Thymeleaf

목차

타임리프 개요

- 타임리프란 무엇인가?
- 왜 타임리프를 쓰는가 (JSP랑 비교)
- 타임리프의 핵심 철학: Natural Template
- HTML5 완벽 지원의 의미
- 스프링부트와의 궁합

기본 문법 구조

- `th:text`, `th:utext` 차이 (텍스트 출력 vs HTML 출력)
- `th:value`, `th:href`, `th:src` 등 속성 설정
- `th:each` 반복문 (forEach)
- `th:if`, `th:unless`, `th:switch`, `th:case` 조건문
- `th:object`, `*{...}` 바인딩 객체 다루기
- `th:with` (지역 변수 선언)

타임리프 표현식 종류

- `${...}`: 변수 표현식
- `*{...}`: 선택 표현식 (form 객체 바인딩)
- `#{...}`: 메시지 표현식 (국제화)
- `@{...}`: URL 링크 표현식
- `~{...}`: fragment 표현식 (템플릿 조각 포함)

HTML 속성 조작

- `th:attr` vs `th:attrappend`, `th:attrprepend`
- `th:class`, `th:classappend`
- `th:style`, `th:styleappend`
- 동적으로 checked, selected, disabled 설정하기

반복 처리 (th:each)

- 리스트 출력
- 인덱스 값 활용 (`stat` 속성)
- 중첩 반복 처리

- 조건 반복 (필터링)

조건 처리 (th:if / th:switch)

- 단일 조건
- 복수 조건
- switch-case 스타일 처리
- 부정 조건 처리 (`th:unless`)

URL 처리

- 절대 경로와 상대 경로
- `@{...}` 링크 표현식
- 쿼리 파라미터 붙이기
- 리디렉션 링크 처리

Form 처리

- `<form>` 태그와 `th:object`
- `<input>`, `<textarea>`, `<select>`에 `th:field` 사용
- 자동 바인딩, 검증 메시지 출력
- 폼 제출과 Spring Controller 연동

템플릿 레이아웃 조각화

- `th:insert`, `th:replace`, `th:include` 차이점
- 공통 레이아웃 구성 (header, footer, sidebar 등)
- 템플릿 조각에서 파라미터 넘기기 (`th:with`)
- 레이아웃 템플릿 라이브러리 (e.g., Layout Dialect)

메시지 처리 (국제화)

- `messages.properties`, `messages_ko.properties`
- `#{message.key}` 사용
- 기본 메시지, 파라미터 메시지
- 폼 오류 메시지랑 연동

예외 상황 처리

- 널(null) 값 처리 방법
- Null-safe 연산 (`${user?.name}` 등)
- 예외 발생 시 화면 대응 전략

Fragment (조각 템플릿)

- 선언: `th:fragment`
- 사용: `th:insert`, `th:replace`, `th:include`
- 파라미터 있는 조각
- 조각 중첩 포함 구조

타임리프에서 자바스크립트 사용

- `th:inline="javascript"`로 JS 안에 변수 쓰기
- JS에서 서버 데이터 바인딩하는 법
- JSON 형태 데이터 삽입

타임리프와 스프링 연동 (심화)

- Spring MVC + 타임리프 구조
- Model 객체 전달
- DTO, Entity, Form 객체 구분해서 전달하기
- RedirectAttributes 사용

사용자 정의 유틸리티

- `#dates`, `#numbers`, `#strings` 등 내장 객체
- 커스텀 유틸리티 클래스 만들기
- Dialect 확장 (고급)

보안 처리

- HTML 이스케이프 (`th:text`)
- `th:utext`를 쓸 때 주의할 점
- CSRF 토큰 삽입 (`<input type="hidden">`)
- 인증된 사용자 정보 출력 (Spring Security 연동)

스프링 시큐리티 + 타임리프

- `sec:authorize`, `sec:authentication`
- 권한별 메뉴 렌더링
- 로그인 사용자 정보 가져오기
- 로그아웃 링크 처리

유닛 테스트 및 디버깅

- 타임리프 단독 테스트 방법
- 테스트에서 `Model` 넘겨서 렌더링 확인
- 로그로 템플릿 디버깅하기

타임리프 확장 기능들

- Layout Dialect
- Extras (Spring Security, Java 8 Time, Data)
- Standard Dialect 이해
- Dialect 직접 만들기

실전 프로젝트 적용 팁

- 폼 바인딩 실수 방지법
- 템플릿 캐시 처리
- 정적 리소스 (JS, CSS) 관리
- 에러 페이지 커스터마이징

실전 프로젝트 예시: "회원 게시판 시스템"

- 기능 요약
  - 회원 가입/로그인/로그아웃
  - 게시글 작성/조회/수정/삭제
  - 댓글 기능
  - 관리자 전용 기능
  - 페이징, 검색, 유효성 검증
  - 타임리프: fragment, th:each, th:if, messages, 유틸, 보안, layout 전부 사용

전체 페이지 구성 및 라우팅

- `/` – 메인 페이지
- `/user/signup` – 회원 가입
- `/user/login` – 로그인
- `/user/mypage` – 마이페이지
- `/post/list` – 게시글 목록 (페이징, 검색 포함)
- `/post/{id}` – 게시글 상세
- `/post/create` – 게시글 작성
- `/post/edit/{id}` – 게시글 수정
- `/admin/user/list` – 관리자: 전체 회원 목록

타임리프 템플릿 구성

- templates/layout
  - `base.html` – 전체 레이아웃
    - `th:fragment="layout(content)"`
    - header, footer, main 포함
    - `layout:decorate="~{layout/base}"`
- templates/fragments
  - `header.html` – 로그인 상태 분기 (`sec:authorize`)
  - `footer.html` – 고정 영역
  - `alert.html` – flash 메시지 출력 fragment
  - `paging.html` – 페이징 처리 fragment
  - `sidebar.html` – 관리자 전용 메뉴
  - `msgBox(message, type)` – 공통 메시지 컴포넌트 (fragment with parameter)

타임리프 문법 적용 포인트

- `th:object`, `th:field` → 모든 form 페이지 (`signup`, `create`, `edit`)
- `th:each` → 목록 페이지 (게시글, 회원, 댓글 반복)
- `th:if`, `th:unless` → 로그인 분기, 데이터 유무 분기
- `th:switch`, `th:case` → 사용자 역할에 따라 버튼 구분
- `th:with` → 지역 변수 선언 (예: `th:with="roleName=${#myUtil.role(user.role)}"`)
- `th:replace`, `th:insert` → header, footer, alert 조각 삽입
- `th:inline="javascript"` → 로그인한 사용자 정보 JS에 전달
- `@{}` → URL 처리 (`th:href`, `th:src`, `th:action`)
- `#{}` → 국제화 메시지 처리
- `${#fields.hasErrors('field')}` → 유효성 검사 메시지 처리

폼 처리 구성

- 회원가입, 게시글 작성, 댓글 입력
- `th:field`로 name/id/value 자동 설정
- `BindingResult`로 에러 메시지 출력
- `th:errors`, `#fields` 활용

보안 처리 (thymeleaf-extras-springsecurity6)

- `sec:authorize="isAuthenticated()"`, `hasRole('ADMIN')` 등으로 메뉴 제어
- 로그인 사용자 정보 출력: `sec:authentication="name"`
- 관리자 메뉴, 사용자 메뉴 분리

유틸리티 적용

- `@Component("myUtil")` 등록
  - `role(String code)` → "관리자", "일반회원"
  - `formatDate(LocalDateTime)` → "yyyy-MM-dd HH:mm"
  - `limitLength(String, int)` → 글자 수 자르기
- `${#myUtil.xxx(...)}` 형태로 호출

메시지 처리

- resources/messages.properties
- 모든 텍스트 `#{key}`로 처리
- 검증 메시지까지 완전 외부화

페이지네이션 처리

- `th:each="page : ${pageList}"`
- `th:href="@{/post/list(page=${i})}"`
- 현재 페이지 `th:classappend="${i == nowPage} ? 'active' : ''"`

JavaScript 연동 (서버 → JS 데이터 전달)

- JSON 객체 전달

에러 처리 & 디버깅

- 예외 페이지: `/templates/error/404.html`, `500.html`, `403.html`
- `th:if="${errorMessage}" th:text="${errorMessage}"`로 커스텀 메시지 처리
- 디버깅 시 `th:text="'[DEBUG] ID: ' + ${user.id}"` 사용

테스트 전략

- `@WebMvcTest`로 컨트롤러 + 뷰 템플릿 테스트
- `MockMvc` 사용해 HTML 렌더링 결과 비교
- `containsString(...)`, `.model().attributeExists(...)` 등 사용

타임리프 개요

타임리프란 무엇인가?

타임리프(Thymeleaf)의 정의

타임리프는 **자바 기반의 서버 사이드 HTML 템플릿 엔진**이다. Spring Framework, 특히 Spring Boot에서 웹 페이지를 만들 때 가장 자주 사용되는 템플릿 엔진 중 하나이다. HTML을 기준으로 작성되기 때문에 **브라우저에서도 바로 열어볼 수 있는 자연스러운 템플릿(Natural Template)** 이라는 개념이 핵심이다.

타임리프의 주요 특징

특징	설명
Natural Templates	HTML을 그대로 브라우저에서 볼 수 있고, 서버에서 동적으로 처리할 수도 있음. 디자인-개발 협업이 쉬워짐
Spring 완전 통합	스프링 MVC의 Model 데이터를 그대로 쓸 수 있고, 스프링 시큐리티도 연동 가능
HTML5 완전 호환	HTML5 문법을 그대로 쓰면서 타임리프의 속성(th:*)을 같이 사용 가능
표준 표현식 제공	`${...}`, `*{...}`, `@{...}`, `#{...}` 등 다양한 표현식 지원
템플릿 조각 재사용	`th:insert`, `th:replace` 등으로 공통 템플릿 재사용 가능
조건문/반복문 내장	자바 없이 템플릿에서 직접 로직 제어 가능
I18N 메시지 지원	`messages.properties` 파일과 연결해 다국어 처리 가능
다양한 Dialect 확장	기본 문법 외에 커스텀 Dialect로 기능 확장 가능

타임리프의 동작 방식

1. **클라이언트가 요청을 보냄**
    - 브라우저가 `/hello` 같은 URL 요청을 보냄
2. **스프링 컨트롤러가 Model을 생성**
    - 예: `model.addAttribute("name", "정석");`
3. **타임리프가 템플릿을 HTML로 렌더링**
    - `templates/hello.html` 파일을 가져와서 `${name}` 자리에 "정석"을 삽입
4. **완성된 HTML을 브라우저에 응답으로 전송**

왜 타임리프를 쓰는가? (JSP나 다른 템플릿 엔진과 비교)

항목	JSP	Thymeleaf
파일 형식	`.jsp`	`.html`
브라우저에서 확인	❌ (JSP는 톰캣 필요)	✅ (HTML로 바로 확인 가능)
자연 템플릿	❌	✅
표현 언어	JSTL, EL	타임리프 표현식
학습 난이도	중간	쉬움
스프링 부트 연동성	좋음	매우 좋음
HTML5 지원	제한적	완전 지원
프론트 개발자와 협업	불편	매우 편함

실제 예시 코드 비교 (JSP vs 타임리프)

JSP

```
<p>${user.name}님 환영합니다</p>
```

타임리프

```
<p th:text="${user.name} + '님 환영합니다'"></p>
```

타임리프는 HTML처럼 생겼기 때문에 디자이너나 프론트 개발자도 바로 보고 이해할 수 있다.

타임리프의 핵심 철학: Natural Template

- HTML 파일이 **서버 없이도 브라우저에서 깨지지 않게 열리는 구조**를 말한다.
- `th:text="..."` 같은 속성은 HTML 렌더링에 영향 없음
- 개발자는 서버에서는 동적으로 처리하고, 디자이너는 정적인 화면을 그대로 확인 가능

예:

```
<!-- 자연 템플릿 -->
<p th:text="${username}">기본 사용자 이름</p>
```

위 코드는 브라우저에서는 "기본 사용자 이름"으로 뜨지만, 서버에선 `${username}`으로 치환된다.

스프링부트에서 타임리프 기본 설정

```
// build.gradle
implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
```

기본적으로 `src/main/resources/templates` 아래 `.html` 파일을 찾고,
응답은 아래처럼 컨트롤러에서 반환:

```
@GetMapping("/hello")
public String hello(Model model) {
    model.addAttribute("username", "홍길동");
    return "hello"; // templates/hello.html
}
```

실무에서 주로 사용하는 이유 요약

- 디자인-개발 협업이 쉽고,
- 에러 메시지/폼 바인딩/보안까지 다 처리되고,
- 구조화된 대규모 프로젝트에서 레이아웃 조각화 기능까지 강력하게 제공하기 때문이다.

기본 문법 구조

타임리프에서 값을 화면에 출력하려면 가장 먼저 배우는 게 바로 `th:text`랑 `th:utext`이다. 이 둘은 아주 자주 쓰이는 기본 문법이고, **XSS 방어**, **HTML 표현** 같은 중요한 기능 차이가 있기 때문에 완전히 구분해서 써야 한다.

`th:text` – 기본 텍스트 출력

HTML 태그 안에 값을 **문자열로 변환해서 출력**하고, 기존 태그 내용을 **대체**

```
<p th:text="${user.name}">기본 이름</p>
```

위 코드는 `${user.name}`의 값이 예를 들어 `"홍길동"`이라면, 다음과 같이 변환된다.

```
<p>홍길동</p>
```

- `${...}` 는 Model에서 넘긴 데이터를 뜻해
- `기본 이름`은 백업값인데, 서버가 없을 땐 이게 보임
- HTML 태그 안에 값이 그대로 들어가지 않고, **태그의 내용이 바뀜**

HTML 이스케이프 기능 (XSS 방어)

`th:text`는 HTML 태그를 그대로 출력하지 않고, **문자열로 변환 + 특수 문자 이스케이프**를 해준다.

예시

```
<p th:text="${message}">hello</p>
```

Model에 있는 message가 아래처럼 들어있다고 하면,

```
model.addAttribute("message", "<b>해킹</b>");
```

실제 출력 결과:

```
<p>&lt;b&gt;해킹&lt;/b&gt;</p>
```

=> 즉, HTML 태그를 문자로 처리해서 **해킹(XSS)을 방어**함.

`th:utext` – HTML 태그 그대로 출력

`th:utext`는 `<b>굵게</b>`, `<i>기울임</i>` 같은 HTML 태그를 **실제로 렌더링**하고 싶을 때 쓴다.

```
<p th:utext="${htmlText}">백업 텍스트</p>
```

Model에 아래처럼 값이 있다면:

```
model.addAttribute("htmlText", "<b>중요</b>");
```

출력 결과는:

```
<p><b>중요</b></p>
```

`th:text`, `th:utext` 차이 (텍스트 출력 vs HTML 출력)

항목	`th:text`	`th:utext`
태그 출력	❌ (`<b>` → `&lt;b&gt;`)	✅ (`<b>` → 실제 `<b>`) 렌더링
XSS 방어	✅ 자동 이스케이프 처리	❌ 개발자가 직접 방어해야 함
사용 목적	일반 텍스트 출력	HTML 태그 포함 콘텐츠 출력
기본 권장	✅ 기본적으로 `th:text` 권장	❌ HTML 조작이 필요할 때만 사용

보안 주의점

- `th:utext`는 **스크립트 삽입 공격에 취약할 수 있어**, 외부 입력을 직접 넣을 땐 쓰면 안 돼.
- 예: `<script>alert('XSS')</script>` 가 실제로 실행될 수 있음.

실제 사용 예시(JSP와 비교)

JSP

```
<p><c:out value="${content}" /></p>
```

타임리프

```
<p th:text="${content}"></p>
```

→ 구조도 간단하고, HTML 그대로 유지된다는 점에서 타임리프가 더 보기 쉽다.

예제 전체 코드

```
@GetMapping("/intro")
public String intro(Model model) {
    model.addAttribute("username", "홍길동");
    model.addAttribute("rawHtml", "<b>VIP</b>");
    return "intro";
}
```

```
<!-- intro.html -->
<h1 th:text="${username}">기본 사용자</h1>
<p th:text="${rawHtml}">기본 HTML</p> <!-- &lt;b&gt;VIP&lt;/b&gt; -->
<p th:utext="${rawHtml}">기본 HTML</p> <!-- <b>VIP</b> -->
```

결론

- 일반적인 텍스트는 `th:text`**로 출력**하자 (보안 자동 처리됨)
- HTML 태그까지 표현하고 싶을 때만 `th:utext` **사용**, 그땐 **보안 필수** 고려

`th:value`, `th:href`, `th:src` 등 속성 설정

기본 개념

타임리프에서는 HTML 태그의 기존 속성을 동적으로 설정 가능. 정적인 값 대신 서버에서 전달한 데이터를 속성에 넣는다.

예시

```
<input type="text" th:value="${user.name}" />
<a th:href="@{/user/view(id=${user.id})}">상세보기</a>
<img th:src="@{/images/${user.profileImg}}" />
```

`th:value` – input, textarea 등 입력값 설정

설명

`<input>`이나 `<textarea>` 등의 **value 속성**을 서버에서 전달한 값으로 설정할 때 사용해.

예시:

```
<input type="text" th:value="${user.name}" />
```

Model에 `user.name = "홍길동"`이라면:

```
<input type="text" value="정석" />
```

- 폼에서 값 바인딩할 때 거의 무조건 씀
- `th:field`의 내부에도 `th:value`가 쓰임

`th:href` – 링크 동적 설정

설명

`<a>` 태그에서 링크를 동적으로 넣고 싶을 때 사용.

예시:

```
<a th:href="@{/user/view(id=${user.id})}">프로필 보기</a>
```

user.id가 12라면 실제 출력은:

```
<a href="/user/view?id=12">프로필 보기</a>
```

- `@{...}`은 타임리프의 URL 표현식이야 (라우팅 자동 지원)
- 쿼리스트링도 쉽게 붙일 수 있음

`th:src` – 이미지, 스크립트, 스타일 경로 설정

설명:

`<img>`, `<script>`, `<link>` 태그 등에서 파일 경로를 동적으로 설정할 때 사용해.

이미지 예시:

```
<img th:src="@{/images/${user.profileImg}}" alt="프로필" />
```

Model에 `user.profileImg = "profile123.jpg"`일 때:

```
<img src="/images/profile123.jpg" alt="프로필" />
```

JS나 CSS 예시:

```
<script th:src="@{/js/app.js}"></script>
<link rel="stylesheet" th:href="@{/css/style.css}" />
```

- 정적 리소스는 `src/main/resources/static` 또는 `public`에 두면 `/` 경로로 접근 가능
- 경로 자동 인코딩됨

그 외 유용한 속성들

`th:alt` – 이미지 대체 텍스트

```
<img th:src="@{/img/${photo}}" th:alt="${photoAlt}" />
```

`th:placeholder` – 입력 힌트 텍스트

```
<input type="text" th:placeholder="${placeholderText}" />
```

`th:title` – 마우스 올리면 나오는 설명

```
<span th:title="'가입일: ' + ${user.joinDate}">정석</span>
```

속성 복합 설정 – `th:attr`

여러 속성을 한 번에 설정하고 싶다면 `th:attr`을 사용

```
<input th:attr="value=${user.name}, placeholder='이름 입력'" />
```

속성 이어붙이기 – `th:attrappend`, `th:classappend`

기존 속성 뒤에 값을 붙이고 싶을 땐 이렇게 써:

```
<div class="box" th:classappend="${active} ? ' active' : ''"></div>
```

조건부 속성 렌더링

```
<input type="checkbox" th:checked="${user.agree}" />
<option th:selected="${user.gender == 'male'}">남자</option>
<input th:disabled="${form.locked}" />
```

정리 요약표

속성	역할 설명	예시
`th:value`	input/textarea의 값 설정	`<input th:value="${user.name}" />`
`th:href`	a 태그 링크 설정	`<a th:href="@{/user(id=${id})}" />`
`th:src`	이미지, 스크립트 경로 설정	`<img th:src="@{/img/${file}}" />`
`th:alt`	이미지 대체 텍스트 설정	`<img th:alt="${text}" />`
`th:placeholder`	입력칸 힌트 설정	`<input th:placeholder="${hint}" />`
`th:attr`	여러 속성을 한 번에 설정	`<input th:attr="value=${v}, alt='a'" />`
`th:checked` / `th:selected` / `th:disabled`	조건부 렌더링	`<input th:checked="${yes}" />`

`th:each` 반복문 (forEach)

`th:each`란?

- `th:each`는 자바의 `for-each`처럼 리스트나 배열을 반복해서 출력할 수 있게 해주는 속성이야.
- HTML 태그에 붙여서, 해당 태그를 리스트 길이만큼 반복해서 출력해줘.

기본 문법

```
<tr th:each="user : ${userList}">
    <td th:text="${user.name}">이름</td>
    <td th:text="${user.age}">나이</td>
</tr>
```

- `user : ${userList}` → 자바의 `for (User user : userList)` 같은 구조
- `userList`는 Model에 담긴 List 객체
- `user`는 각각의 요소

실제 컨트롤러 코드 예시

```
@GetMapping("/users")
public String userList(Model model) {
    List<User> users = List.of(
        new User("홍길동", 32),
        new User("유관순", 28),
        new User("이순신", 25)
    );
    model.addAttribute("userList", users);
    return "user_list";
}
```

출력 결과

```
<tr>
  <td>홍길동</td>
  <td>32</td>
</tr>
<tr>
  <td>유관순</td>
  <td>28</td>
</tr>
<tr>
  <td>이순신</td>
  <td>25</td>
</tr>
```

`th:each` 문법의 형태들

형태	설명
`item : ${list}`	일반적인 반복
`item, stat : ${list}`	상태 객체 포함 반복
`item : ${#lists.sort(list)}`	정렬된 리스트 반복

반복 상태 추적 - 상태 변수 `stat`

```
<tr th:each="user, stat : ${userList}">
    <td th:text="${stat.index}">인덱스</td>
    <td th:text="${user.name}">이름</td>
</tr>
```

속성명	설명
`stat.index`	0부터 시작하는 인덱스
`stat.count`	1부터 시작하는 개수
`stat.size`	전체 리스트 길이
`stat.first`	첫 번째 요소면 true
`stat.last`	마지막 요소면 true
`stat.even` / `stat.odd`	짝수/홀수 판별

홀짝 줄 스타일 주기 예시

```
<tr th:each="user, stat : ${userList}" th:class="${stat.odd} ? 'odd-row' : 'even-row'">
    <td th:text="${user.name}">이름</td>
</tr>
```

- 홀수 줄이면 `class="odd-row"`, 짝수면 `even-row`

중첩 반복 처리

```
<div th:each="group : ${groups}">
    <h3 th:text="${group.name}">그룹 이름</h3>
    <ul>
        <li th:each="member : ${group.members}" th:text="${member}">이름</li>
    </ul>
</div>
```

리스트가 비었을 때 처리

```
<tr th:if="${#lists.isEmpty(userList)}">
    <td colspan="2">데이터가 없습니다</td>
</tr>
<tr th:each="user : ${userList}">
    <td th:text="${user.name}">이름</td>
</tr>
```

순번 출력 (1부터)

```
<span th:each="item, stat : ${items}" th:text="${stat.count} + '. ' + ${item}">1. 아이템
</span>
```

리스트 아닌 것도 반복 가능

- 배열
- `java.util.Map` → key, value 반복
- `Set`

```
<tr th:each="entry : ${map}">
  <td th:text="${entry.key}">Key</td>
  <td th:text="${entry.value}">Value</td>
</tr>
```

예외/주의사항

- `th:each` 안에서는 `th:text`나 다른 속성에도 `${item.xxx}`처럼 item을 잘 연결해줘야 돼
- HTML 태그 구조가 꼬이면 반복 결과가 의도와 다르게 나올 수 있음

요약

기능	문법 예시
기본 반복	`th:each="item : ${list}"`
상태 추적	`item, stat : ${list}`
인덱스 출력	`${stat.index}` or `${stat.count}`
중첩 반복	`th:each="inner : ${outer.innerList}"`
비었을 때 처리	`th:if="${#lists.isEmpty(list)}"`
스타일 동적 부여	`th:class="${stat.odd} ? 'odd' : 'even'"`

`th:if`, `th:unless`, `th:switch`, `th:case` 조건문

`th:if` – 조건 만족 시 렌더링

기본 구조

```
<p th:if="${user.age > 18}">성인입니다</p>
```

- 조건이 `true`면 렌더링됨
- `false`면 HTML에 **아예 안 보임 (출력 자체 안 됨)**

예시

```
<span th:if="${user.login}">로그인 중</span>
```

user.login이 true일 때만 보여진다.

`th:unless` – 조건이 거짓일 때 렌더링

- `if`의 반대 역할

  ```
  <span th:unless="${user.login}">로그인 해주세요</span>
  ```

- login이 false일 때만 출력

`th:if` + `th:unless` 조합 예시

```
<span th:if="${user.admin}">관리자입니다</span>
<span th:unless="${user.admin}">일반 사용자입니다</span>
```

`th:switch` + `th:case` – 여러 조건 분기 (switch-case)

기본 구조

```
<div th:switch="${user.grade}">
    <p th:case="'VIP'">VIP 고객</p>
    <p th:case="'GOLD'">골드 고객</p>
    <p th:case="*">일반 고객</p>
</div>
```

- `${user.grade}` 값이 `VIP`이면 첫 번째 `<p>`만 출력
- `*`는 default 역할 (어떤 case에도 해당 안 되면 출력)

실제 예시

```
model.addAttribute("user", new User("홍길동", "GOLD"));
```

```
<div th:switch="${user.grade}">
    <span th:case="'VIP'">VIP 혜택 제공</span>
    <span th:case="'GOLD'">골드 등급 혜택</span>
    <span th:case="*">일반 혜택</span>
</div>
```

→ 출력: “골드 등급 혜택”

조건 안에서 논리 연산도 가능

```
<span th:if="${user.age > 18 and user.age < 65}">성인입니다</span>
<span th:if="${user.gender == 'M' or user.gender == 'F'}">성별 선택됨</span>
```

- `and`, `or`, `not` 전부 가능
- `==`, `!=`, `>`, `<`, `>=`, `<=` 가능

조건문 중첩 가능

```
<div th:if="${user != null}">
    <span th:if="${user.age > 60}">노년층</span>
    <span th:if="${user.age <= 60}">일반</span>
</div>
```

HTML 안에 조건 삽입하는 방식

조건문은 태그 전체에 적용돼서, 해당 조건이 false면 **태그 자체가 생기지 않는다.**

예:

```
<p th:if="${false}">이 문장은 출력되지 않음</p>
```

→ HTML 코드상에도 `<p>` 태그가 아예 없음

조건문 안에서 변수 선언 (`th:with` 같이 사용)

```
<div th:if="${user != null}" th:with="isAdult=${user.age > 18}">
  <span th:if="${isAdult}">성인입니다</span>
</div>
```

정리표

속성	설명	예시
`th:if`	조건이 true일 때 렌더링	`<p th:if="${age > 19}">성인</p>`
`th:unless`	조건이 false일 때 렌더링	`<p th:unless="${age > 19}">미성년</p>`
`th:switch`	switch 대상 설정	`th:switch="${type}"`

속성	설명	예시
`th:case`	case 조건별 분기	`th:case="'A'"`, `th:case="*"`

조건 렌더링과 속성 동시 처리 예시

```
<a th:href="@{/login}" th:if="${!login}">로그인</a>
<a th:href="@{/logout}" th:if="${login}">로그아웃</a>
```

`th:object`, `*{...}` 바인딩 객체 다루기

바인딩 객체의 목적

HTML `<form>` 태그 안에서 서버에 넘길 객체랑 필드를 자동으로 **양방향 바인딩**해주는 구조

- `th:object` : 폼 전체에서 사용할 **커맨드 객체(폼 객체)**를 지정
- `*{...}` : `th:object`에 지정된 객체의 **필드 값을 참조**할 때 사용

> 주의 : `${...}` 대신 `*{...}` 쓴다.

기본 구조

```
<form th:action="@{/register}" th:object="${user}" method="post">
    <input type="text" th:field="*{name}" />
    <input type="text" th:field="*{email}" />
    <button type="submit">등록</button>
</form>
```

```
@GetMapping("/register")
public String form(Model model) {
    model.addAttribute("user", new User());
    return "register";
}
```

`th:object`란?

- 폼 전체의 **기준 객체**를 지정
- 이후에 `*{}` 표현식은 여기서 지정된 객체의 속성을 참조
- Spring의 `@ModelAttribute`, `@PostMapping`과 자동 연결됨

```
<form th:object="${user}">
```

`th:field`와 `*{}`는 함께 사용

```
<input type="text" th:field="*{name}" />
```

- 내부적으로 name="name", value="${user.name}" 자동 생성됨
- 이게 있어야 스프링이 `@ModelAttribute`를 통해 데이터 바인딩할 수 있음

실제 HTML 결과 예시

```
<input type="text" name="name" value="정석" id="name">
```

> 타임리프는 `id`, `name`, `value` 전부 자동으로 설정해줘

폼 객체 예시 DTO

```
@Data
public class User {
    private String name;
    private String email;
}
```

`th:field`와 `th:value` 차이점

항목	`th:field="*{name}"`	`th:value="${user.name}"`
name 속성	자동 설정됨	수동 지정 필요
스프링 바인딩	자동 바인딩 됨	바인딩 X
오류 처리	자동 반영 (BindingResult)	불가능
사용 위치	form 전용	어디든 사용 가능

타임리프 폼 전체 구조 예시

```
<form th:action="@{/user/save}" th:object="${user}" method="post">
    <label for="name">이름</label>
    <input type="text" th:field="*{name}" />

    <label for="email">이메일</label>
    <input type="email" th:field="*{email}" />

    <button type="submit">저장</button>
</form>
```

유효성 검증과 함께 쓰기 (`BindingResult`)

```
@PostMapping("/user/save")
public String save(@Valid @ModelAttribute("user") User user,
                   BindingResult result) {
    if (result.hasErrors()) {
        return "register";
    }
    // 저장 로직
    return "redirect:/user/list";
}
```

에러 메시지 출력하기

```
<div th:if="${#fields.hasErrors('name')}" th:errors="*{name}">이름 오류</div>
```

중첩 객체 바인딩도 가능

```
<form th:object="${user}">
    <input th:field="*{address.zipcode}" />
</form>
```

- user.address.zipcode을 자동으로 바인딩

요약 정리표

항목	설명
`th:object`	폼 기준이 되는 객체 지정
`*{name}`	`th:object`의 name 속성 참조
`th:field`	입력 필드 자동 처리 (name, value, id 등)
`th:value`	값만 출력 (폼 바인딩과는 별개)
`th:errors`	해당 필드의 검증 오류 메시지 표시
`th:if="${#fields.hasErrors('name')}"`	필드에 오류가 있는지 조건 분기

`th:with` (지역 변수 선언)

`th:with`란?

- 템플릿 안에서 **지역 변수(임시 변수)를 선언**할 수 있는 기능이야.
- 반복문, 조건문, fragment, layout, 심지어 일반 div에서도 사용 가능
- 값 계산을 여러 번 안 해도 되니까 코드가 **깔끔해지고 효율도 올라감**

기본 문법

```
<div th:with="adult=${user.age > 18}">
  <span th:if="${adult}">성인입니다</span>
</div>
```

- `adult`라는 이름으로 **지역 변수 선언**
- 이후 `adult`는 `${adult}`로 참조 가능

여러 변수 동시에 선언 가능

```
<div th:with="price=${item.price}, vat=${item.price * 0.1}, total=${price + vat}">
  <p th:text="'총액: ' + ${total} + '원'"></p>
</div>
```

- 변수끼리도 의존 가능 (위 예제에서 `total`은 `price` + `vat`)

반복문 안에서 `th:with` 쓰기

```
<tr th:each="user : ${userList}" th:with="isAdult=${user.age >= 20}">
  <td th:text="${user.name}">이름</td>
  <td th:text="${isAdult} ? '성인' : '미성년'">나이 구분</td>
</tr>
```

- `user.age >= 20` 결과를 `isAdult` 변수로 저장
- if 조건에 쓰거나, 출력에 재활용 가능

조건문 안에서 변수 선언

```
<div th:if="${user != null}" th:with="welcomeMsg='안녕하세요 ' + ${user.name} + '님'">
  <p th:text="${welcomeMsg}"></p>
</div>
```

fragment 안에서 쓰기

```
<div th:replace="fragments/info :: userInfo(${user.name}, ${user.age})"
     th:with="ageGroup=${user.age >= 60 ? '노년' : '청년'}">
</div>
```

- 조각 템플릿으로 넘기기 전 가공해서 보내는 것도 가능

자주 쓰이는 상황 예시

상황	예시
나이 계산	`th:with="isAdult=${user.age >= 20}"`

상황	예시
총합 계산	`th:with="total=${item.price + item.tax}"`
필터링 변수	`th:with="enabledUsers=${#lists.select(users, u -> u.active)}"`
날짜 포맷 저장	`th:with="fmtDate=${#dates.format(user.birth, 'yyyy-MM-dd')}"`

중첩 선언도 가능 (단, 보기 어렵게는 하지 말 것)

```
<div th:with="price=${item.price}, vat=${price * 0.1}, total=${price + vat}">
  <p th:text="'총액: ' + ${total}"></p>
</div>
```

유의사항

- `th:with`는 **현재 태그 내부에서만 유효**
- 스코프를 벗어나면 변수는 사라짐
- 단순 계산, 조건, 텍스트 조합 등에 쓰면 최고

요약표

항목	설명
선언 방식	`th:with="변수명=값"`
사용 위치	`div`, `if`, `each`, `fragment` 등 어디든 가능
여러 개 선언	`,`로 구분
변수 참조	`${변수명}`
효과	코드 가독성 증가, 반복 계산 줄임

타임리프 표현식 종류

`\${}`, `*{}`, `@{}`, `#{}`, `~{}`

전체 표현식 한눈에 보기

표현식	이름	사용 용도
`${...}`	변수 표현식	Model 데이터 접근, 일반 변수 출력
`*{...}`	선택 표현식	`th:object` 바인딩 시 내부 필드 접근
`#{...}`	메시지 표현식	다국어 처리, messages.properties 참조
`@{...}`	링크(URL) 표현식	동적 링크, 경로 생성

표현식	이름	사용 용도
`~{...}`	fragment 표현식	템플릿 조각 포함할 때 사용

`${...}`: 변수 표현식

- 가장 기본 표현식
- Model에 담긴 데이터를 가져올 때 사용

```
<span th:text="${user.name}">이름</span>
```

사용 예시:

```
model.addAttribute("user", new User("홍길동", 32));
```

```
<p th:text="${user.age}">32</p>
```

> 자바의 `user.getAge()`와 동일

`*{...}`: 선택 표현식 (form 객체 바인딩)

- `th:object`로 지정한 객체의 내부 필드에 접근할 때 사용
- 폼 처리와 함께 쓰임

```
<form th:object="${user}">
  <input th:field="*{name}" />
  <input th:field="*{email}" />
</form>
```

→ 내부적으로 `${user.name}`, `${user.email}`과 같음

`#{...}`: 메시지 표현식 (국제화)

- 국제화(i18n)를 위한 messages.properties 값을 가져올 때 사용

messages.properties

```
welcome=환영합니다, {0}님!
```

HTML

```
<p th:text="#{welcome(${user.name})}"></p>
```

출력 결과: "환영합니다, 홍길동님!"

- 파라미터도 넘길 수 있고, 다국어 처리할 때 필수

`@{...}`: URL 링크 표현식

- 동적 URL 생성할 때 사용
- 정적 리소스(JS/CSS/IMG), 라우팅 링크 전부 여기에 해당

```
<a th:href="@{/user/view(id=${user.id})}">프로필 보기</a>
```

- 위 예시 결과: `/user/view?id=3`
- 경로 변수 방식도 가능:

```
<a th:href="@{/user/{id}(id=${user.id})}">프로필</a>
```

→ `/user/3`

`~{...}`: fragment 표현식 (템플릿 조각 포함)

조각 템플릿을 포함하거나 재사용할 때 사용

```
<div th:replace="~{fragments/header :: nav}"></div>
```

- `fragments/header.html` 파일에서 `th:fragment="nav"` 부분을 가져옴

표현식 중 중첩 사용

```
<p th:text="#{greeting(${user.name})}"></p>     <!-- 메시지 안에 변수 -->
<a th:href="@{/user/edit(id=${user.id})}"></a>  <!-- URL 안에 변수 -->
```

예시 정리

```
<!-- 변수 표현식 -->
<p th:text="${user.name}">이름</p>

<!-- 선택 표현식 (폼 바인딩) -->
<form th:object="${user}">
  <input th:field="*{email}" />
</form>

<!-- 메시지 표현식 -->
<p th:text="#{welcome.message}">환영</p>

<!-- 링크 표현식 -->
<a th:href="@{/user/{id}(id=${user.id})}">상세</a>

<!-- 템플릿 조각 포함 -->
<div th:replace="~{layout/footer :: footerSection}"></div>
```

HTML 속성 조작

속성	설명	예시
`th:attr`	여러 속성 동시 지정	`th:attr="value=${v}, alt='이미지'"`
`th:class`	클래스 전체 덮어쓰기	`th:class="'btn ' + ${activeClass}"`
`th:classappend`	클래스 덧붙이기	`th:classappend="' active'"`
`th:style`	인라인 스타일 지정	`th:style="'color:' + ${color}"`
`th:styleappend`	스타일 덧붙이기	`th:styleappend="'font-size:14px;'"`
`th:checked`	체크 여부 조건부 설정	`th:checked="${checked}"`
`th:selected`	드롭다운 선택 여부	`th:selected="${gender == 'F'}"`
`th:disabled`	버튼 비활성화	`th:disabled="${!valid}"`
`th:readonly`	읽기 전용 필드	`th:readonly="${readonly}"`

`th:attr` vs `th:attrappend`, `th:attrprepend`

기본 문법

```
<input th:attr="name=${fieldName}, value=${fieldValue}" />
```

- 여러 속성을 동시에 동적으로 설정할 수 있음
- ,(쉼표)로 구분해서 설정

예시

```
<img th:attr="src=@{/img/logo.png}, alt='로고'" />
```

→ HTML 결과:

```
<img src="/img/logo.png" alt="로고" />
```

`th:class`, `th:classappend`

`th:class` - 클래스 속성 전체를 덮어쓰기

```
<div th:class="'box ' + ${isActive} ? 'active' : 'inactive'"></div>
```

`th:classappend` – 기존 class 뒤에 클래스 추가

```
<div class="box" th:classappend="${isError} ? ' error' : ''"></div>
```

→ HTML 결과:

```
<div class="box error"></div>
```

`th:style`, `th:styleappend`

`th:style`

```
<div th:style="'color:' + ${fontColor}"></div>
```

`th:styleappend`

```
<div style="margin:10px;" th:styleappend="'color:' + ${color}"></div>
```

→ 결과:

```
<div style="margin:10px;color:red;"></div>
```

동적으로 checked, selected, disabled 설정하기

조건부 속성 처리.

타임리프는 조건에 따라 특정 속성을 **존재하게 만들지** 말지를 제어할 수 있다.

조건이 true면 속성이 생성되고, false면 **속성 자체가 출력되지 않는다**.

체크박스 예시 (`th:checked`)

```
<input type="checkbox" th:checked="${user.agree}" />
```

- `user.agree == true` 면 `checked` 출력됨
- 결과:

```
<input type="checkbox" checked />
```

드롭다운 선택 (`th:selected`)

```
<option value="MALE" th:selected="${user.gender == 'MALE'}">남자</option>
<option value="FEMALE" th:selected="${user.gender == 'FEMALE'}">여자</option>
```

버튼 비활성화 (`th:disabled`)

```
<button th:disabled="${!form.valid}">제출</button>
```

읽기 전용 필드 (`th:readonly`)

```
<input th:readonly="${readonlyMode}" />
```

동적 속성 조작 예제 모음

```
<!-- 텍스트 필드 -->
<input type="text"
       th:attr="name=${fieldName}, value=${fieldValue}"
       th:readonly="${readonlyMode}" />

<!-- 버튼 -->
<button type="submit" th:disabled="${!canSubmit}">저장</button>

<!-- 라디오 버튼 -->
<input type="radio" name="role" value="admin"
       th:checked="${user.role == 'admin'}" />

<!-- 클래스 동적 조합 -->
<div class="base" th:classappend="${highlight ? ' highlight' : ''}"></div>

<!-- 스타일 동적 조합 -->
<div style="padding:10px;" th:styleappend="'background:' + ${color}"></div>
```

반복 처리 (th:each)

기능	문법
기본 반복	`th:each="item : ${list}"`
상태 추적	`item, stat : ${list}"`
중첩 반복	리스트 안의 리스트 반복
반복 + 변수	`th:each="item : ${list}" th:with="sum=..."`
리스트 없음 처리	`th:if="${#lists.isEmpty(list)}"`
맵 반복	`th:each="entry : ${map}"`

리스트 출력

```
<tr th:each="user : ${userList}">
  <td th:text="${user.name}">이름</td>
  <td th:text="${user.age}">나이</td>
</tr>
```

의미:

- `${userList}` : Model에서 넘겨준 리스트
- `user` : 리스트 안의 한 항목 (자바에서 for-each랑 같음)

> Java 문법: `for (User user : userList)`

컨트롤러 예제

```
@GetMapping("/users")
public String users(Model model) {
    List<User> userList = List.of(
        new User("홍길동", 32),
        new User("이순신", 28),
        new User("유관순", 25)
    );
    model.addAttribute("userList", userList);
    return "user_list";
}
```

출력 결과

```
<tr><td>홍길동</td><td>32</td></tr>
<tr><td>이순신</td><td>28</td></tr>
<tr><td>유관순</td><td>25</td></tr>
```

인덱스 값 활용 (`stat` 속성)

상태 추적 stat 변수 사용

stat 속성	의미
`index`	0부터 시작하는 인덱스
`count`	1부터 시작하는 카운트
`size`	리스트 전체 크기
`even`	짝수 번째 요소일 때 true
`odd`	홀수 번째 요소일 때 true
`first`	첫 번째 요소면 true

stat 속성	의미
`last`	마지막 요소면 true

```
<tr th:each="user, stat : ${userList}">
  <td th:text="${stat.index}">0</td>
  <td th:text="${stat.count}">1</td>
  <td th:text="${stat.even} ? '짝수' : '홀수'">짝수여부</td>
  <td th:text="${user.name}">이름</td>
</tr>
```

중첩 반복 처리

```
<div th:each="group : ${groupList}">
  <h3 th:text="${group.name}">그룹 이름</h3>
  <ul>
    <li th:each="member : ${group.members}" th:text="${member.name}">멤버 이름</li>
  </ul>
</div>
```

- 그룹별로 하위 요소까지 출력 가능

조건 반복 (필터링)

반복문에서 변수 선언(with 같이 쓰기)

```
<tr th:each="item : ${items}" th:with="total=${item.price + item.tax}">
  <td th:text="${item.name}"></td>
  <td th:text="${total}"></td>
</tr>
```

리스트가 비었을 때 처리

```
<tr th:if="${#lists.isEmpty(userList)}">
  <td colspan="2">등록된 사용자가 없습니다.</td>
</tr>
<tr th:each="user : ${userList}">
  <td th:text="${user.name}"></td>
  <td th:text="${user.age}"></td>
</tr>
```

Map 반복 처리

```
<tr th:each="entry : ${userMap}">
  <td th:text="${entry.key}">ID</td>
  <td th:text="${entry.value.name}">이름</td>
</tr>
```

인덱스 + 항목 같이 출력

```
<tr th:each="user, stat : ${userList}">
  <td th:text="${stat.count} + '. ' + ${user.name}">번호. 이름</td>
</tr>
```

조건 처리 (th:if / th:switch)

속성	설명	예시
`th:if`	조건이 true일 때 태그 출력	`<div th:if="${user != null}">`
`th:unless`	조건이 false일 때 태그 출력	`<p th:unless="${user.admin}">`
`th:switch`	여러 조건 기준 값 지정	`<div th:switch="${user.role}">`
`th:case`	조건 분기	`<span th:case="'ADMIN'">`
`th:case="*"`	default 분기	`<span th:case="*">기본</span>`

단일 조건

`th:if` - 조건이 true일 때 렌더리

```
<p th:if="${user.age >= 20}">성인입니다</p>
```

- 조건이 `true`면 태그가 출력됨
- 조건이 `false`면 **아예 HTML에 아무것도 안 나옴**

> 실제 출력: `<p>성인입니다</p>` 또는 출력 자체 없음

복수 조건

`th:if` 안에서 `th:with`로 변수 선언

```
<div th:if="${user != null}" th:with="isAdult=${user.age >= 20}">
  <p th:text="${isAdult} ? '성인' : '미성년자'"></p>
</div>
```

논리 연산자 사용

연산자	설명
`and`	그리고
`or`	또는
`not`	부정

```
<p th:if="${user.age >= 20 and user.age < 65}">일반 성인</p>
<p th:if="${user.age < 20 or user.age >= 65}">청소년 또는 노년층</p>
```

null 체크

```
<div th:if="${user != null}">
  <p th:text="${user.name}"></p>
</div>
```

조건부 속성 처리

```
<input type="checkbox" th:checked="${user.agree}" />
<button th:disabled="${!form.valid}">제출</button>
<option value="M" th:selected="${user.gender == 'M'}">남자</option>
```

switch-case 스타일 처리

`th:switch` + `th:case` 여러 조건 분기 처리(Java의 switch-case)

기본 문법

```
<div th:switch="${user.grade}">
  <p th:case="'A'">VIP 고객</p>
  <p th:case="'B'">골드 고객</p>
  <p th:case="*">일반 고객</p> <!-- default -->
</div>
```

- `th:switch`는 분기 기준이 되는 값
- `th:case`는 조건에 맞는 분기 처리
- `*`는 default (어느 것도 해당 안 되면)

실제 예시

```
model.addAttribute("user", new User("홍길동", "A"));
```

```
<div th:switch="${user.grade}">
  <span th:case="'A'">최상위 고객</span>
  <span th:case="'B'">우수 고객</span>
  <span th:case="*">일반 고객</span>
</div>
```

→ 출력 결과: "최상위 고객"

부정 조건 처리(`th:unless`)

`th:unless` – 조건이 false일 때 렌더링

```
<p th:unless="${user.age >= 20}">미성년자입니다</p>
```

- `th:if`의 반대 역할
- 조건이 거짓일 때만 출력됨

URL 처리

표현식	설명
`@{/path}`	기본 URL 처리
`@{/user/{id}(id=${user.id})}`	경로 변수 삽입
`@{/search(keyword=${kw})}`	쿼리 파라미터 처리
`@{/img/${filename}}`	이미지 동적 경로
`@{https://...}`	외부 URL

절대 경로와 상대 경로

표현식	의미
`@{/main}`	`/main` → 루트 기준 절대 경로
`@{main}`	현재 경로에 상대적인 `/현재경로/main`
`@{../up}`	상위 디렉터리 상대 경로 (`../`) 사용 가능

절대 경로 처리(외부 URL 처리)

```
<a th:href="@{https://www.google.com}">구글로</a>
```

→ 그대로 링크됨

상대 경로 처리

경로 변수(path variable) 처리

```
<a th:href="@{/user/{id}(id=${user.id})}">프로필</a>
```

user.id = 7일 경우:

```
<a href="/user/7">프로필</a>
```

- `{id}`는 경로 변수
- `(id=${...})`는 그 값을 지정

쿼리스트링 처리 (query parameter)

```
<a th:href="@{/search(keyword=${keyword})}">검색</a>
```

keyword = "타임리프"일 경우:

```
<a href="/search?keyword=타임리프">검색</a>
```

경로 변수 + 쿼리 파라미터 동시 사용

```
<a th:href="@{/user/{id}/edit(id=${user.id}, mode='admin')}">수정</a>
```

→ `/user/7/edit?mode=admin`

정적 리소스 경로 지정(이미지, CSS, JS)

```
<img th:src="@{/images/logo.png}" alt="로고" />
<link rel="stylesheet" th:href="@{/css/main.css}" />
<script th:src="@{/js/app.js}"></script>
```

정적 파일은 `src/main/resources/static` 아래 두면 `/경로`로 접근 가능

`@{...}` 링크 표현식

- 타임리프에서 동적 URL을 만드는 표현식
- 컨텍스트 루트(`http://localhost:8080`)를 자동으로 붙여줌
- HTML의 `href`, `src`, `action` 등에 동적으로 URL을 넣을 수 있음

기본 사용법

```
<a th:href="@{/home}">홈으로</a>
```

```
<a href="/home">홈으로</a>
```

쿼리 파라미터 붙이기

리디렉션 링크 처리

자주 쓰는 예시 모음

```
<!-- 홈으로 이동 -->
<a th:href="@{/}">홈</a>

<!-- 게시글 보기 -->
<a th:href="@{/post/{id}(id=${post.id})}">상세</a>

<!-- 회원 수정 -->
<a th:href="@{/member/{id}/edit(id=${member.id})}">수정</a>

<!-- 검색 (GET 파라미터) -->
<a th:href="@{/search(keyword=${keyword}, page=${page})}">검색</a>

<!-- 이미지 로딩 -->
<img th:src="@{/img/${item.imageFile}}" />

<!-- 스크립트/스타일 -->
<link th:href="@{/css/style.css}" rel="stylesheet" />
<script th:src="@{/js/script.js}"></script>
```

Form 처리

태그	쓰는 이유
`th:object`	바인딩할 전체 객체 지정
`th:field="*{field}"`	입력 필드 자동 바인딩, name/value/id 설정
`th:errors="*{field}"`	해당 필드의 검증 오류 출력
`#fields.hasErrors('field')`	필드에 오류 있는지 확인

`<form>` 태그와 `th:object`

타임리프에서 Form을 만들 때는 `th:object`로 바인딩할 객체를 지정하고,
각 input 필드에는 `th:field="*{필드명}"`을 써서 자동으로 name/value/id를 설정해준다.

HTML 예시

```
<form th:action="@{/user/save}" th:object="${user}" method="post">
  <input type="text" th:field="*{name}" />
  <input type="email" th:field="*{email}" />
  <button type="submit">저장</button>
</form>
```

Java DTO 예시

```
@Data
public class User {
    private String name;
    private String email;
}
```

컨트롤러

```
@GetMapping("/user/form")
public String form(Model model) {
    model.addAttribute("user", new User());
    return "userForm";
}
```

`<input>`, `<textarea>`, `<select>`에 `th:field` 사용

`th:field="*{name}"`은 아래와 같은 속성을 자동 생성:

```
<input type="text" name="name" id="name" value="홍길동" />
```

- **name**: 필드 이름 자동 설정
- **value**: 현재 값 자동 삽입
- **id**: name과 동일하게 설정됨

select, radio, checkbox도 가능

select (드롭다운)

```
<select th:field="*{gender}">
  <option value="M">남자</option>
  <option value="F">여자</option>
</select>
```

radio 버튼

```
<input type="radio" th:field="*{gender}" value="M" /> 남자
<input type="radio" th:field="*{gender}" value="F" /> 여자
```

checkbox

```
<input type="checkbox" th:field="*{agree}" /> 약관 동의
```

자동 바인딩, 검증 메시지 출력

유효성 검증 메시지 출력 (`BindingResult`)

```
@PostMapping("/user/save")
public String save(@Valid @ModelAttribute("user") User user,
                   BindingResult result) {
    if (result.hasErrors()) {
        return "userForm";
    }
    // 저장 처리
    return "redirect:/user/list";
}
```

타임리프에서 에러 출력

```
<div th:if="${#fields.hasErrors('name')}" th:errors="*{name}">이름 오류</div>
```

- `#fields.hasErrors('필드명')`: 해당 필드에 오류 있는지 확인
- `th:errors`: 오류 메시지 자동 출력

전체 예제 정리

HTML

```
<form th:action="@{/user/save}" th:object="${user}" method="post">
  <label for="name">이름</label>
  <input type="text" th:field="*{name}" />
  <div th:if="${#fields.hasErrors('name')}" th:errors="*{name}"></div>

  <label for="email">이메일</label>
  <input type="email" th:field="*{email}" />
  <div th:if="${#fields.hasErrors('email')}" th:errors="*{email}"></div>

  <button type="submit">저장</button>
</form>
```

DTO (javax.validation)

```java
import jakarta.validation.constraints.*;

@Data
public class User {
    @NotBlank
    private String name;

    @Email
    @NotBlank
    private String email;
}
```

기타 자주 쓰는 패턴

기능	코드 예시
라디오 그룹 바인딩	`th:field="*{gender}"` + 여러 input[type=radio]
선택된 옵션 유지	`<option value="F" selected>` → 자동 처리됨
체크박스 체크 여부	`th:field="*{agree}"`로 자동 처리
텍스트에 바인딩된 값 출력	`<input th:field="*{username}" />`
유효성 실패 시 오류 표시	`th:errors="*{username}"`

폼 제출과 Spring Controller 연동

템플릿 레이아웃 조각화

속성	설명	비고
`th:fragment="이름"`	조각 정의	템플릿 내부에서 사용
`th:replace="파일 :: 조각명"`	조각으로 해당 태그 전체 대체	가장 많이 씀
`th:insert="파일 :: 조각명"`	조각을 현재 태그 내부에 삽입	`div` 유지됨
`~{::main}`	현재 페이지의 특정 태그 전달	레이아웃 적용 시

조각화(Fragmentation)란?

- 여러 HTML 파일에 반복되는 구조(예: header, footer, nav)를 **하나의 파일로 관리하고 재사용**하는 것
- 유지보수가 쉬워지고, 일관성 있는 레이아웃 구성 가능
- 타임리프는 이를 위해 **fragment(조각)** 기능을 제공

기본 흐름

1. `fragment` 정의하기 → 레이아웃 파일에서 조각을 만든다
2. `replace`, `insert`, `include`로 조각을 다른 템플릿에 포함한다.

`th:fragment` - 조각 선언

```
<!-- fragments/layout.html -->
<div th:fragment="header">
  <header>
    <h1>사이트 제목</h1>
    <nav>...네비게이션...</nav>
  </header>
</div>
```

- `th:fragment="header"` → 이 블록은 "header"라는 이름의 조각
- 원하는 이름으로 선언 가능

`th:insert`, `th:replace`, `th:include` 차이점

`th:insert` - 조각 "내부에 삽입"

```
<div th:insert="fragments/layout :: header"></div>
```

- 현재 `<div>`는 유지하고, 안쪽에 fragment가 **삽입됨**
- `<div><fragment내용></div>` 구조가 됨

`th:replace` - 조각 "전체를 대체"

```
<div th:replace="fragments/layout :: header"></div>
```

- 해당 `<div>` 태그를 **통째로 fragment의 내용으로 바꿈**
- HTML이 깔끔하게 대체됨 (가장 많이 쓰임)

`th:include` - deprecated (예전 방식)

- 거의 안 씀. `insert`나 `replace` 사용 권장

실제 예제 구조

- `templates/fragments/layout.html`

```
<!DOCTYPE html>
<html>
  <body>
    <div th:fragment="header">
      <header><h1>헤더입니다</h1></header>
    </div>

    <div th:fragment="footer">
      <footer><p>© 2025 MySite</p></footer>
    </div>
  </body>
</html>
```

- `templates/main.html`

```
<html>
  <body>
    <div th:replace="fragments/layout :: header"></div>

    <main>
      <h2>본문 영역</h2>
    </main>

    <div th:replace="fragments/layout :: footer"></div>
  </body>
</html>
```

공통 레이아웃 구성 (header, footer, sidebar 등)

템플릿 조각에서 파라미터 넘기기 (`th:with`)

fragment 선언

```
<div th:fragment="greeting(name)">
  <p th:text="'안녕하세요, ' + ${name} + '님!'"></p>
</div>
```

fragment 호출

```
<div th:replace="fragments/greet :: greeting('홍길동')"></div>
```

→ 출력:

```
<p>안녕하세요, 정석님!</p>
```

레이아웃 템플릿 라이브러리 (e.g., Layout Dialect)

보통 이렇게 구성함

- `templates/layout/layout.html`

```
<!DOCTYPE html>
<html>
  <head th:fragment="head(title)">
    <title th:text="${title}">기본 제목</title>
    <link rel="stylesheet" href="/css/style.css" />
  </head>
  <body>
    <div th:fragment="body(content)">
      <header th:replace="layout/common :: header"></header>
      <main th:replace="${content}"></main>
      <footer th:replace="layout/common :: footer"></footer>
    </div>
  </body>
</html>
```

- `templates/page/user.html`

```
<!DOCTYPE html>
<html th:replace="layout/layout :: body(~{::main})">
  <main>
    <h2>회원 페이지</h2>
    <p>내용입니다.</p>
  </main>
</html>
```

메시지 처리 (국제화)

메시지 표현식이란? (`#{...}`)

```
<p th:text="#{welcome}">환영합니다</p>
```

- `#{...}`는 `messages.properties` 등에서 지정한 메시지를 찾아서 출력해주는 표현식이다.
- 실제 문장은 `src/main/resources/messages.properties`에서 정의함

기능	표현식/파일	설명
메시지 표현식	`#{key}`	메시지 파일에서 값 불러오기
파라미터 메시지	`#{key(${val})}`	`{0}`, `{1}` 치환
메시지 파일	`messages.properties`	기본 메시지 정의
다국어 파일	`messages_en.properties`	언어별 메시지 정의
유효성 메시지	`@NotBlank(message = "{key}")`	검증 오류 메시지 연동
대체 메시지	`#{key} ?: '기본값'`	메시지 없을 때 fallback

`messages.properties`, `messages_ko.properties`

메시지 파일 만들기

기본 메시지 파일

- `src/main/resources/messages.properties`

```
welcome=환영합니다!
greeting=안녕하세요, {0}님!
```

영어 메시지 파일

- `src/main/resources/messages_en.properties`

```
welcome=Welcome!
greeting=Hello, {0}!
```

Spring Boot 기본 설정

자동으로 설정된다.
application.yml이나 application.properties에서 별도로 설정하지 않아도
`messages.properties`는 자동으로 적용된다.

> 다만 메시지 파일 이름은 `messages`여야 함

메시지 사용 예시

```
<!-- 일반 메시지 -->
<p th:text="#{welcome}">기본 텍스트</p>

<!-- 파라미터 있는 메시지 -->
<p th:text="#{greeting(${user.name})}">안녕하세요!</p>
```

→ 메시지 내용:

```
greeting=안녕하세요, {0}님!
```

→ 출력 결과:

```
<p>안녕하세요, 정석님!</p>
```

`#{message.key}` 사용

기본 메시지, 파라미터 메시지

폼 오류 메시지랑 연동

유효성 검증 메시지와 연동

```
@NotBlank(message = "{user.name.required}")
private String name;
```

- messages.properties

  ```
  user.name.required=이름은 필수입니다.
  ```

- HTML

  ```
  <div th:if="${#fields.hasErrors('name')}" th:errors="*{name}">기본 오류 메시지</div>
  ```

  → 자동으로 `messages.properties`의 `user.name.required`를 찾아서 출력해줘

예외 상황 처리

널(null) 값 처리 방법

문제 상황

타임리프에서 아래처럼 작성했을 때:

```
<p th:text="${user.name}">이름 없음</p>
```

- `user`가 **null이면 예외가 발생**해서 페이지 렌더링이 멈출 수 있어.
- 특히 서버에서 Model 데이터를 제대로 못 넣었을 때 이런 일이 자주 생김.

해결법

1. **`th:if`로 null 체크**

   ```
   <div th:if="${user != null}">
     <p th:text="${user.name}">이름</p>
   </div>
   ```

   - `user`가 null일 경우, `<p>` 태그 자체가 렌더링되지 않음.

2. `?:` 엘비스 연산자로 기본값 제공

   ```
   <p th:text="${user.name} ?: '이름 없음'"></p>
   ```

- `user.name`이 null일 경우 → `'이름 없음'` 이 출력됨.

3. `#strings.isEmpty()` 같이 유틸 활용

```
<p th:if="${!#strings.isEmpty(user.name)}" th:text="${user.name}"></p>
```

Null-safe 연산 (`${user?.name}` 등)

안전한 네비게이션 연산자 (`?.`)

타임리프 3.1+에서는 **null-safe 호출 연산자** `?.` 를 지원해.

```
<p th:text="${user?.name}">이름</p>
```

- `user`가 null이면 → 아무것도 출력 안 됨.
- `user`가 null이 아니면 → `user.name` 출력.

예시 비교

표현식	동작
`${user.name}`	user가 null이면 예외 발생
`${user?.name}`	user가 null이면 빈 문자열 출력

예외 발생 시 화면 대응 전략

디폴트 메시지 제공

```
<span th:text="${user?.name} ?: '이름 없음'"></span>
```

- user 또는 user.name이 null이면 '이름 없음' 출력.

서버 측 예외 발생 시 처리

Spring Boot에서는 기본적으로 다음 경로의 템플릿을 제공하면 에러 페이지로 자동 연결.

예시

- `/templates/error/404.html`
- `/templates/error/500.html`
- `/templates/error/403.html`

> 해당 상태 코드에 맞춰 자동 라우팅됨.

커스텀 오류 메시지 영역 추가

```
<div th:if="${errorMessage}">
  <p th:text="${errorMessage}"></p>
</div>
```

- 서버에서 에러 메시지를 넘길 수 있음:

  ```
  model.addAttribute("errorMessage", "사용자 정보가 없습니다.");
  ```

`BindingResult` 검증 실패 메시지 처리

```
<form th:object="${user}">
  <input th:field="*{name}" />
  <div th:errors="*{name}"></div>
</form>
```

- `@Valid` 검증 실패 시 자동으로 메시지를 표시함

실전 디버깅 팁

- 디버깅할 땐 아래처럼 `DEBUG` 출력 코드를 활용:

  ```
  <span th:text="'[DEBUG] user: ' + ${user}"></span>
  ```

  - `user == null` 여부를 한눈에 알 수 있어

Fragment (조각 템플릿)

항목	예시	설명
선언	`th:fragment="footer"`	조각 정의
사용	`th:replace="file :: fragment"`	해당 위치에 조각 삽입
파라미터	`th:fragment="name(p1, p2)"`	인자 전달 가능
중첩 조각	`layout → fragment → fragment`	계층적 레이아웃 구성

선언: `th:fragment`

목적

템플릿의 일부분(예: header, footer, nav, alert 등)을 조각으로 만들어 **재사용**할 수 있게 함.

기본 문법

```
<!-- templates/fragments/layout.html -->
<div th:fragment="footer">
  <footer>
    <p>© 2025 MySite</p>
  </footer>
</div>
```

- `th:fragment="footer"`: 이 블록을 "footer"라는 이름의 조각으로 선언한 것.
- 이름만 지정하면 되고, 다른 파일에서 호출 가능.

사용: `th:insert`, `th:replace`, `th:include`

기본 구조

```
<!-- 조각 사용 -->
<div th:replace="fragments/layout :: footer"></div>
```

각 방식 차이

속성	설명	렌더링 결과
`th:replace`	**자기 자신을 fragment로 완전히 대체**	`<div>` 사라지고 fragment로 바뀜
`th:insert`	fragment를 **내부에 삽입**	`<div>` 안에 fragment 삽입됨
`th:include`	`insert`와 유사하지만 거의 안 씀 (deprecated)	비권장

예시

```
<!-- header.html -->
<div th:fragment="header">
  <h1>사이트 제목</h1>
</div>

<!-- base.html -->
<body>
  <div th:replace="fragments/header :: header"></div>
</body>
```

파라미터 있는 조각

fragment 선언 시 파라미터 받기

```
<!-- fragments/alert.html -->
<div th:fragment="msgBox(message, type)">
  <div th:class="'alert ' + ${type}">
    <p th:text="${message}">메시지</p>
  </div>
</div>
```

- `message`, `type` 파라미터를 전달받음

호출할 때 인자 넘기기

```
<div th:replace="fragments/alert :: msgBox('저장되었습니다', 'alert-success')"></div>
```

조각 중첩 포함 구조

레이아웃 구조에 조각이 또 들어가는 경우

```
<!-- layout/base.html -->
<html>
  <body>
    <div th:fragment="layout(content)">
      <div th:replace="fragments/header :: header"></div>
      <main th:replace="${content}">본문</main>
      <div th:replace="fragments/footer :: footer"></div>
    </div>
  </body>
</html>
```

실제 페이지에서 조각 전달

```
<!-- page/user.html -->
<html th:replace="layout/base :: layout(~{::main})">
  <main>
    <h2>회원 전용 페이지</h2>
  </main>
</html>
```

- `~{::main}` 은 현재 페이지의 `<main>` 태그를 통째로 넘기는 것
- 중첩된 구조에서 레이아웃의 'content' 영역에 채워짐

중첩 구조 요약

```
page.html → layout.html → header/footer fragment
```

- 하나의 템플릿이 여러 fragment를 포함하고,
- 그 fragment가 또 다른 fragment를 포함하는 방식으로 **모듈화** 가능

타임리프에서 자바스크립트 사용

기능	문법	설명
JS 안에서 타임리프 사용	`th:inline="javascript"`	반드시 필요
값 삽입	`[[${...}]]`	숫자/문자/불린 모두 가능
URL 삽입	`[[@{/url}]]`	컨텍스트 자동 포함
JSON 객체	`[[${object}]]`	자동 직렬화
JS 주석에서도 동작	`/*[[...]]*/`	가능

`th:inline="javascript"`로 JS 안에 변수 쓰기

기본 개념

타임리프는 JavaScript 코드 안에 `${...}` 표현식을 넣으면 **기본적으로 인식하지 못한다.**
→ **그래서** `<script>` **태그에** `th:inline="javascript"` ** 속성을 줘야 한다.

```
<script th:inline="javascript">
  let username = [[${user.name}]];
</script>
```

- `[[...]]` 는 타임리프의 JS 전용 표현식
- 이 코드는 실제 렌더링 시:

```
<script>
  let username = "홍길동";
</script>
```

JS에서 서버 데이터 바인딩하는 법

숫자, 문자열, 불린 값 바인딩

```
<script th:inline="javascript">
  let id = [[${user.id}]];                  // 숫자 → 1
  let name = [[${user.name}]];              // 문자열 → "홍길동"
  let isAdmin = [[${user.admin}]];          // 불린 → true
```

```
</script>
```

- `user`는 컨트롤러에서 전달한 Model 객체라고 가정

URL 바인딩 (`@{}`도 사용 가능)

```
<script th:inline="javascript">
  let baseUrl = [[@{/}]]; // → "/"
  let profileUrl = [[@{/user/profile(id=${user.id})}]];
</script>
```

- 타임리프의 URL 표현식도 JS에서 사용 가능
- `@{}`는 컨텍스트 루트를 자동으로 포함함

JSON 형태 데이터 삽입

Java 객체 전체를 JSON으로 넘기기

```
<script th:inline="javascript">
  let user = [[${user}]];
</script>
```

- `user`가 자바 객체일 경우, 자동으로 JSON 변환됨
- 예:

  ```
  user = new User("홍길동", "admin", true);
  ```

  → 렌더링 결과:

  ```
  <script>
    let user = {"name":"홍길동","role":"admin","active":true};
  </script>
  ```

배열, 리스트, 맵

```
<script th:inline="javascript">
  let roles = [[${user.roles}]];
</script>
```

- `roles`가 `List<String> roles = List.of("USER", "ADMIN")`이라면 → `["USER", "ADMIN"]`

추가 팁

상황	표현식
JS 문자열 안전하게 출력	`[[${'문자열 "안전" 처리'}]]`

상황	표현식
변수 존재 여부에 따라 JS 로직 제어	`th:if="${user != null}"` 로 `<script>` 자체 조건 분기
주석으로 감싸기	`/*[[${user.name}]]*/` → JS 주석 안에서도 작동
유틸 함수 호출도 가능	`[[${#dates.format(user.createdAt, 'yyyy-MM-dd')}]]`

타임리프와 스프링 연동 (심화)

항목	설명
`Model.addAttribute()`	기본 데이터 전달
`@ModelAttribute`	객체 자동 바인딩
DTO vs Form vs Entity	계층 분리 권장
Form 객체	입력/검증 전용
`RedirectAttributes`	리다이렉트 후 메시지 전달
`${key}`	타임리프에서 모델 접근 방법

Spring MVC + 타임리프 구조

전체 흐름 요약

```
Controller → Model 데이터 생성 → ViewResolver → Thymeleaf Template → HTML 렌더링
```

- 컨트롤러는 데이터를 Model에 담고
- 뷰 리졸버(ViewResolver)가 `.html` 템플릿을 찾아서 타임리프로 렌더링

컨트롤러 예시

```
@GetMapping("/user/profile")
public String profile(Model model) {
    model.addAttribute("user", new User("정석", "admin"));
    return "user/profile";
}
```

- `/templates/user/profile.html`

  ```
  <p th:text="${user.name}">이름</p>
  ```

Model 객체 전달

기본 형태

```
model.addAttribute("key", value);
```

타임리프에서는 `${key}` 로 접근 가능

@ModelAttribute 자동 전달

```
@GetMapping("/form")
public String form(@ModelAttribute("user") User user) {
    return "user/form";
}
```

- 자동으로 `user` 를 model에 바인딩
- 타임리프에서는 `th:object="${user}"` 로 사용 가능

DTO, Entity, Form 객체 구분해서 전달하기

실무에서는 이렇게 구분

종류	설명	용도
Entity	DB와 직접 매핑되는 클래스	Repository에서 사용
DTO	계층 간 데이터 전달용 (View용)	서비스 ↔ 컨트롤러 ↔ 뷰
Form 객체	입력값 받기 전용	`@Valid` 검증 대상

예시

```
@GetMapping("/post/edit/{id}")
public String edit(@PathVariable Long id, Model model) {
    Post post = postService.findById(id);
    PostForm form = new PostForm(post);  // Entity → Form 변환
    model.addAttribute("postForm", form);
    return "post/edit";
}
```

- 타임리프

  ```
  <form th:object="${postForm}">
    <input th:field="*{title}" />
  </form>
  ```

  - 직접 Entity를 바인딩하지 않고, 반드시 별도의 Form 객체 사용
  - 이유: 유효성 검증, 보안, 입력 제한 등을 위해

RedirectAttributes 사용

목적

- 리다이렉트 이후에도 메시지를 전달하기 위해
- `Model`은 리다이렉트 후 사라짐 → 대신 Flash 사용

사용 방법

```
@PostMapping("/user/save")
public String save(@Valid @ModelAttribute("user") User user,
                   BindingResult result,
                   RedirectAttributes redirectAttributes) {

    if (result.hasErrors()) return "user/form";

    userService.save(user);

    redirectAttributes.addFlashAttribute("message", "저장 완료!");
    return "redirect:/user/list";
}
```

- 타임리프 메시지 출력

```
<div th:if="${message}" th:text="${message}"></div>
```

핵심 개념

`addFlashAttribute()`는 **리다이렉트 이후 1회만 유지되는 세션 스코프 저장소**

사용자 정의 유틸리티

항목	설명	예시
내장 유틸	기본 제공 기능	`#dates`, `#strings`, `#lists`
커스텀 유틸	Java 클래스 + `@Component("myUtil")`	`${#myUtil.xxx()}`
Dialect 확장	DSL 수준의 커스텀 속성	`my:money="${price}"`
사용 위치	`th:text`, `th:if`, `th:each`, JS 등	모든 표현식 내 가능

`#dates`, `#numbers`, `#strings` 등 내장 객체

타임리프는 기본적으로 자주 쓰이는 기능을 위한 **내장 유틸리티 객체**들을 제공.

템플릿에서 `#strings`, `#dates` 같은 방식으로 호출할 수 있다.

내장 객체	대표 기능	예시
`#dates`	날짜 포맷/비교	`#dates.format(post.createdAt, 'yyyy-MM-dd')`
`#strings`	문자열 다루기	`#strings.substring(str, 0, 10)`
`#numbers`	숫자 포맷	`#numbers.formatDecimal(123456.789, 1, 2)`
`#lists`	리스트 유틸	`#lists.isEmpty(users)`
`#maps`	맵 관련 함수	`#maps.entries(myMap)`
`#fields`	검증 메시지	`#fields.hasErrors('email')`
`#request`, `#session`, `#response`	HTTP API 접근	`#session.getAttribute('loginUser')`

커스텀 유틸리티 클래스 만들기

언제 쓰는가?

- 자주 쓰는 문자열 변환, 날짜 포맷, 권한 변환 등을 뷰 단에서 처리하고 싶을 때
- 복잡한 `th:if`, `th:text` 조건식을 깔끔하게 만들고 싶을 때

유틸리티 클래스 작성

```
@Component("myUtil")
public class MyUtil {

    public String formatDate(LocalDateTime time) {
        return time.format(DateTimeFormatter.ofPattern("yyyy.MM.dd HH:mm"));
    }

    public String limitLength(String input, int max) {
        return input.length() > max ? input.substring(0, max) + "…" : input;
    }

    public String getRoleName(String code) {
        return switch (code) {
            case "ADMIN" -> "관리자";
            case "USER" -> "회원";
            default -> "알 수 없음";
        };
    }
}
```

- 반드시 `@Component("이름")` 으로 빈으로 등록해줘야 타임리프에서 사용 가능

타임리프에서 사용하기

```
<p th:text="${#myUtil.formatDate(post.createdAt)}"></p>
<p th:text="${#myUtil.limitLength(post.title, 30)}"></p>
<p th:text="${#myUtil.getRoleName(user.role)}"></p>
```

- `${#myUtil.메서드(...)}` 형태로 호출

Dialect 확장 (고급)

목적

- **커스텀 `th:\*` 속성을 직접 만들고 싶을 때**
- 예: `th:money="item.price"` → 자동으로 원화 포맷 적용되게 하기

기본 구성

Dialect를 직접 만들려면 3가지를 구현해야 해:

1. **Dialect 클래스**
   → `IDialect`, `AbstractProcessorDialect` 상속
2. **Processor 클래스**
   → `AbstractAttributeTagProcessor` 상속해서 `th:xxx` 구현
3. **템플릿 엔진에 등록**

예시: `th:money` 확장 만들기

```
public class MoneyAttributeTagProcessor extends AbstractAttributeTagProcessor {

    public MoneyAttributeTagProcessor(String dialectPrefix) {
        super(TemplateMode.HTML, dialectPrefix, "money", true, null, false, 1000);
    }

    @Override
    protected void doProcess(...) {
        BigDecimal value = (BigDecimal) expression.getValue(context);
        String formatted = NumberFormat.getCurrencyInstance().format(value);
        structureHandler.setBody(formatted, false);
    }
}
```

```java
public class MyDialect extends AbstractProcessorDialect {
    public MyDialect() {
        super("My Dialect", "my", 1000);
    }

    @Override
    public Set<IProcessor> getProcessors(...) {
        return Set.of(new MoneyAttributeTagProcessor(getPrefix()));
    }
}
```

그리고 `SpringTemplateEngine`에 등록:

```java
@Bean
public SpringTemplateEngine templateEngine(...) {
    SpringTemplateEngine engine = new SpringTemplateEngine();
    engine.addDialect(new MyDialect());
    return engine;
}
```

결과적으로 HTML에서 이렇게 사용 가능:

```html
<span my:money="${product.price}"></span>
```

보안 처리

항목	설명	보안 효과
`th:text`	HTML 이스케이프 기본 적용	✅ XSS 방지
`th:utext`	이스케이프 안 됨	❌ 위험 (신뢰된 HTML만 사용)
`_csrf`	POST 요청에 토큰 삽입	✅ CSRF 방지
`sec:authorize`	권한에 따른 조건 렌더링	✅ 정보 노출 차단
`sec:authentication`	사용자 정보 출력	✅ 사용자 상태 확인

HTML 이스케이프 (`th:text`)

목적: XSS(크로스사이트 스크립팅) 방지

사용자 입력을 HTML에 출력할 때, 스크립트나 태그가 그대로 노출되면 보안 위협 발생

기본 방식

```
<p th:text="${comment.content}"></p>
```

- `th:text`는 기본적으로 **HTML 이스케이프가 자동 적용됨**
- 예: `<script>alert(1)</script>` → `<script>alert(1)</script>`로 출력

기본적으로 안전하다.

`th:utext`를 쓸 때 주의할 점

`th:utext`란?

```
<div th:utext="${content}"></div>
```

- utext = "Unescaped text"
- HTML 태그를 **그대로 렌더링함 (안전장치 없음)**

위험성

- `<script>`나 악성 `<img onerror="...">` 등이 그대로 실행됨
- 사용자 입력, 외부 입력에는 절대 쓰지 말 것.

안전하게 쓰는 조건

- 오직 **신뢰된 관리자/시스템에서 생성한 HTML만** 넣을 것.
- 또는 Markdown → Safe HTML 변환처럼 사전 필터링 거쳐야 함.

CSRF 토큰 삽입 (`<input type="hidden">`)

Spring Security에서 CSRF 보호 기본 활성화됨

```
<form th:action="@{/post/create}" method="post">
  <input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}" />
  <input type="text" name="title" />
</form>
```

- `${_csrf.parameterName}` → 예: `_csrf`
- `${_csrf.token}` → 토큰 값

자동으로 삽입되는 방법

- Spring Security + Thymeleaf를 쓰면
- `<form:form>` 또는 타임리프 `<form>` 안에 위 내용을 **자동으로** 삽입해줘야 POST가 성공함

인증된 사용자 정보 출력 (Spring Security 연동)

타임리프에 Spring Security 연동하기

1. 의존성 추가 (thymeleaf-extras)

```
implementation 'org.thymeleaf.extras:thymeleaf-extras-springsecurity6'
```

2. HTML 네임스페이스 선언

```
<html xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
```

로그인 여부 확인

```
<div sec:authorize="isAuthenticated()">
  <p>환영합니다, <span sec:authentication="name"></span>님</p>
</div>

<div sec:authorize="isAnonymous()">
  <a th:href="@{/login}">로그인</a>
</div>
```

- `isAuthenticated()` → 로그인한 사용자
- `isAnonymous()` → 비로그인 사용자

사용자 정보 출력 예시

```
<span sec:authentication="principal.username"></span>
<span sec:authentication="principal.authorities"></span>
```

표현	설명
`name`	사용자 이름 (보통 아이디)
`principal.username`	UserDetails.getUsername()
`principal.authorities`	권한 목록

스프링 시큐리티 + 타임리프

항목	예시	설명
`sec:authorize`	`"isAuthenticated()"`, `"hasRole('ADMIN')"`	조건부 렌더링
`sec:authentication`	`"name"`, `"principal.username"`	사용자 정보 출력
권한별 메뉴 분기	`<li sec:authorize="...">`	로그인 상태/권한별 메뉴 구성

항목	예시	설명
사용자 정보 주입	`@AuthenticationPrincipal`	컨트롤러에서 정보 접근
로그아웃 처리	`/logout` 링크 or form	시큐리티에서 자동 처리

`sec:authorize`, `sec:authentication`

준비: 의존성 추가

```
implementation 'org.thymeleaf.extras:thymeleaf-extras-springsecurity6'
```

> 버전에 따라 `springsecurity5`, `springsecurity6` 구분 필요
> Spring Boot 3.x 이상은 `6` 사용

네임스페이스 선언 (꼭 필요)

```
<html xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
```

sec:authorize

- 조건에 따라 특정 HTML 블록을 보여줄지 말지 결정

```
<div sec:authorize="isAuthenticated()">
  로그인한 사용자만 볼 수 있음
</div>

<div sec:authorize="hasRole('ADMIN')">
  관리자 전용 메뉴
</div>

<div sec:authorize="hasAnyRole('ADMIN', 'MANAGER')">
  관리자 또는 매니저 가능
</div>
```

sec:authentication

- 현재 로그인된 사용자의 정보를 템플릿에 출력

```
<span sec:authentication="name"></span> <!-- 사용자 이름 -->
<span sec:authentication="principal.username"></span> <!-- UserDetails 기반 -->
<span sec:authentication="principal.authorities"></span> <!-- 권한 목록 -->
```

권한별 메뉴 렌더링

실전 예시

```
<ul>
  <li><a th:href="@{/}">홈</a></li>

  <li sec:authorize="isAuthenticated()">
    <a th:href="@{/user/mypage}">마이페이지</a>
  </li>

  <li sec:authorize="hasRole('ADMIN')">
    <a th:href="@{/admin/dashboard}">관리자 페이지</a>
  </li>

  <li sec:authorize="isAnonymous()">
    <a th:href="@{/login}">로그인</a>
  </li>

  <li sec:authorize="isAuthenticated()">
    <a th:href="@{/logout}">로그아웃</a>
  </li>
</ul>
```

- 권한에 따라 메뉴 구성 분기 → 메뉴 하드코딩 안 해도 됨
- 비로그인/로그인/관리자 모두 분기 가능

로그인 사용자 정보 가져오기

방법 1: `sec:authentication`으로 타임리프에서 직접 출력

```
<p>반갑습니다, <span sec:authentication="name"></span>님</p>
```

방법 2: 컨트롤러에서 `@AuthenticationPrincipal` 사용

```
@GetMapping("/mypage")
public String myPage(@AuthenticationPrincipal UserDetails user, Model model) {
    model.addAttribute("username", user.getUsername());
    return "user/mypage";
}
```

```
<span th:text="${username}">로그인 사용자</span>
```

- 보안 로직을 컨트롤러에서 처리하고 싶을 때 이 방법을 사용

로그아웃 링크 처리

로그아웃은 반드시 POST 또는 Spring Security에서 지정한 `/logout` 사용

```
<!-- GET 방식 로그아웃 링크 (Spring Security 기본 설정과 호환) -->
<a th:href="@{/logout}">로그아웃</a>
```

- 스프링 시큐리티가 `/logout` URL을 자동으로 핸들링함
- POST로 제한하는 경우에는 form 전송으로 바꿔야 함:

```
<form th:action="@{/logout}" method="post">
  <button type="submit">로그아웃</button>
</form>
```

유닛 테스트 및 디버깅

항목	설명
TemplateEngine 테스트	타임리프 단독 실행 테스트
MockMvc 테스트	HTML 렌더링 결과 포함한 통합 테스트
`model().attributeExists()`	Model 값 존재 확인
`th:text="'DEBUG: ' + ${val}"`	디버깅용 출력
로그 레벨 DEBUG	표현식 오류 추적 가능

타임리프 단독 테스트 방법

목적

- Spring 없이 타임리프 템플릿 엔진만으로 **HTML 렌더링이 정상적으로 되는지 확인**
- 로직 복잡한 fragment, 반복문, 조건문을 테스트할 때 유용

기본 코드 예시

```
@Test
void 타임리프_단독_렌더링_테스트() {
    TemplateEngine engine = new TemplateEngine();
    ClassLoaderTemplateResolver resolver = new ClassLoaderTemplateResolver();
    resolver.setPrefix("templates/");
    resolver.setSuffix(".html");
    resolver.setTemplateMode(TemplateMode.HTML);
    engine.setTemplateResolver(resolver);

    Context context = new Context();
    context.setVariable("name", "홍길동");
```

```
    String result = engine.process("hello", context);
    System.out.println(result);
    assertTrue(result.contains("홍길동"));
}
```

- `templates/hello.html` 템플릿을 읽어서 `"홍길동"`이 잘 들어갔는지 확인
- **프래그먼트나 JSON 조각 테스트**에도 응용 가능

테스트에서 `Model` 넘겨서 렌더링 확인

Spring MVC + MockMvc 테스트로 전체 흐름 검증

```
@WebMvcTest(MyController.class)
class MyControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void 회원_리스트_렌더링_테스트() throws Exception {
        mockMvc.perform(get("/users"))
            .andExpect(status().isOk())
            .andExpect(content().string(containsString("회원 목록")))
            .andExpect(content().string(containsString("정석")));
    }
}
```

Model 전달 확인

```
@Test
void 모델_전달_확인() throws Exception {
    mockMvc.perform(get("/form"))
        .andExpect(status().isOk())
        .andExpect(model().attributeExists("userForm"));
}
```

- 컨트롤러에서 `model.addAttribute("userForm", new UserForm())` 했는지 테스트 가능

로그로 템플릿 디버깅하기

디버깅용 출력 태그 작성

```
<span th:text="'[DEBUG] 사용자 이름: ' + ${user.name}"></span>
```

- 직접 화면에 값을 출력해서 **null인지, 잘 들어왔는지, 조건 분기 잘 되는지** 확인

조건 분기 테스트

```
<div th:if="${user != null}">✅ 유저 있음</div>
<div th:if="${user == null}">❌ 유저 없음</div>
```

로그 설정을 통한 디버깅 (application.yml)

```
logging:
  level:
    org.thymeleaf: DEBUG
```

- 템플릿 렌더링 과정에서 문제 생기면 **표현식 오류** 로그 확인 가능

Null 오류 예시 로그

```
Exception evaluating SpringEL expression: “${user.name}” (template: "profile" - line 10)
```

- `${user}`가 null이면 이런 에러 발생 → `th:if="${user != null}"`로 방어 필요

타임리프 확장 기능들

기능	설명	예시
Layout Dialect	레이아웃 상속/구성	`layout:decorate`, `layout:fragment`
Extras: Security	인증 상태 분기	`sec:authorize`, `sec:authentication`
Extras: Java8 Time	LocalDateTime 포맷	`#temporals.format(...)`
Standard Dialect	기본 문법 구성	`th:text`, `th:each`, ...
Custom Dialect	DSL 확장 가능	`my:money`, `my:roleName` 등

Layout Dialect

목적

타임리프 기본 기능만으로는 복잡한 레이아웃 구조 구현이 번거롭기 때문에,

템플릿 상속, **영역(fragment) 바꾸기**, **슬롯 채우기** 같은 기능을 제공해주는 확장 모듈이야.

의존성 추가 (Gradle)

```
implementation 'nz.net.ultraq.thymeleaf:thymeleaf-layout-dialect:3.1.0'
```

기본 사용법

- `layout/base.html` – 레이아웃 템플릿

```
<html layout:decorate="~{fragments/layout}">
  <body>
    <header th:replace="fragments/header :: header"></header>
    <main layout:fragment="content">
      기본 콘텐츠
    </main>
    <footer th:replace="fragments/footer :: footer"></footer>
  </body>
</html>
```

- `user/mypage.html` – 자식 페이지

```
<html layout:decorate="~{layout/base}">
  <section layout:fragment="content">
    <h2>마이페이지</h2>
  </section>
</html>
```

주요 태그 요약

태그	설명
`layout:decorate`	부모 템플릿 지정
`layout:fragment`	자식이 덮어쓸 영역 지정
`layout:include`	내부 삽입 (조각)
`layout:replace`	조각 대체 (기존과 유사)

Extras (Spring Security, Java 8 Time, Data)

타임리프는 다음과 같은 **부가 모듈(Extras)** 들을 공식으로 지원해:

thymeleaf-extras-springsecurity6

- 인증 정보, 권한 분기, 로그인 상태 확인 가능

```
implementation 'org.thymeleaf.extras:thymeleaf-extras-springsecurity6'
```

사용 예:

```
<div sec:authorize="hasRole('ADMIN')">관리자 전용</div>
<span sec:authentication="name"></span>
```

thymeleaf-extras-java8time

- Java 8의 `LocalDateTime`, `ZonedDateTime` 포맷팅 지원

```
implementation 'org.thymeleaf.extras:thymeleaf-extras-java8time'
```

예시:

```
<p th:text="${#temporals.format(post.createdAt, 'yyyy-MM-dd HH:mm')}"></p>
```

thymeleaf-extras-data

- Spring Data와 통합하여 `Page`, `Slice` 객체의 페이징 지원

  페이징 예시:

```
<a th:href="@{/list(page=${page.number + 1})}" th:if="${!page.last}">다음</a>
```

Standard Dialect 이해

기본적으로 타임리프는 `Standard Dialect`를 기반으로 동작

이 안에는 `th:text`, `th:each`, `th:if`, `th:object`, `th:field` 같은 모든 속성이 포함된다.

주요 처리기 구성

처리기	설명
`StandardTextTagProcessor`	`th:text` 해석
`StandardIfTagProcessor`	`th:if`
`StandardEachTagProcessor`	`th:each`
`SpringStandardDialect`	Spring의 확장 포함 (예: Spring EL 지원)

Dialect 직접 만들기

왜 만드는가?

- HTML DSL을 확장하고 싶을 때
  예: `th:money="item.price"` → 자동 ₩##,### 포맷
- 로직을 Java로 숨기고 싶을 때

핵심 구성요소

1. Dialect 클래스

```
public class MyDialect extends AbstractProcessorDialect {
    public MyDialect() {
        super("My Dialect", "my", 1000);
    }

    @Override
    public Set<IProcessor> getProcessors(...) {
        return Set.of(new MyMoneyProcessor(getPrefix()));
    }
}
```

2. 커스텀 Processor

```
public class MyMoneyProcessor extends AbstractAttributeTagProcessor {
    public MyMoneyProcessor(String prefix) {
        super(TemplateMode.HTML, prefix, "money", true, null, false, 1000);
    }

    @Override
    protected void doProcess(...) {
        BigDecimal value = (BigDecimal) expression.getValue(context);
        String formatted = "₩" + NumberFormat.getInstance().format(value);
        structureHandler.setBody(formatted, false);
    }
}
```

3. SpringTemplateEngine에 등록

```
@Bean
public SpringTemplateEngine templateEngine() {
    SpringTemplateEngine engine = new SpringTemplateEngine();
    engine.addDialect(new MyDialect());
    return engine;
}
```

사용 예시:

```
<span my:money="${product.price}"></span>
```

실전 프로젝트 적용 팁

항목	핵심 포인트
Form 바인딩	`th:object`, `th:field`, `th:errors` 반드시 세트

항목	핵심 포인트
템플릿 캐시	개발 시 `false`, 운영 시 `true`
정적 리소스	`@{/...}` 사용으로 경로 자동 처리
에러 페이지	`templates/error/*.html` 자동 처리
Global 예외 핸들링	`@ControllerAdvice + Model`로 커스터마이징 가능

폼 바인딩 실수 방지법

실수 1: `th:object` 없이 `th:field`만 사용

```
<!-- 잘못된 예 -->
<form>
  <input th:field="*{email}" />
</form>
```

- `th:object`가 없으면 `*{}` 구문이 동작하지 않음 → 오류 발생

  올바른 예:

  ```
  <form th:object="${userForm}">
    <input th:field="*{email}" />
  </form>
  ```

실수 2: 검증 메시지 누락

```
<input th:field="*{name}" />
<!-- ↓ 빠뜨리면 에러 메시지 출력 안 됨 -->
<div th:errors="*{name}">이름 오류</div>
```

- 모든 입력 필드마다 `th:errors`, `#fields.hasErrors(...)` 같이 사용하기

실수 3: name 속성 중복 또는 없음

타임리프는 `th:field`가 `name`, `id`, `value`를 자동 생성하지만, 직접 `name`을 추가하면 **충돌 발생** 가능.

- 팁: `th:field`만 쓰고, 수동 `name=""` 제거할 것.

템플릿 캐시 처리

개발 중에는 캐시 끄기

```
spring:
  thymeleaf:
    cache: false
```

- 변경할 때마다 자동 반영되므로 **개발 속도 향상**

운영 환경에서는 반드시 켜기

```yaml
spring:
  thymeleaf:
    cache: true
```

- 템플릿 파일이 **캐싱되어 성능 향상**

정적 리소스 (JS, CSS) 관리

위치 규칙 (Spring Boot 기본)

리소스	경로
JS, CSS, 이미지	`/static/`, `/public/`, `/resources/`, `/META-INF/resources/`

> 예: `src/main/resources/static/js/app.js`

링크 연결 시 `@{}` 사용

```html
<link th:href="@{/css/style.css}" rel="stylesheet" />
<script th:src="@{/js/app.js}"></script>
```

- `@{}`를 써야 **컨텍스트 루트** 자동 적용됨
- 직접 `/css/...` 쓰면 서브 디렉토리 배포 시 문제 발생 가능

버전 캐싱 무력화 (선택)

```html
<script th:src="@{/js/app.js}?v=202505"></script>
```

- JS/CSS 캐싱 문제 해결할 때 유용

에러 페이지 커스터마이징

Spring Boot의 기본 오류 페이지 경로

- `/templates/error/404.html`
- `/templates/error/500.html`
- `/templates/error/403.html` 등

> 파일만 만들어두면 Spring Boot가 자동 렌더링

공통 에러 페이지 구성 예

- `/templates/error/404.html`

```
<html layout:decorate="~{layout/base}">
  <section layout:fragment="content">
    <h2>페이지를 찾을 수 없습니다</h2>
    <a th:href="@{/}">홈으로 돌아가기</a>
  </section>
</html>
```

Global 예외 핸들러 + 뷰 반환

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public String userNotFound(Model model) {
        model.addAttribute("message", "사용자를 찾을 수 없습니다.");
        return "error/custom";
    }
}
```

- `templates/error/custom.html`

```
<p th:text="${message}">오류 메시지</p>
```

실전 프로젝트 예시 : "회원 게시판 시스템"

타임리프 기능 전체 + Spring MVC + Security + Validation + REST 통합까지 사용하는 실무형 구조

기능 요약

회원 가입/로그인/로그아웃

게시글 작성/조회/수정/삭제

댓글 기능

관리자 전용 기능

페이징, 검색, 유효성 검증

타임리프: fragment, th:each, th:if, messages, 유틸, 보안, layout 전부 사용

전체 페이지 구성 및 라우팅

`/` – 메인 페이지

`/user/signup` – 회원 가입

`/user/login` – 로그인

`/user/mypage` – 마이페이지

`/post/list` – 게시글 목록 (페이징, 검색 포함)

`/post/{id}` – 게시글 상세

`/post/create` – 게시글 작성

`/post/edit/{id}` – 게시글 수정

`/admin/user/list` – 관리자: 전체 회원 목록

타임리프 템플릿 구성

templates/layout

`base.html` – 전체 레이아웃

`th:fragment="layout(content)"`

header, footer, main 포함

`layout:decorate="~{layout/base}"`

templates/fragments

`header.html` – 로그인 상태 분기 (`sec:authorize`)

`footer.html` – 고정 영역

`alert.html` – flash 메시지 출력 fragment

`paging.html` – 페이징 처리 fragment

`sidebar.html` – 관리자 전용 메뉴

`msgBox(message, type)` – 공통 메시지 컴포넌트 (fragment with parameter)

타임리프 문법 적용 포인트

`th:object`, `th:field` → 모든 form 페이지 (`signup`, `create`, `edit`)

`th:each` → 목록 페이지 (게시글, 회원, 댓글 반복)

`th:if`, `th:unless` → 로그인 분기, 데이터 유무 분기

`th:switch`, `th:case` → 사용자 역할에 따라 버튼 구분

`th:with` → 지역 변수 선언 (예: `th:with="roleName=${#myUtil.role(user.role)}"`)

`th:replace`, `th:insert` → header, footer, alert 조각 삽입

`th:inline="javascript"` → 로그인한 사용자 정보 JS에 전달

`@{}` → URL 처리 (`th:href`, `th:src`, `th:action`)

`#{}` → 국제화 메시지 처리

`${#fields.hasErrors('field')}` → 유효성 검사 메시지 처리

폼 처리 구성

회원가입, 게시글 작성, 댓글 입력

`th:field`로 name/id/value 자동 설정

`BindingResult`로 에러 메시지 출력

`th:errors`, `#fields` 활용

보안 처리 (thymeleaf-extras-springsecurity6)

`sec:authorize="isAuthenticated()"`,

`hasRole('ADMIN')` 등으로 메뉴 제어

로그인 사용자 정보 출력: `sec:authentication="name"`

관리자 메뉴, 사용자 메뉴 분리

유틸리티 적용

`@Component("myUtil")` 등록

`role(String code)` → "관리자", "일반회원"

`formatDate(LocalDateTime)` → "yyyy-MM-dd HH:mm"

`limitLength(String, int)` → 글자 수 자르기

`${#myUtil.xxx(...)}` 형태로 호출

메시지 처리

resources/messages.properties

```
user.signup.title=회원가입
user.name.required=이름은 필수입니다.
post.create.success=게시글이 성공적으로 등록되었습니다.
```

- 모든 텍스트 `#{key}`로 처리
- 검증 메시지까지 완전 외부화

페이지네이션 처리

`th:each="page : ${pageList}"`

`th:href="@{/post/list(page=${i})}"`

현재 페이지 `th:classappend="${i == nowPage} ? 'active' : ''"`

JavaScript 연동 (서버 → JS 데이터 전달)

```
<script th:inline="javascript">
  const username = [[${#authentication.name}]];
  const postId = [[${post.id}]];
</script>
```

JSON 객체 전달

에러 처리 & 디버깅

예외 페이지: `/templates/error/404.html`, `500.html`, `403.html`

`th:if="${errorMessage}" th:text="${errorMessage}"`로 커스텀 메시지 처리

디버깅 시 `th:text="'[DEBUG] ID: ' + ${user.id}"` 사용

테스트 전략

`@WebMvcTest`로 컨트롤러 + 뷰 템플릿 테스트

`MockMvc` 사용해 HTML 렌더링 결과 비교

`containsString(...)`, `.model().attributeExists(...)` 등 사용