PART =====
name=itemName
header content-disposition: form-data; name="itemName"
submittedFileName=null
size=7
body=상품A
===== PART =====
name=file
header content-disposition: form-data; name="file"; filename="스크린샷.png"
header content-type: image/png
submittedFileName=스크린샷.png
size=112384
body=qwlkjek2ljlese...
파일 저장 fullPath=/Users/kimyounghan/study/file/스크린샷.png
```

파일 저장 경로에 가보면 실제 파일이 저장된 것을 확인할 수 있다. 만약 저장이 되지 않았다면 파일 저장 경로를 다시 확인하자.

### 참고

큰 용량의 파일을 업로드를 테스트 할 때는 로그가 너무 많이 남아서 다음 옵션을 끄는 것이 좋다.

```
logging.level.org.apache.coyote.http11=debug
```

다음 부분도 파일의 바이너리 데이터를 모두 출력하므로 끄는 것이 좋다.

```
log.info("body={}", body);
```

서블릿이 제공하는 `Part` 는 편하기는 하지만, `HttpServletRequest` 를 사용해야 하고, 추가로 파일 부분만 구분하려면 여러가지 코드를 넣어야 한다. 이번에는 스프링이 이 부분을 얼마나 편리하게 제공하는지 확인해보자.

## 스프링과 파일 업로드

스프링은 `MultipartFile` 이라는 인터페이스로 멀티파트 파일을 매우 편리하게 지원한다.

### SpringUploadController

```
package hello.upload.controller;

import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.MultipartFile;

import javax.servlet.http.HttpServletRequest;
import java.io.File;
import java.io.IOException;
```

```
@Slf4j
```

```

@Controller
@RequestMapping("/spring")
public class SpringUploadController {

 @Value("${file.dir}")
 private String fileDir;

 @GetMapping("/upload")
 public String newFile() {
 return "upload-form";
 }

 @PostMapping("/upload")
 public String saveFile(@RequestParam String itemName,
 @RequestParam MultipartFile file, HttpServletRequest
request) throws IOException {
 log.info("request={}", request);
 log.info("itemName={}", itemName);
 log.info("multipartFile={}", file);

 if (!file.isEmpty()) {
 String fullPath = fileDir + file.getOriginalFilename();
 log.info("파일 저장 fullPath={}", fullPath);
 file.transferTo(new File(fullPath));
 }

 return "upload-form";
 }
}

```

코드를 보면 스프링 답게 딱 필요한 부분의 코드만 작성하면 된다.

`@RequestParam MultipartFile file`

업로드하는 HTML Form의 name에 맞추어 `@RequestParam`을 적용하면 된다. 추가로

`@ModelAttribute`에서도 `MultipartFile`을 동일하게 사용할 수 있다.

### MultipartFile 주요 메서드

```
file.getOriginalFilename() : 업로드 파일 명
```

```
file.transferTo(...) : 파일 저장
```

### 실행

<http://localhost:8080/spring/upload>

### 실행 로그

```
request=org.springframework.web.multipart.support.StandardMultipartHttpServletRequest@5c022dc6
itemName=상품A
multipartFile=org.springframework.web.multipart.support.StandardMultipartHttpServletRequest$StandardMultipartFile@274ba730
파일 저장 fullPath=/Users/kimyounghan/study/file/스크린샷.png
```

### 예제로 구현하는 파일 업로드, 다운로드

실제 파일이나 이미지를 업로드, 다운로드 할 때는 몇가지 고려할 점이 있는데, 구체적인 예제로 알아보자.

#### 요구사항

- 상품을 관리
  - 상품 이름
  - 첨부파일 하나
  - 이미지 파일 여러개
- 첨부파일을 업로드 다운로드 할 수 있다.
- 업로드한 이미지를 웹 브라우저에서 확인할 수 있다.

#### Item - 상품 도메인

```
package hello.upload.domain;
```

```
import lombok.Data;
```

```
import java.util.List;

@Data
public class Item {

 private Long id;
 private String itemName;
 private UploadFile attachFile;
 private List<UploadFile> imageFiles;
}
```

## ItemRepository - 상품 리포지토리

```
package hello.upload.domain;

import org.springframework.stereotype.Repository;

import java.util.HashMap;
import java.util.Map;

@Repository
public class ItemRepository {

 private final Map<Long, Item> store = new HashMap<>();
 private long sequence = 0L;

 public Item save(Item item) {
 item.setId(++sequence);
 store.put(item.getId(), item);
 return item;
 }

 public Item findById(Long id) {
 return store.get(id);
 }

}
```

## UploadFile - 업로드 파일 정보 보관

```
package hello.upload.domain;

import lombok.Data;

@Data
public class UploadFile {

 private String uploadFileName;
 private String storeFileName;

 public UploadFile(String uploadFileName, String storeFileName) {
 this.uploadFileName = uploadFileName;
 this.storeFileName = storeFileName;
 }
}
```

uploadFileName : 고객이 업로드한 파일명

storeFileName : 서버 내부에서 관리하는 파일명

고객이 업로드한 파일명으로 서버 내부에 파일을 저장하면 안된다. 왜냐하면 서로 다른 고객이 같은 파일이름을 업로드 하는 경우 기존 파일 이름과 충돌이 날 수 있다. 서버에서는 저장할 파일명이 겹치지 않도록 내부에서 관리하는 별도의 파일명이 필요하다.

## FileStore - 파일 저장과 관련된 업무 처리

```
package hello.upload.file;

import hello.upload.domain.UploadFile;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.multipart.MultipartFile;
```

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

@Component
public class FileStore {

 @Value("${file.dir}")
 private String fileDir;

 public String getFullPath(String filename) {
 return fileDir + filename;
 }

 public List<UploadFile> storeFiles(List<MultipartFile> multipartFiles)
 throws IOException {
 List<UploadFile> storeFileResult = new ArrayList<>();
 for (MultipartFile multipartFile : imageFiles) {
 if (!imageFile.isEmpty()) {
 storeFileResult.add(storeFile(multipartFile));
 }
 }
 return storeFileResult;
 }

 public UploadFile storeFile(MultipartFile multipartFile) throws IOException
 {
 if (multipartFile.isEmpty()) {
 return null;
 }

 String originalFilename = multipartFile.getOriginalFilename();
 String storeFileName = createStoreFileName(originalFilename);
 multipartFile.transferTo(new File(getFullPath(storeFileName)));
 return new UploadFile(originalFilename, storeFileName);
 }
}
```

```

private String createStoreFileName(String originalFilename) {
 String ext = extractExt(originalFilename);
 String uuid = UUID.randomUUID().toString();
 return uuid + "." + ext;
}

private String extractExt(String originalFilename) {
 int pos = originalFilename.lastIndexOf(".");
 return originalFilename.substring(pos + 1);
}

```

멀티파트 파일을 서버에 저장하는 역할을 담당한다.

- `createStoreFileName()` : 서버 내부에서 관리하는 파일명은 유일한 이름을 생성하는 `UUID`를 사용해서 충돌하지 않도록 한다.
- `extractExt()` : 확장자를 별도로 추출해서 서버 내부에서 관리하는 파일명에도 붙여준다. 예를 들어서 고객이 `a.png`라는 이름으로 업로드 하면 `51041c62-86e4-4274-801d-614a7d994edb.png`와 같이 저장한다.

## ItemForm

```

package hello.upload.controller;

import lombok.Data;
import org.springframework.web.multipart.MultipartFile;

import java.util.List;

@Data
public class ItemForm {
 private Long itemId;
 private String itemName;
 private List<MultipartFile> imageFiles;
 private MultipartFile attachFile;
}

```

상품 저장용 폼이다.

List<MultipartFile> imageFiles : 이미지를 다중 업로드 하기 위해 MultipartFile 를 사용했다.

MultipartFile attachFile : 멀티파트는 @ModelAttribute 에서 사용할 수 있다.

## ItemController

```
package hello.upload.controller;

import hello.upload.domain.UploadFile;
import hello.upload.domain.Item;
import hello.upload.domain.ItemRepository;
import hello.upload.file.FileStore;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.io.*;
import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import org.springframework.web.util.UriUtils;

import java.io.IOException;
import java.net.MalformedURLException;
import java.nio.charset.StandardCharsets;
import java.util.List;

@Slf4j
@Controller
@RequiredArgsConstructor
public class ItemController {

 private final ItemRepository itemRepository;
 private final FileStore fileStore;

 @GetMapping("/items/new")
 public String newItem(@ModelAttribute ItemForm form) {
 return "item-form";
 }
}
```

```

 @PostMapping("/items/new")
 public String saveItem(@ModelAttribute ItemForm form, RedirectAttributes
 redirectAttributes) throws IOException {

 UploadFile attachFile = fileStore.storeFile(form.getAttachFile());
 List<UploadFile> storeImageFiles =
 fileStore.storeFiles(form.getImageFiles());

 //데이터베이스에 저장
 Item item = new Item();
 item.setItemName(form.getItemName());
 item.setAttachFile(attachFile);
 item.setImageFiles(storeImageFiles);
 itemRepository.save(item);

 redirectAttributes.addAttribute("itemId", item.getId());

 return "redirect:/items/{itemId}";
 }

 @GetMapping("/items/{id}")
 public String items(@PathVariable Long id, Model model) {
 Item item = itemRepository.findById(id);
 model.addAttribute("item", item);
 return "item-view";
 }

 @ResponseBody
 @GetMapping("/images/{filename}")
 public Resource downloadImage(@PathVariable String filename) throws
 MalformedURLException {
 return new UrlResource("file:" + fileStore.getFullPath(filename));
 }

 @GetMapping("/attach/{itemId}")
 public ResponseEntity<Resource> downloadAttach(@PathVariable Long itemId)
 throws MalformedURLException {

```

```

 Item item = itemRepository.findById(itemId);
 String storeFileName = item.getAttachFile().getStoreFileName();
 String uploadFileName = item.getAttachFile().getUploadFileName();

 UrlResource resource = new UrlResource("file:" +
fileStore.getFullPath(storeFileName));

 log.info("uploadFileName={}", uploadFileName);
 String encodedUploadFileName = UriUtils.encode(uploadFileName,
StandardCharsets.UTF_8);
 String contentDisposition = "attachment; filename=\"{}\" +
encodedUploadFileName + "\"";
 return ResponseEntity.ok()
 .header(HttpHeaders.CONTENT_DISPOSITION, contentDisposition)
 .body(resource);
 }

}

```

- `@GetMapping("/items/new")` : 등록 폼을 보여준다.
- `@PostMapping("/items/new")` : 폼의 데이터를 저장하고 보여주는 화면으로 리다이렉트 한다.
- `@GetMapping("/items/{id}")` : 상품을 보여준다.
- `@GetMapping("/images/{filename}")` : `<img>` 태그로 이미지를 조회할 때 사용한다. `UrlResource`로 이미지 파일을 읽어서 `@ResponseBody`로 이미지 바이너리를 반환한다.
- `@GetMapping("/attach/{itemId}")` : 파일을 다운로드 할 때 실행한다. 예제를 더 단순화 할 수 있지만, 파일 다운로드 시 권한 체크같은 복잡한 상황까지 가정한다 생각하고 이미지 `id`를 요청하도록 했다. 파일 다운로드시에는 고객이 업로드한 파일 이름으로 다운로드 하는게 좋다. 이때는 `Content-Disposition` 헤더에 `attachment; filename="업로드 파일명"` 값을 주면 된다.

## 등록 폼 뷰

`resources/templates/item-form.html`

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
</head>
<body>

```

```

<div class="container">

 <div class="py-5 text-center">
 <h2>상품 등록</h2>
 </div>

 <form th:action method="post" enctype="multipart/form-data">

 상품명 <input type="text" name="itemName">
 첨부파일<input type="file" name="attachFile" >
 이미지 파일들<input type="file" multiple="multiple"
name="imageFiles" >

 <input type="submit"/>
 </form>

</div> <!-- /container -->
</body>
</html>

```

다중 파일 업로드를 하려면 `multiple="multiple"` 옵션을 주면 된다.

`ItemForm`의 다음 코드에서 여러 이미지 파일을 받을 수 있다.

```
private List<MultipartFile> imageFiles;
```

## 조회 뷰

`resources/templates/item-view.html`

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="utf-8">
</head>
<body>

<div class="container">

 <div class="py-5 text-center">

```

```
<h2>상품 조회</h2>
</div>

상품명: 상품명

첨부파일: <a th:if="${item.attachFile}" th:href="/attach/${item.id}|"
th:text="${item.getAttachFile().getUploadFileName()}" />

</div> <!-- /container -->
</body>
</html>
```

첨부 파일은 링크로 걸어두고, 이미지는 `<img>` 태그를 반복해서 출력한다.

## 실행

<http://localhost:8080/items/new>

실행해보면 하나의 첨부파일을 다운로드 업로드 하고, 여러 이미지 파일을 한번에 업로드 할 수 있다.

## 정리

# 12. 다음으로

#인강/5. 스프링 MVC 2/강의#

## 학습 내용 정리

### 전체 목차

- 1. 타임리프 - 기본 기능
- 2. 타임리프 - 스프링 통합과 품
- 3. 메시지, 국제화
- 4. 검증1 - Validation
- 5. 검증2 - Bean Validation
- 6. 로그인 처리1 - 쿠키, 세션
- 7. 로그인 처리2 - 필터, 인터셉터
- 8. 예외 처리와 오류 페이지
- 9. API 예외 처리
- 10. 스프링 탑입 컨버터
- 11. 파일 업로드

### 1. 타임리프 - 기본 기능

- 1. 타임리프 - 기본 기능 - 프로젝트 생성
- 1. 타임리프 - 기본 기능 - 타임리프 소개
- 1. 타임리프 - 기본 기능 - 텍스트 - text, utext
- 1. 타임리프 - 기본 기능 - 변수 - SpringEL
- 1. 타임리프 - 기본 기능 - 기본 객체들
- 1. 타임리프 - 기본 기능 - 유틸리티 객체와 날짜
- 1. 타임리프 - 기본 기능 - URL 링크
- 1. 타임리프 - 기본 기능 - 리터럴
- 1. 타임리프 - 기본 기능 - 연산
- 1. 타임리프 - 기본 기능 - 속성 값 설정
- 1. 타임리프 - 기본 기능 - 반복
- 1. 타임리프 - 기본 기능 - 조건부 평가
- 1. 타임리프 - 기본 기능 - 주석
- 1. 타임리프 - 기본 기능 - 블록
- 1. 타임리프 - 기본 기능 - 자바스크립트 인라인

- 1. 타임리프 - 기본 기능 - 템플릿 조작
- 1. 타임리프 - 기본 기능 - 템플릿 레이아웃1
- 1. 타임리프 - 기본 기능 - 템플릿 레이아웃2
- 1. 타임리프 - 기본 기능 - 정리

## 2. 타임리프 - 스프링 통합과 폼

- 2. 타임리프 - 스프링 통합과 폼 - 프로젝트 설정
- 2. 타임리프 - 스프링 통합과 폼 - 타임리프 스프링 통합
- 2. 타임리프 - 스프링 통합과 폼 - 입력 폼 처리
- 2. 타임리프 - 스프링 통합과 폼 - 요구사항 추가
- 2. 타임리프 - 스프링 통합과 폼 - 체크 박스 - 단일1
- 2. 타임리프 - 스프링 통합과 폼 - 체크 박스 - 단일2
- 2. 타임리프 - 스프링 통합과 폼 - 체크 박스 - 멀티
- 2. 타임리프 - 스프링 통합과 폼 - 라디오 버튼
- 2. 타임리프 - 스프링 통합과 폼 - 셀렉트 박스
- 2. 타임리프 - 스프링 통합과 폼 - 정리

## 3. 메시지, 국제화

- 3. 메시지, 국제화 - 프로젝트 설정
- 3. 메시지, 국제화 - 메시지, 국제화 소개
- 3. 메시지, 국제화 - 스프링 메시지 소스 설정
- 3. 메시지, 국제화 - 스프링 메시지 소스 사용
- 3. 메시지, 국제화 - 웹 애플리케이션에 메시지 적용하기
- 3. 메시지, 국제화 - 웹 애플리케이션에 국제화 적용하기
- 3. 메시지, 국제화 - 정리

## 4. 검증1 - Validation

- 4. 검증1 - Validation - 검증 요구사항
- 4. 검증1 - Validation - 프로젝트 설정 V1
- 4. 검증1 - Validation - 검증 직접 처리 - 소개
- 4. 검증1 - Validation - 검증 직접 처리 - 개발
- 4. 검증1 - Validation - 프로젝트 준비 V2
- 4. 검증1 - Validation - BindingResult1
- 4. 검증1 - Validation - BindingResult2
- 4. 검증1 - Validation - FieldError, ObjectError
- 4. 검증1 - Validation - 오류 코드와 메시지 처리1
- 4. 검증1 - Validation - 오류 코드와 메시지 처리2

- 4. 검증1 - Validation - 오류 코드와 메시지 처리3
- 4. 검증1 - Validation - 오류 코드와 메시지 처리4
- 4. 검증1 - Validation - 오류 코드와 메시지 처리5
- 4. 검증1 - Validation - 오류 코드와 메시지 처리6
- 4. 검증1 - Validation - Validator 분리1
- 4. 검증1 - Validation - Validator 분리2
- 4. 검증1 - Validation - 정리

## 5. 검증2 - Bean Validation

- 5. 검증2 - Bean Validation - Bean Validation - 소개
- 5. 검증2 - Bean Validation - Bean Validation - 시작
- 5. 검증2 - Bean Validation - Bean Validation - 프로젝트 준비 V3
- 5. 검증2 - Bean Validation - Bean Validation - 스프링 적용
- 5. 검증2 - Bean Validation - Bean Validation - 에러 코드
- 5. 검증2 - Bean Validation - Bean Validation - 오브젝트 오류
- 5. 검증2 - Bean Validation - Bean Validation - 수정에 적용
- 5. 검증2 - Bean Validation - Bean Validation - 한계
- 5. 검증2 - Bean Validation - Bean Validation - groups
- 5. 검증2 - Bean Validation - Form 전송 객체 분리 - 프로젝트 준비 V4
- 5. 검증2 - Bean Validation - Form 전송 객체 분리 - 소개
- 5. 검증2 - Bean Validation - Form 전송 객체 분리 - 개발
- 5. 검증2 - Bean Validation - Bean Validation - HTTP 메시지 컨버터
- 5. 검증2 - Bean Validation - 정리

## 6. 로그인 처리1 - 쿠키, 세션

- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 요구사항
- 6. 로그인 처리1 - 쿠키, 세션 - 프로젝트 생성
- 6. 로그인 처리1 - 쿠키, 세션 - 홈 화면
- 6. 로그인 처리1 - 쿠키, 세션 - 회원 가입
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 기능
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 쿠키 사용
- 6. 로그인 처리1 - 쿠키, 세션 - 쿠키와 보안 문제
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 세션 동작 방식
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 세션 직접 만들기
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 직접 만든 세션 적용
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 서블릿 HTTP 세션1
- 6. 로그인 처리1 - 쿠키, 세션 - 로그인 처리하기 - 서블릿 HTTP 세션2

- 6. 로그인 처리1 - 쿠키, 세션 - 세션 정보와 타임아웃 설정
- 6. 로그인 처리1 - 쿠키, 세션 - 정리

## 7. 로그인 처리2 - 필터, 인터셉터

- 7. 로그인 처리2 - 필터, 인터셉터 - 서블릿 필터 - 소개
- 7. 로그인 처리2 - 필터, 인터셉터 - 서블릿 필터 - 요청 로그
- 7. 로그인 처리2 - 필터, 인터셉터 - 서블릿 필터 - 인증 체크
- 7. 로그인 처리2 - 필터, 인터셉터 - 스프링 인터셉터 - 소개
- 7. 로그인 처리2 - 필터, 인터셉터 - 스프링 인터셉터 - 요청 로그
- 7. 로그인 처리2 - 필터, 인터셉터 - 스프링 인터셉터 - 인증 체크
- 7. 로그인 처리2 - 필터, 인터셉터 - ArgumentResolver 활용
- 7. 로그인 처리2 - 필터, 인터셉터 - 정리

## 8. 예외 처리와 오류 페이지

- 8. 예외 처리와 오류 페이지 - 프로젝트 생성
- 8. 예외 처리와 오류 페이지 - 서블릿 예외 처리 - 시작
- 8. 예외 처리와 오류 페이지 - 서블릿 예외 처리 - 오류 화면 제공
- 8. 예외 처리와 오류 페이지 - 서블릿 예외 처리 - 오류 페이지 작동 원리
- 8. 예외 처리와 오류 페이지 - 서블릿 예외 처리 - 필터
- 8. 예외 처리와 오류 페이지 - 서블릿 예외 처리 - 인터셉터
- 8. 예외 처리와 오류 페이지 - 스프링 부트 - 오류 페이지1
- 8. 예외 처리와 오류 페이지 - 스프링 부트 - 오류 페이지2
- 8. 예외 처리와 오류 페이지 - 정리

## 9. API 예외 처리

- 9. API 예외 처리 - API 예외 처리 - 시작
- 9. API 예외 처리 - API 예외 처리 - 스프링 부트 기본 오류 처리
- 9. API 예외 처리 - API 예외 처리 - HandlerExceptionResolver 시작
- 9. API 예외 처리 - API 예외 처리 - HandlerExceptionResolver 활용
- 9. API 예외 처리 - API 예외 처리 - 스프링이 제공하는 ExceptionResolver1
- 9. API 예외 처리 - API 예외 처리 - 스프링이 제공하는 ExceptionResolver2
- 9. API 예외 처리 - API 예외 처리 - @ExceptionHandler
- 9. API 예외 처리 - API 예외 처리 - @ControllerAdvice
- 9. API 예외 처리 - API 예외 처리 - 정리

## 10. 스프링 타입 컨버터

- 10. 스프링 타입 컨버터 - 프로젝트 생성
- 10. 스프링 타입 컨버터 - 스프링 타입 컨버터 소개
- 10. 스프링 타입 컨버터 - 타입 컨버터 - Converter
- 10. 스프링 타입 컨버터 - 컨버전 서비스 - ConversionService
- 10. 스프링 타입 컨버터 - 스프링에 Converter 적용하기
- 10. 스프링 타입 컨버터 - 뷰 템플릿에 컨버터 적용하기
- 10. 스프링 타입 컨버터 - 포맷터 - Formatter
- 10. 스프링 타입 컨버터 - 포맷터를 지원하는 컨버전 서비스
- 10. 스프링 타입 컨버터 - 포맷터 적용하기
- 10. 스프링 타입 컨버터 - 스프링이 제공하는 기본 포맷터
- 10. 스프링 타입 컨버터 - 정리

## 11. 파일 업로드

- 11. 파일 업로드 - 파일 업로드 소개
- 11. 파일 업로드 - 프로젝트 생성
- 11. 파일 업로드 - 서블릿과 파일 업로드1
- 11. 파일 업로드 - 서블릿과 파일 업로드2
- 11. 파일 업로드 - 스프링과 파일 업로드
- 11. 파일 업로드 - 예제로 구현하는 파일 업로드, 다운로드
- 11. 파일 업로드 - 정리

## 다음으로

### 스프링 데이터(DB) 접근 기술 안내

예제를 통해 단계적으로 기능을 발전시키며, 각 기능을 코드로 개발하면서 자연스럽게 학습

- 데이터베이스 커넥션
  - JDBC 커넥션
  - 커넥션 풀
- 스프링 트랜잭션
  - 트랜잭션 내부 원리
  - 자세한 사용법
  - 실무 주의 사항

- 다양한 데이터 접근 기술 이해
  - JDBC
  - JdbcTemplate
  - 마이바티스 - MyBatis
  - JPA
  - 스프링 데이터 JPA
  - QueryDSL
- 데이터베이스 예외 처리
  - 스프링 데이터베이스 예외 추상화
  - 올바른 예외 처리 사용법

## 로드맵 소개

### 스프링 완전 정복 시리즈(진행중)

스프링을 완전히 마스터 할 수 있는 로드맵

URL: <https://www.inflearn.com/roadmaps/373>

#### 강의 목록

- 스프링 입문 - 코드로 배우는 스프링 부트, 웹 MVC, DB 접근 기술
- 스프링 핵심 원리 - 기본편
- 모든 개발자를 위한 HTTP 웹 기본 지식
- 스프링 웹 MVC 1편
- 스프링 웹 MVC 2편
- 스프링 데이터(DB) 접근 기술 (예정)
- 스프링 부트 (예정)

실전 REST API 설계, 개발 강의 요청이 많아서, 고민중 → 스프링 시리즈를 일단 마무리 짓고 진행

### 스프링 부트와 JPA 실무 완전 정복 로드맵

최신 실무 기술로 웹 애플리케이션을 만들어보면서 학습하고 싶으면 [스프링 부트와 JPA 실무 완전 정복 로드맵](#)을 추천

URL: <https://www.inflearn.com/roadmaps/149>

#### 강의 목록

- 자바 ORM 표준 JPA 프로그래밍
- 실전! 스프링 부트와 JPA 활용1 - 웹 애플리케이션 개발
- 실전! 스프링 부트와 JPA 활용2 - API 개발과 성능 최적화
- 실전! 스프링 데이터 JPA
- 실전! Querydsl

Q: 전체 코스 순서는 어떻게 듣는게 좋나요?

목표: 최신 기술의 자바 백엔드 개발자

- **스프링 완전 정복 로드맵**
  - 스프링 입문 - 코드로 배우는 스프링 부트, 웹 MVC, DB 접근 기술
  - 스프링 핵심 원리 - 기본편
  - 모든 개발자를 위한 HTTP 웹 기본 지식
  - 스프링 웹 MVC 1편
  - 스프링 웹 MVC 2편
- **스프링 부트와 JPA 실무 완전 정복 로드맵**
- **학자형 코스 vs 야생형 코스**
  - 학자형 코스: 이론을 먼저 차근차근 쌓아올린 다음에 실무 활용으로 넘어가는 방식
  - 야생형 코스: 일단 실무에서 어떤 기술을 어떤식으로 활용하는지, 깊이가 부족해도 일단 경험해본 다음에 필요에 의해서 이론 기술들을 공부하는 방식

### 추천 학습 방법

스프링 입문과 스프링 핵심 원리를 듣고 나면 스프링으로 개발하는 가장 중요한 기본 지식을 쌓은 상태가 됩니다. 그래서 바로 **스프링 부트와 JPA 실무 완전 정복 로드맵**에 들어가는 것도 좋은 선택입니다.

스프링을 실무에서 어떤 식으로 활용해서 개발하는지 먼저 배워두고, 이후에 **스프링 완전정복 로드맵**를 통해 스프링 MVC나 스프링 데이터 접근 기술 같은 부분은 더 깊이있게 학습하시면 됩니다.

### 문제 해결 방안과 야생형

야생형 코스가 맞는 사람도 있고, 학자형 코스가 맞는 사람도 있음