Spring Boot 05 [웹 계층 개발]

실전! 스프링 부트와 JPA 활용 1

김영한

2022.08.13

웹 계층 개발

기본 홈화면 (컨트롤러 기초)

```
@Controller
@Slf4j
public class HomeController {

   @RequestMapping("/")  → 매핑되는 주소
   public String home(){
       log.info("home controller");
       return "home";    → resource/template + “home” + .html 로 prefix postfix적용
   }
}
```

home.html <head>

INCLUDE 스타일의 중복 코드 제거

```
<head th:replace="fragments/header :: header">
 <title>Hello</title>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
```

아래의 위치의 헤더로 대체된다. - fragments/header.html

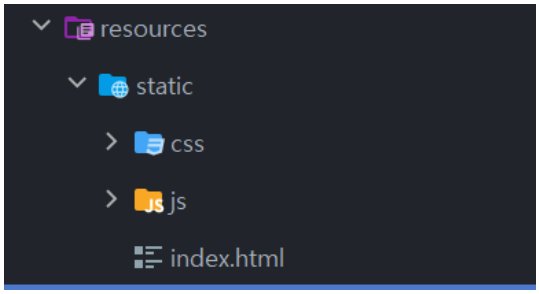
```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head th:fragment="header">
 <!-- Required meta tags -->
 <meta charset="utf-8">
 <meta name="viewport" content="width=device-width, initial-scale=1,
shrinkto-fit=no">
 <!-- Bootstrap CSS -->
 <link rel="stylesheet" href="/css/bootstrap.min.css" integrity="sha384-
```

```
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
        crossorigin="anonymous">
 <!-- Custom styles for this template -->
 <link href="/css/jumbotron-narrow.css" rel="stylesheet">
 <title>Hello, world!</title>
</head>
```

Hierarchical 스타일로 하면 중복마저 제거 가능하다.

Bootstrap 적용

Bootstrap 다운로드 후



resources 에

static(정적 폴더)에 적용하고

붙여넣어주면 css가 적용된다.

회원등록 웹

MemberController

등록시 → validation 가능

```
@PostMapping("/members/new")
public String create(@Valid MemberForm memberForm, BindingResult result, Address
address2){

   if(result.hasErrors()){        ← BindingResult 를 가지고 가져갈 수있다.
       return "members/createMemberForm";
   }
```

멤버 폼 에서 **Validation**에 관한 정보를 넣을 수 있다.

```
public class MemberForm {

   @NotEmpty(message = "회원 이름은 필수 입니다.") → @Valid어노테이션으로 가능
   private String name;
}
```

BindingResult - 웹 에러 확인

```
<form role="form" action="/members/new" th:object="${memberForm}"
     method="post">
   <div class="form-group">
       <label th:for="name">이름</label>
       <input type="text" th:field="*{name}" class="form-control"
              placeholder="이름을 입력하세요"
              th:class="${#fields.hasErrors('name')}? 'form-control fieldError' :
'form-control'">
       <p th:if="${#fields.hasErrors('name')}"
          th:errors="*{name}">Incorrect date</p>  →에러 정보를 가지고 html로 간다
   </div>
```


```
@NotEmpty(message = "회원 이름은 필수 입니다.")
```

에러가 있다면 어노테이션 메시지를 BindingResult로 가지고 간다.

https://www.thymeleaf.org/documentation 문서에 잘 정리되어있다.

?왜 Member 도메인 (엔티티)를 사용하지 않는가?

엔티티가 Validation 하기 지저분해지고 엔티티를 화면에서 바로 끌고 오는 것이 바람직하지 않다. 말그대로 필요한 Form 객체, DTO설계하고 사용하는것이 여러 부분에서 안전. 엔티티를 순수하게 유지하는게 중요하다! (핵심 비즈니스 로직만 가지고 있는 것이 좋다.)

회원목록 보기

```
<table class="table table-striped">
   <thead>
   <tr>
       <th>#</th>
       <th>이름</th>
       <th>도시</th>
       <th>주소</th>
       <th>우편번호</th>
   </tr>
   </thead>
```

```
    <tbody>
    <tr th:each="member : ${members}">  → 바로 바인딩이 된걸 사용가능
        <td th:text="${member.id}"></td>
        <td th:text="${member.name}"></td>
        <td th:text="${member.address?.city}"></td>   → null일때 걸러준다
        <td th:text="${member.address?.street}"></td>
        <td th:text="${member.address?.zipcode}"></td>
    </tr>
    </tbody>
</table>
```

→ 사실 이 부분에도 엔티티를 바로 사용하는 것보다 **DTO**를 사용하는 것이 좋다. 화면 탬플릿에 찍어보내는건 사용하는것은 괜찮지만 **API**를 설계할 때는 절때 외부로 반환하면 안된다. 노출 될 수도 있도 엔티티에 로직 추가하면 **API**가 변경되어버린다.

상품 등록 웹

```
@PostMapping("/items/new")
public String create(BookForm form){
    Book book = new Book();
    book.setName(form.getName());
    book.setPrice(form.getPrice());
    book.setStockQuantity(form.getStockQuantity());
    book.setAuthor(form.getAuthor());
    book.setIsbn(form.getIsbn());

    itemService.saveItem(book);

    return "redirect:/items";
}
```

→ 이 부분에도 **set**을 사용하는 것보다 생성자 메소드를 만드는 것이 좋다.

상품 수정

```
<td>
 <a href="#" th:href="@{/items/{id}/edit (id=${item.id})}"
    class="btn btn-primary" role="button">수정</a>
</td>
```

수정

이런식으로 표현된다.

```
@GetMapping("items/{itemId}/edit")
public String updateItemForm(@PathVariable("itemId") Long itemId, Model model){
   Book item = (Book) itemService.findOne(itemId);

   BookForm form = new BookForm();
   form.setId(item.getId());
   form.setName(item.getName());
   form.setPrice(item.getPrice());
   form.setStockQuantity(item.getStockQuantity());
   form.setAuthor(item.getAuthor());
   form.setIsbn(item.getIsbn());

   model.addAttribute("form", form);
   return "items/updateItemForm";
}
```

→ Book 엔티티가 아닌 Form 을 수정 페이지로 보내게 된다.
동적으로 `@GetMapping("items/{itemId}/edit")`에 itemId를 PathVariable를 받는다.

권한 체크 로직이 있어야 한다. itemId가 들어오기 때문에 위험.

변경 감지와 병합

준영속 엔티티

- 영속성 컨텍스트가 더이상 관리하지 않는 엔티티이다.

```
Book book = new Book();
book.setId(form.getId());
```

→ form 에 아이디가 있고 JPA 과 관리했던 Id가 있는 준영속 엔티티이다.

준영속 엔티티를 수정하는 2가지 방법 !

1. 영속 엔티티로 변경 감지 사용
2. 병합을 사용

영속 엔티티로 변경감지 사용

```
   @Transactional
   public void updateItem(Long itemId, String name, int price, int stockQuantity){
       Item findItem = itemRepository.findOne(itemId);
//        findItem.change(price, name, stockQuantity); 와 같이 의미있는 매서드를
```

```
만들어준다.
        findItem.setPrice(price);
        findItem.setName(name);
        findItem.setStockQuantity(stockQuantity);
    }
```

병합 사용 **(Merge)**

-준영속상태 → 영속상태로 변경해줌

Merge : 영속성 컨텍스트에서 같은 아이디로 영속 엔티티를 찾아내고 모든 데이터를 변경시킨다**.**

→ 모두 변경시키기 때문에 누락된것은 **null**로 처리된다**.** 즉**,** 실무에서 사용하기 힘들다**.**