타임리프 - 기본기능

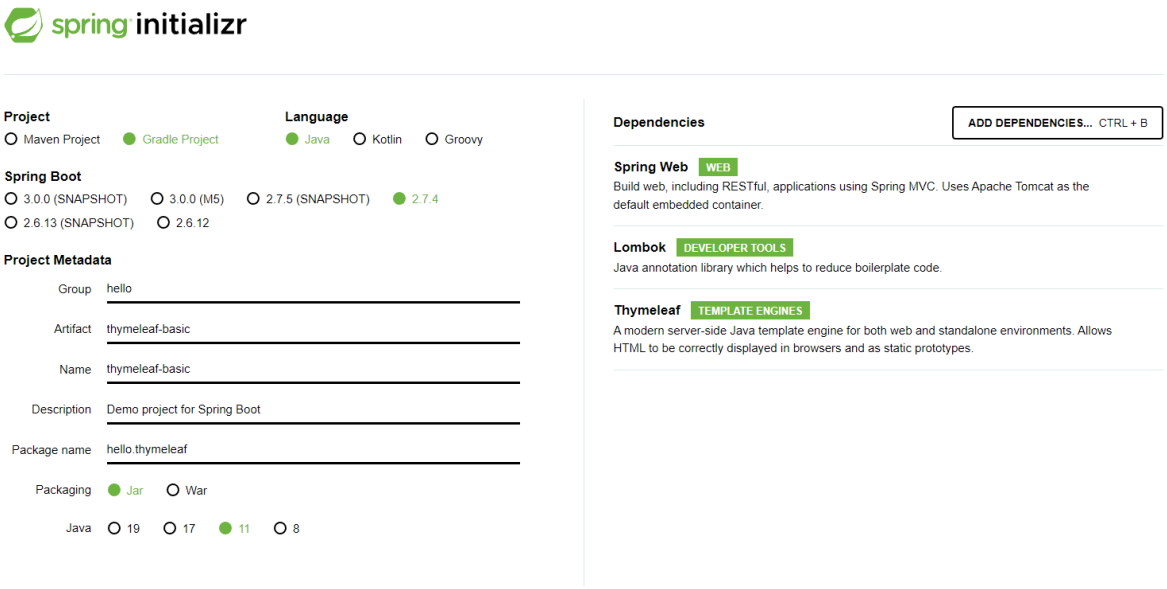
프로젝트 생성

사전 준비물

- Java 11
  - 컴퓨터 설정이 Java 8이기에 강의와 달리 버전 8로 실습하고자 한다.
- IDE : intelliJ

**스프링 부트 스타터 사이드로 이동해서 스프링 프로젝트 생성

https://start.spring.io/



Spring Boot가 자동으로 home으로 인식하는 path : /resources/static/index.html

Lombok 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

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

타임리프는 무엇인가?

타임리프는 스프링 부트에서 공식적으로 지원하는 웹 및 독립 실행형 환경 모두를 위한 최신 서버 측 Java 템플릿 엔진이다. JSP와 달리 Thymeleaf 문서는 html 확장자를 갖고 있어 JSP처럼 Servlet이 문서를 표현하는 방식이 아니기 때문에 서버 없이도 동작 가능하다.

컨트롤러가 전달하는 데이터를 이용해 동적으로 화면을 만들어주는 역할을 하는 뷰 템플릿 엔진이다.

타임리프 특징

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

1. 서버 사이드 HTML 렌더링 (SSR)

: 타임리프는 백엔드 서버에서 HTML을 동적으로 렌더링 하는 용도로 사용된다.(JSP 같은 거)

** 클라이언트 서버 렌더링은 JavaScript 같은 거

2. 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.

3. 스프링 통합 지원

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

텍스트 - text, utext

- HTML의 콘텐츠(content)에 데이터를 출력할 때는 다음과 같이 th:text 를 사용하면 된다.
  - `<span th:text="${data}">`
- HTML 테그의 속성이 아니라 HTML 콘텐츠 영역안에서 직접 데이터를 출력하고 싶으면 다음과 같이 [[...]] 를 사용하면 된다.
  - 컨텐츠 안에서 직접 출력하기 = `[[${data}]]`

HTML 엔티티

HTML 엔티티 : 웹 브라우저는 < 를 HTML 태그의 시작으로 인식한다. 따라서 < 를 태그의 시작이 아니라 문자로 표현하는 방법

이스케이프(escape) : HTML에서 사용하는 특수 문자를 HTML 엔티티로 변경하는 것

이스케이프 기능을 사용하지 않으려면 어떻게 해야 할까? (Unescape)
타임리프는 다음 두 기능을 제공한다.

- th:text ⇒ th:utext
- [[...]] ⇒ [(...)]

```
<!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>
```

이스케이프 처리를 왜 해야할까 ?

게시판을 예로 들면 이해하기 쉽다. 사용자가 <,>를 게시물로 작성하였을 때 이스케이프 처리가 안되어있다면 HTML이 다 깨질 것이다.

변수 - SpringEL

변수 표현식 : ${…}

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 호

지역변수 선언

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

scope를 벗어나면 사용할 수 없다. 벗어나서 사용할 경우 오류가 발생하여 서버가 돌아가지 않는다.

`th:with="first=${users[0]}"` : first에 users[0] 객체가 들어간다.

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

기본 객체들

타임리프는 기본 객체들을 제공한다.

```
# 기본 객체

${#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.

```
# 편의 객체

HTTP 요청 파라미터 접근: param
예) ${param.paramData}

HTTP 세션 접근: session
예) ${session.sessionData}

스프링 빈 접근: @
예) ${@helloBean.hello('Spring!')}
```

유틸리티 객체와 날짜

> 참고
> 이런 유틸리티 객체들은 대략 이런 것이 있다 알아두고, 필요할 때 찾아서 사용하면 된다

타임리프 유틸리티 객체
https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#expression-utilityobjects

유틸리티 객체 예시
https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#appendix-b-expressionutility-objects

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 유틸리티 객체 : #temporals

URL 링크

단순한 URL
: @{/hello} ⇒ /hello

쿼리 파라미터
: @{/hello(param1=${param1}, param2=${param2})} ⇒ /hello?param1=data1&param2=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

: 리터럴 - 소스 코드 상에서 고정된 값을 말하는 용어

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.

이어지는 문자 종류 : A-Z , a-z , 0-9 , [] , . , - , _

흔히 하는 실수 : 띄어쓰기는 포함되지 않는다.
`<span th:text="hello">`

연산

- 비교연산 : HTML 엔티티를 사용해야 하는 부분을 주의하자!

> >(gt), < (lt), >= (ge), <= (le), ! (not), == (eq), != (neq, ne)

- Elvis 연산자(`?:`)
  - 조건식의 편의 버전
  - 데이터가 없으면 오른쪽을 출력한다.
- No-Operation: _ 인 경우 마치 타임리프가 실행되지 않는 것 처럼 동작한다. 이것을 잘 사용하면 HTML의 내용 그대로 활용할 수 있다. 마지막 예를 보면 데이터가 없습니다. 부분이 그대로 출력된다.

속성 값 설정

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

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

HTML에서 checked 속성은 checked 속성의 값과 상관없이 checked라는 속성만 있어도 체크를 나타낼 수 있다. 하지만 true, false값을 고정되게 사용해야 하는 부분에서 개발자 입장에서는 불편하다.

th:checked를 사용하면 변수를 받아 가변적으로 checked속성을 나타낼 수 있다.

```
<input type="checkbox" name="active" th:checked="${isChecked}"/>
```

타임리프의 th:checked 는 값이 false 인 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:each 를 사용한다. 추가로 반복에서 사용할 수 있는 여러 상태 값을 지원한다.

반복 상태 유지

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

반복의 두번째 파라미터를 설정해서 반복의 상태를 확인 할 수 있다.

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

여기서는 user + Stat = userStat 이므로 생략 가능하다.

조건부 평가

if, unless

- if , unless (if 의 반대)
- 조건이 만족하지 않으면 관련 태그가 날라간다.(태그 자체를 렌더링하지 않는다.)

switch

- 만족하는 조건이 없을 때 사용하는 디폴트이다.

주석

1. HTML 주석

   : 타임리프가 렌더링 하지 않고, 그대로 남겨둔다.

2. 타임리프 파서 주석 (주로 사용함!!)

   : 렌더링에서 주석이 아예 안 나온다.

3. 타임리프 프로토타입 주석

   : 타임리프로 렌더링 했을 때만 보여진다.
   : 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.

사용하기 애매한 경우에 사용하면 되지만 안 쓰는 게 제일 좋음

<th:block>은 렌더링 시에 제거된다.

자바스크립트 인라인

서버에서 실행하는 것이 아니라 웹 브라우저에서 실행한다.

자바스크립트도 타임리프를 쓸 경우 내추럴 템플릿을 쓸 수 있다.

**실습 결과를 확인하고자 한다면 우클릭 후 '페이지 소스 보기'를 누르면 된다.

인라인 사용 전 문제

1. 들어갈 데이터가 문자일 경우
   - 오류가 발생한다.

2. 객체일 경우
    - toString으로 되어 toString 결과값이 적혀있다.

3. 네추럴 템플릿을 사용하기 어려움
    - 주석처리한 부분이 보여서 문제 생김

인라인 사용 후

1. 문자열

```
var username = [[${user.username}]]; // 오류뜸

위 코드로 실행 시 오류가 발생하기에 ""를 넣어줘야 한다.

var username = "[[${user.username}]]"; // 오류안뜸
```

2. 객체

타임리프의 자바스크립트 인라인 기능을 사용하면 객체를 JSON으로 자동으로 변환해준다.
var user = [[${user}]];

인라인 사용 전

- var user = BasicController.User(username=userA, age=10);
- 객체의 toString()이 호출된 값

인라인 사용 후

- var user = {"username":"userA","age":10};
- 객체를 JSON으로 변환

자바스크립트 인라인 each

```
<!-- 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 때 공통 영역이 많다. 공통인 영역을 한 파일로 만들어 놓고 불러다가 쓰기 위해 타임리프에서는 템플릿 조각과 레이아웃 기능을 지원한다.

th:fragment : fragment이름 지정

▼ template/fragment/footer.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
  <footer th:fragment="copy">
     푸터 자리 입니다.
  </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>
```

부분 포함 insert

: insert는 해당 태그 아래에 템플릿 조각을 넣는다.

: th:insert="~{경로 :: fragment이름}"

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

- 아래에 `template/fragment/footer` 조각이 들어간다.

```
<div>
  <footer>
     푸터 자리 입니다.
  </footer>
</div>
```

부분 포함 replace

: replace는 해당 태그를 템플릿 조각으로 대체한다.

: th:replace="~{경로 :: fragment이름}"

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

- 는 사라지고 대신에 `template/fragment/footer` 조각이 들어간다.

```
<footer>
    푸터 자리 입니다.
</footer>
```

단순 표현식

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

파라미터 사용

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

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

출력 결과

```
<footer>
    <p>Footer 입니다.</p>
    <p>파라미터 자리 입니다.</p>
    <p>데이터1</p>
    <p>데이터2</p>
</footer>
```

템플릿 레이아웃1

위에 살펴본 템플릿 조각과 다른 것은 title, links에 데이터가 아닌 태그 자체가 들어간다.

```
<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>
```

common_header(~{::title},~{::link}) 이 부분이 핵심이다.

- ::title 은 현재 페이지의 title 태그들을 전달한다.
- ::link 는 현재 페이지의 link 태그들을 전달한다.

만약, 같은 태그가 여러 개 있을 경우 태그를 여러 번 실행한다.

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>
    <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>
```

```
<!DOCTYPE html>
<html>
<head>
    <title>메인 타이틀</title><title>서브 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>
```

템플릿 레이아웃2

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

장점 : 레이아웃 파일 1개만 바꿔도 적용된 많은 페이지들이 같이 바뀐다.

단점 : 체계적으로 잘 관리해야한다.

```
<html th:replace="~{template/layoutExtend/layoutFile :: layout(~{::title}, ~{::section})}" xmlns:th="http://www.thymeleaf.org">
```

- title과 section은 태그 이름이다.
    - section이 여러 개이면 section 여러 개를 순차적으로 돌아서 실행한다.

** 공통인 부분을 개발하는 두 가지 방법

- fragment로 공통인 조각을 넣어 관리
    - 페이지가 적을 때 사용하는 것이 좋음
- layout으로 공통적인 부분을 관리
    - 페이지가 많을 때 사용하는 것이 좋음