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.gradle` 에 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 폼

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

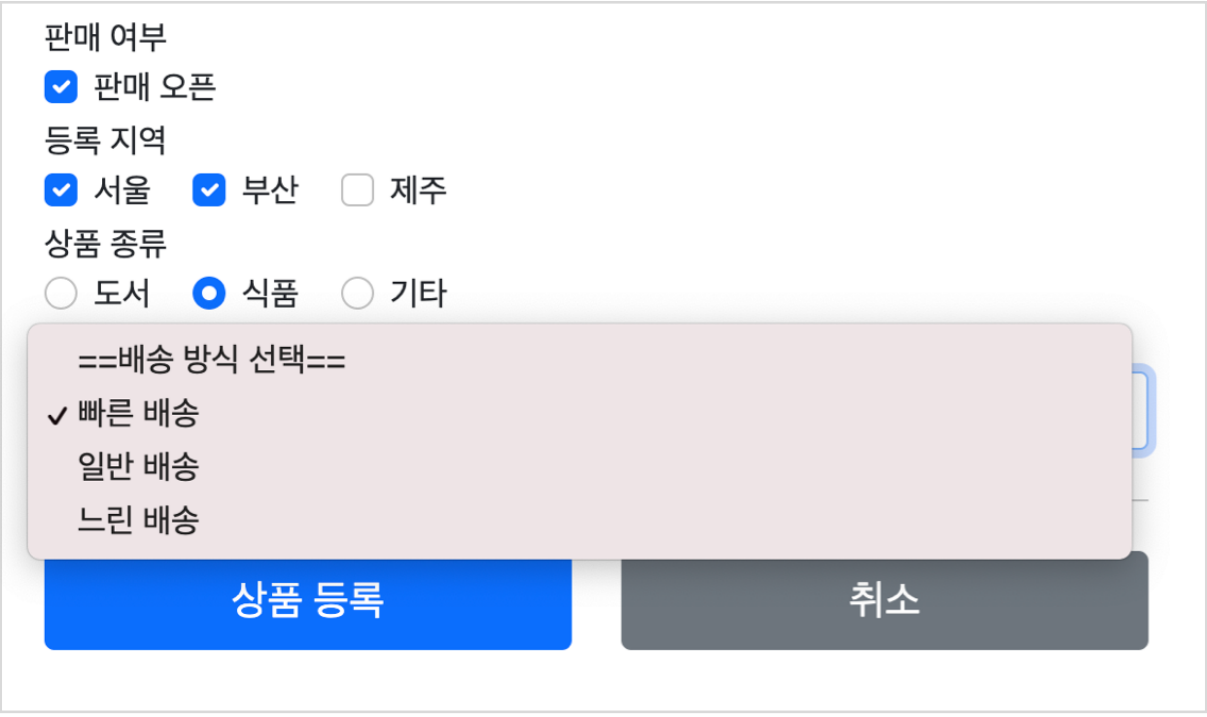
`th:object`, `th:field` 덕분에 폼을 개발할 때 약간의 편리함을 얻었다.
쉽고 단순해서 크게 어려움이 없었을 것이다.
사실 이것의 진짜 위력은 뒤에 설명할 검증(Validation)에서 나타난다. 이후 검증 부분에서 폼 처리와 관련된 부분을 더 깊이있게 알아보자.

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 폼의 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());
    ...
}
```

`FormItemController`에 `@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>
```

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 때 마다 이렇게 히든 필드를 추가하는 것은 상당히 번거롭다. 타임리프가 제공하는 폼 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.html`에는 `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&regions=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, 폼 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.html`에는 `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;
}
```

`DeliveryCode`라는 자바 객체를 사용하는 방법으로 진행하겠다.
`DeliveryCode`를 등록 폼, 조회, 수정 폼에서 모두 사용하므로 `@ModelAttribute`의 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 샐랙트 박스가 유지되는 것을 확인할 수 있다.

정리