# 5. 검증2 - Bean Validation

#2.인강/5. 스프링 MVC 2/강의#

## 목차



## 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/
- 검증 애노테이션 모음: 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` : `null` 을 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 ItemValidator를 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 `Validator` 로 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 애노테이션 마다 다르다.

**실행**

실행해보면 방금 등록한 메시지가 정상 적용되는 것을 확인할 수 있다.

**BeanValidation 메시지 찾는 순서**

1. 생성된 메시지 코드 순서대로 `messageSource`에서 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 억지로 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>
```

```
            <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 기존 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 변경**할 수 있다.
- 등록시에는 `id` 에 값이 없어도 되지만, **수정시에는 id 값이 필수**이다.

**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 폼 객체와 수정용 폼 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 로그인id, 주민번호 등등을 받을 수 있지만, 수정시에는 이런 부분이 빠진다. 그리고 검증 로직도 많이 달라진다. 그래서 `ItemUpdateForm` 이라는 별도의 객체로 데이터를 전달받는 것이 좋다.

`Item` 도메인 객체를 폼 전달 데이터로 사용하고, 그대로 쭉 넘기면 편리하겠지만, 앞에서 설명한 것과 같이 실무에서는 `Item` 의 데이터만 넘어오는 것이 아니라 무수한 추가 데이터가 넘어온다. 그리고 더 나아가서 `Item` 을 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 `groups` 를 적용할 일은 드물다.

**Q: 이름은 어떻게 지어야 하나요?**
이름은 의미있게 지으면 된다. `ItemSave` 라고 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 객체 처럼 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;
    }
}
```

**Postman을 사용해서 테스트 해보자.**

**성공 요청**

```
POST http://localhost:8080/validation/api/items/add
{"itemName":"hello", "price":1000, "quantity": 10}
```

**Postman에서 Body → raw → JSON을 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 `ItemSaveForm` 객체로 생성하는데 실패한다.
이 경우는 `ItemSaveForm` 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 `10000` 이면 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 파리미터를 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 `ItemSaveForm` 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 발생시 원하는 모양으로 예외를 처리하는 방법은 예외 처리 부분에서 다룬다.

## 정리