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 - 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 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">유틸리티 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>
```

```
    <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

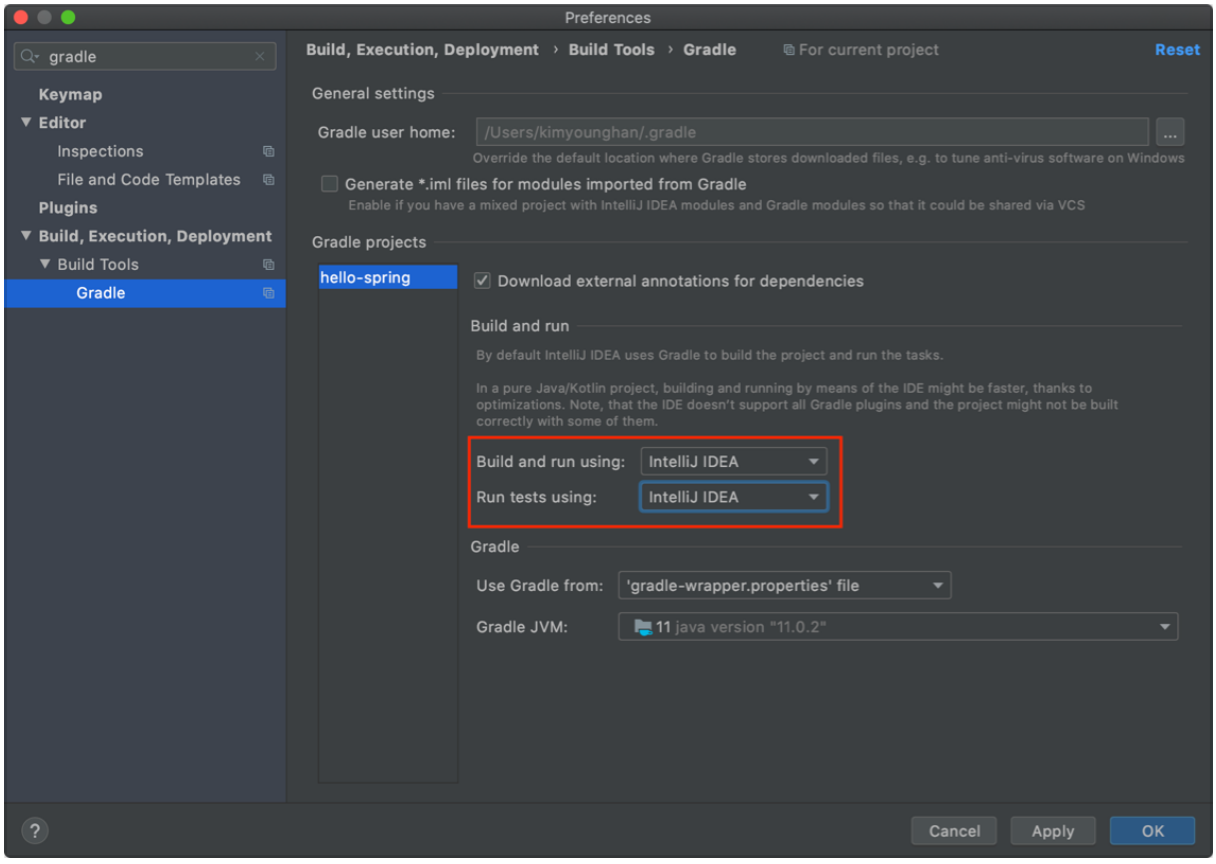
IntelliJ Gradle 대신에 자바 직접 실행

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

> **윈도우 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

윈도우 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 폼 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.

```
컨텐츠 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;
```

```java
@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

```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>"
```

`<b>` 테그를 사용해서 **Spring!**이라는 단어가 진하게 나오도록 해보자.

웹 브라우저에서 실행결과를 보자.

- 웹 브라우저: `Hello <b>Spring!</b>`
- 소스보기: `Hello <b>Spring!</b>`

개발자가 의도한 것은 `<b>`가 있으면 해당 부분을 강조하는 것이 목적이었다. 그런데 `<b>` 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 제공**한다.

- `<` ➡ `<`
- `>` ➡ `>`
- 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-unescaped

실행해보면 다음과 같이 정상 수행되는 것을 확인할 수 있다.

- 웹 브라우저: Hello **Spring!**
- 소스보기: `Hello <b>Spring!</b>`

> **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

유틸리티 객체 예시

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

> **참고**
> 이런 유틸리티 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 유틸리티 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>
</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&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

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 것 처럼 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>
    <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 것 처럼 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" />` ➜ 이 경우에도 checked 속성이 있기 때문에 checked 처리가 되어버린다.

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

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.

BasicController 추가

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

```java
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

```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 `<span>...<span>` 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>

</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 `userA` 라는 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 each를 지원하는데, 다음과 같이 사용한다.

`/resources/templates/basic/javascript.html` 에 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 비효율 적이다. 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 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>
```

`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', '데이터
2')}"></div>
</body>
</html>
```

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

`footer.html` 의 `copy` 부분

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

부분 포함 insert

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

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

`th:insert` 를 사용하면 현재 태그(`div`) 내부에 추가한다.

부분 포함 replace

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

```
<h2>부분 포함 replace</h2>
<footer>
푸터 자리 입니다.
</footer>
```

`th:replace` 를 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 자리 입니다.
</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 자리 입니다.</p>
<p>데이터1</p>
<p>데이터2</p>
</footer>
```

`footer.html`의 `copyParam` 부분

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