Tutorial: Using Thymeleaf

Document version	20131006-06 October 2013	
Project version	2.0.19	
Project web site	http://www.thymeleaf.org	
Translation info.	Translated by Kunner	
	v1.0	2013.12.09
	v1.1	2013.12.13

1.Thymeleaf 소개

1.1. Thymeleaf 란?

Thymeleaf는 일종의 자바 라이브러리로, 어플리케이션에 의해 생성된 Display Data나 Text를 표시하기 위해 템플릿 파일을 변환해 주는 XML/XHTML/HTML5 템플릿 엔진입니다. Thymeleaf는 XHTML/HTML5에 더 적합하지만, 웹이나 Standalone 어플리케이션에서 사용되는 XML 파일도 모두 처리할 수 있습니다. Thymeleaf의 주요 목표는 템플릿을 만드는 우아하고 올바른 방법을 제공하는 것입니다. 이를 위해 템플릿 파일에 명시적인 로직 코드 대신, DOM에 사전 정의된 XML 태그와 어트리뷰트를 바탕으로 코드를 작성합니다. 이러한 구조는 I/O 연산에 최소한의 자원을 사용할 수 있도록 하여 파싱된 파일의 지능형 캐싱을 통해 빠른 템플릿의 처리가 가능하도록 합니다.
마지막으로, Thymeleaf는 충분히 검증된 템플릿을 만들 수 있도록 처음부터 XML 및 웹표준을 준수하도록 고안되어 있습니다.

1.2. Thymeleaf이 처리할 수 있는 템플릿의 종류

Thymeleaf는 "템플릿 모드" 라고 불리는 다음 여섯 가지 템플릿을 바로 사용할 수 있습니다.

- XML
- Valid XML
- XHTML
- Valid XHTML
- HTML5
- Legacy HTML5

Legacy HTML5를 제외한 모든 모드는 well-formed XML 파일을 참조하기 때문에, HTML5를 연산할 때 다중 어트리뷰트에서 어트리뷰트 값 사이에 ",(콤마)"가 없거나, 어트리뷰트 값이 없거나, 닫힘 태그 없는 독립 태그와 같은 기능을 이용할 수 있습니다. 이러한 특정 모드의 파일을 처리하기 위해, Thymeleaf 는 완벽한 HTML5 문법으로 작성된 파일도 일단 well-formed XML 파일로 변환합니다.

또 Thymeleaf의 유효성 검사는 오직 XML과 XHTML 템플릿에서만 사용 가능합니다.

그러나 Thymeleaf가 처리할 수 있는 템플릿에 이런 종류들만 있는 것은 아닙니다. 사용자는 템플릿을 해석하고 결과를 처리하는 방법을 지정하는 방식으로, 언제든 자신만의 템플릿 모드를 정의할 수 있습니다. 템플릿 모드를 정의하면, DOM 트리(XML이든 아니든) 형태로 모델링 되어 Thymeleaf 템플릿과 같이 효과적으로 처리 될 수 있습니다.

1.3. Dialects: Standard Dialects

Thymeleaf는 고도의 확장성을 가진 템플릿 엔진(=실은 프레임워크라고 불리는 게 맞습니다)으로 사용자의 템플릿에서 처리될 DOM 노드들과 이 노드들의 처리 방식을 모두 정의할 수 있습니다.

로직을 DOM 노드에 적용하는 Object를 "Processor(프로세서)" 라고 부르며, 프로세서(와 몇 가지 부산물)의 집합을 "Dialects(방언)" 이라 부르는데, Thymeleaf의 Core Library는 Standard Dialects(표준 방언)을 OOTB(즉시 사용 가능한 완제품)로 제공합니다. 이 Standard Dialects들은 사용자들에게 꽤 유용할 것입니다.

본 자습서에서는 Standard Dialects를 다루게 됩니다. 앞으로 다루게 될 모든 어트리뷰트와 구문 기능은 명시적으로 언급되지 않은 경우에도 이 Standard Dialects에 의해 정의 됩니다.

물론, 라이브러리의 고급 기능을 사용하면서 각자 필요한 처리 로직을 정의하려는 경우 커스텀 Dialects를 직접 만들 수도 있고, Standard dialects를 확장할 수도 있습니다. 템플릿 엔진은 한 번에 여러 Dialects를 구성할 수도 있습니다.

> 오피셜 thymeleaf-spring3 통합 패키지는 "SpringStandard Dialects"라고 불리는 dialect 를 정의하고 있습니다. 이 Dialects 는 Standard Dialects 와 거의 유사하지만, Spring Framework 에서 보다 편리하게 사용할 수 있도록 몇 가지 기능이 적용되었습니다(예를 들어, SpEL 을 사용하는 대신 Thymeleaf's 표준 OGNL 을 사용). 따라서 만약 당신이 Spring MVC 의 사용자이고, 시간을 낭비하고 싶지 않다면 여기서 배운 것들을 거의 모든 어플리케이션에 적용하게 될 것입니다.

Thymeleaf의 Standard Dialect는 어떤 모드의 템플릿이든 처리할 수 있으나, 특히 웹에 특화된 템플릿 모드(XHTML 또는 HTML5)에 적합합니다. 또한 HTML5는 다음과 같은 XHTML의 규격을 적절히 지원합니다.

- XHTML 1.0 Transitional
- XHTML 1.0 Strict
- XHTML 1.0 Frameset
- XTHML 1.1

또한, Standard Dialect는 태그들에 적용될 어떤 프로세서도 포함하고 있지 않은 *attribute-only* Dialect 입니다. 이러한 특성은 XHTML/HTML5 템플릿 파일을 처리하지 않더라도 브라우저에 적절히 보여 질 수 있도록 하는데, 브라우저가 이러한 추가적인 어트리뷰트를 그저 무시해 버리기 때문입니다. 예를 들어 JSP 태그 라이브러리를 이용하게 되면 아래처럼 브라우저가 바로 표시할 수 없는 코드들이 포함되게 됩니다.

```
<form:inputText name="userName" value="${user.name}" />
```

Thymeleaf의 Standard Dialect는 동일한 기능을 수행하기 위한 다음과 같은 방법을 제공합니다.

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

이 코드의 결과는 브라우저에 제대로 표시될 뿐 아니라, 프로토타입을 브라우저에서 바로 열었을 때는 특정 값이 보여지고, 서버의 Thymeleaf 엔진에 의해 템플릿이 처리 될 때는 결과값을 태그의 Value 값에 대체하도록 할 수 있습니다. 이 예에서는 value의 어트리뷰트 값 'James Carrot '이 서버의 user.name 연산 결과에 따라 대체 됩니다. 이러한 기능은 디자이너와 개발자가 매우 유사한 템플릿 파일로 협업할 수 있도록 해 주기 때문에, 프로토타입(퍼블리싱된 HTML)을 서버에서 동작 가능한 템플릿 파일로 변경하는 데 드는 수고를 줄여 줄 수 있습니다. 이 기능이 바로 "Natural Templating" 입니다.

1.4. 전체적인 아키텍처

Thymeleaf의 핵심(Core)은 DOM 처리 엔진입니다. 특히, Thymeleaf는 템플릿 표현 트리를 메모리에

상주시키기 위해 표준 DOM API가 아닌, 고유의 고성능 DOM 을 이용합니다. 이 DOM은 노드를 탐색하고 프로세서가 처리될 때 Context에 기술된 설정과 데이터셋에 따라 템플릿을 표시하도록 DOM을 수정합니다.

많은 경우 웹문서는 오브젝트 트리 구조로 표현되기 때문에 DOM 템플릿 표현을 사용하는 것은 웹 어플리케이션에 매우 적합합니다(브라우저가 웹 페이지를 메모리에 올리기 위해 DOM 트리를 이용합니다). 또한 대부분의 웹 어플리케이션은 그다지 많지 않은 수의 템플릿을 이용하는데, 각 템플릿 파일의 크기가 크지 않으며 어플리케이션이 실행될 때는 거의 수정되지 않는다는 점을 떠올려 보면 Thymeleaf는 실행 환경에서 보다 빠른 동작 속도를 보여 줄 것입니다. 템플릿을 파싱하는 DOM 트리를 메모리에 상주시키면, 템플릿을 표현할 때 필요에 따라 극히 적은 I/O 처리를 하게 되기 때문입니다.

> 이에 대해 보다 자세한 내용을 알고 싶다면, 자습서에서 dedicate 캐싱 및 메모리와 리소스 사용을 최적화 하는 방법에 대해 살펴 보시기 바랍니다.

그렇지만 여기에는 다음과 같은 제한이 있습니다.

이 아키텍처는 다른 템플릿 엔진에 비해 보다 큰 메모리 할당을 필요로 하기 때문에, 대용량 XML 문서를 생성하는 라이브러리를 사용해서는 안 됩니다. JVM 메모리 사이즈에 따라 다르겠지만, 만약 단일 템플릿이 10메가 이상의 XML 파일로 이뤄진 경우에는 Thymeleaf를 이용해서는 안 됩니다.

> 이러한 제한을 XHTML/HTML5 가 아닌 XML data 파일에만 적용한 이유는, 절대 사용자의 브라우저에 지나친 부하를 줄 만큼 큰 용량으로 만들어서는 안 되기 때문입니다. 브라우저들은 페이지를 표시할 때 늘 DOM 트리를 만든다는 것을 기억하시기 바랍니다.

1.5. 필독 참고 사항

Thymeleaf는 웹 어플리케이션을 개발할 때 특히 적합합니다. 웹 어플리케이션은 몇 가지 표준에 기초하고 있습니다(모두가 잘 알아야 하지만, 수년 째 이 분야에 종사하고 있는 사람들조차 잘 모르는).

HTML5의 등장으로 웹표준을 준수하는 일은 그 어느 때 보다 어려워 졌습니다. - *XHTML이나 HTML로 회귀하고 있는 걸까요? XML 구문을 포기해야 할까요? 왜 더 이상 누구도 XHTML 2.0에 대해 이야기 하지 않는 걸까요?*

튜토리얼을 진행하기 앞서, Thymeleaf 사이트에서 다음 기사를 읽어 보시길 강력히 권합니다.

"From HTML to HTML (via HTML)" http://www.thymeleaf.org/fromhtmltohtmlviahtml.html

2.The Good Thymes Virtual Grocery

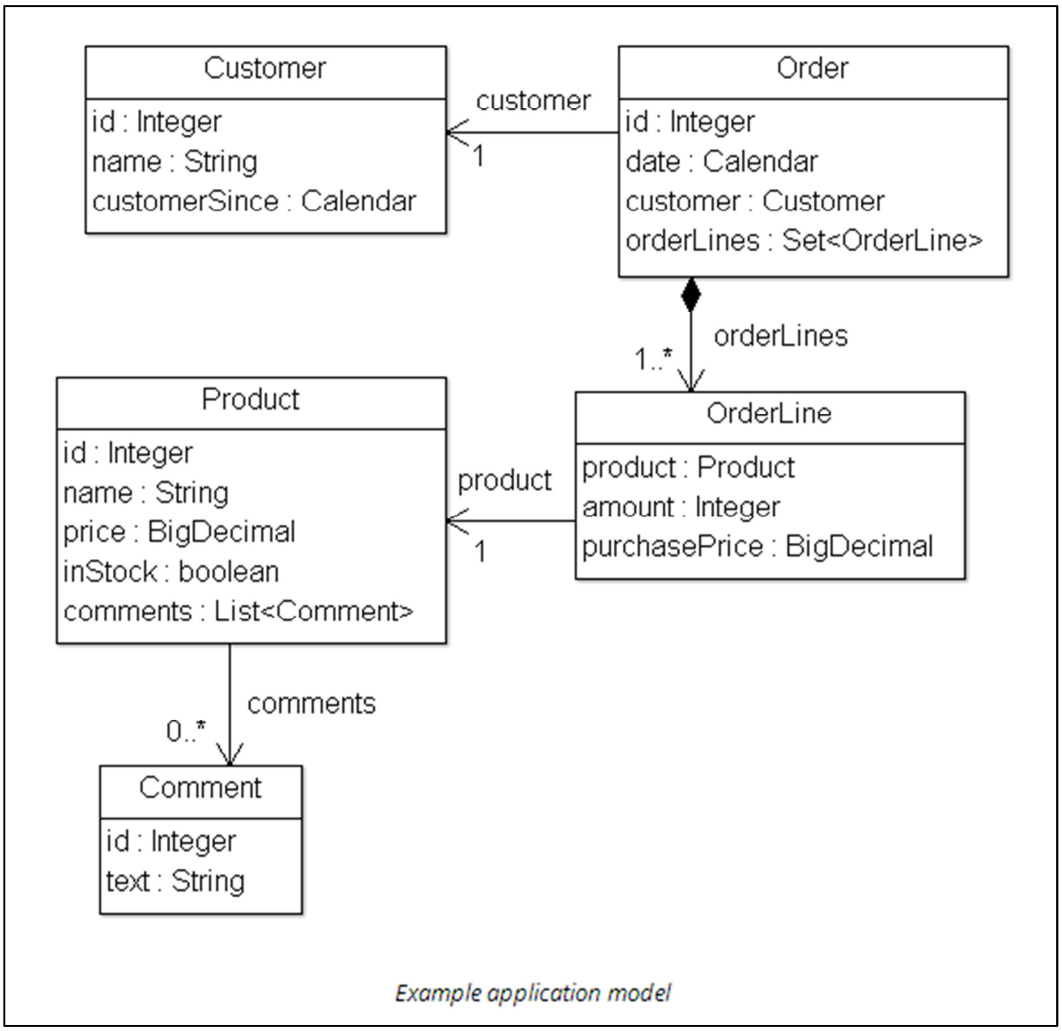
2.1. 식료품 판매용 웹사이트

Thymeleaf에서 템플릿을 처리하는 개념을 보다 쉽게 설명하기 위해, 본 튜토리얼에서는 데모 어플리케이션을 이용하겠습니다. 이 어플리케이션은 Thymeleaf 웹사이트에서 다운로드 할 수

있습니다.

이 어플리케이션은 가상의 식료품 판매 웹사이트로, Thymeleaf의 다양한 기능을 설명하기 위해 적절한 시나리오를 제공합니다.

이 어플리케이션에는 매우 심플한 엔티티 모델 셋이 필요합니다. *상품Product*은 *고객Customer*의 *주문Order*에 의해 판매 됩니다. 또, 각 *상품Product*들에 달린 *상품평Comment*을 관리할 것입니다.



Example application model

이 작은 어플리케이션은 다음과 같은 메소드를 포함한 Service 오브젝트에 의해 간단한 서비스 레이어가 구성됩니다.

```
public class ProductService {

    ...

    public List<Product> findAll() {
        return ProductRepository.getInstance().findAll();
    }

    public Product findById(Integer id) {
        return ProductRepository.getInstance().findById(id);
    }

}
```

마지막으로 어플리케이션의 웹 레이어는 요청받은 URL에 따라 Thymeleaf에서 사용 가능한 명령어들의 처리를 위임하는 필터를 가지게 될 것입니다.

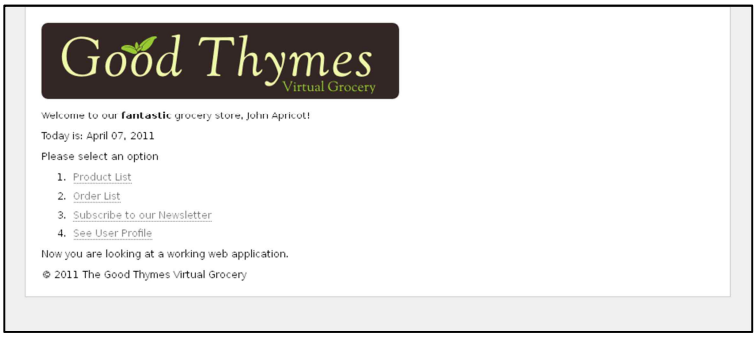
```
private boolean process(HttpServletRequest request, HttpServletResponse response)
        throws ServletException {

    try {

        /*
         * Query controller/URL mapping and obtain the controller
         * that will process the request. If no controller is available,
         * return false and let other filters/servlets process the request.
         */
        IGTVGController controller =
GTVGApplication.resolveControllerForRequest(request);
        if (controller == null) {
            return false;
        }
        /*
         * Obtain the TemplateEngine instance.
         */
        TemplateEngine templateEngine = GTVGApplication.getTemplateEngine();

        /*
         * Write the response headers
         */
        response.setContentType("text/html;charset=UTF-8");
        response.setHeader("Pragma", "no-cache");
        response.setHeader("Cache-Control", "no-cache");
        response.setDateHeader("Expires", 0);

        /*
         * Execute the controller and process view template,
         * writing the results to the response writer.
         */
        controller.process(
                request, response, this.servletContext, templateEngine);

        return true;

    } catch (Exception e) {
        throw new ServletException(e);
    }

}
```

다음은 인터페이스 입니다.

```
public interface IGTVGController {


    public String process(

            HttpServletRequest request, HttpServletResponse response,

            ServletContext servletContext, TemplateEngine templateEngine);


}
```

이제 해야 할 일은 서비스에서 데이터를 검색하고, TemplateEngine 오브젝트를 이용해 템플릿을 처리하기 위해 IGTVGController 인터페이스의 implementation을 생성하는 것입니다.

결국, 이것은 다음과 같이 보여질 것입니다.



그럼 먼저 템플릿 엔진을 초기화 하는 방법에 대해 알아 보기로 하겠습니다.

2.2. 템플릿 엔진의 생성과 구성

필터의 process(...) 메소드는 다음과 같은 코드를 포함하고 있습니다.

```
TemplateEngine templateEngine = GTVGApplication.getTemplateEngine();
```

GTVGApplication 클래스는 어플리케이션에서 Thymeleaf를 사용 가능하게 하는 가장 중요한 오브젝트들 중 하나인 TemplateEngine 인스턴스의 생성과 구성을 담당하고 있습니다.

org.thymeleaf.TemplateEngine 오브젝트는 다음과 같이 초기화 됩니다.

```
public class GTVGApplication {


    ...
    private static TemplateEngine templateEngine;
    ...


    static {
        ...
        initializeTemplateEngine();
        ...
    }


    private static void initializeTemplateEngine() {

        ServletContextTemplateResolver templateResolver = new
ServletContextTemplateResolver();
        // XHTML is the default mode, but we will set it anyway for better
understanding of code
        templateResolver.setTemplateMode("XHTML");
        // This will convert "home" to "/WEB-INF/templates/home.html"
        templateResolver.setPrefix("/WEB-INF/templates/");
        templateResolver.setSuffix(".html");
        // Set template cache TTL to 1 hour. If not set, entries would live in cache
until expelled by LRU
        templateResolver.setCacheTTLMs(3600000L);

        templateEngine = new TemplateEngine();
        templateEngine.setTemplateResolver(templateResolver);

    }

    ...

}
```

물론 TemplateEngine 오브젝트를 구성하는 다양한 방법이 있지만, 이 몇 줄의 코드는 이번 단계에서

필요한 내용을 알려 주기에 충분합니다.

2.2.1. The Template Resolver

이제 템플릿 리졸버에 대해 알아 보겠습니다.

```
ServletContextTemplateResolver templateResolver = new ServletContextTemplateResolver();
```

템플릿 리졸버는 Thymeleaf API에서 호출되는 인터페이스의 implement 오브젝트 입니다.

```
public interface ITemplateResolver {

    ...

    /*
     * Templates are resolved by String name
(templateProcessingParameters.getTemplateName())
     * Will return null if template cannot be handled by this template resolver.
     */
    public TemplateResolution resolveTemplate(TemplateProcessingParameters
templateProcessingParameters);

}
```

이 오브젝트들은 템플릿에 접근하는 방법을 결정하는 역할을 담당하며, GTVG 어플리케이션(본 예제)에서 템플릿 파일과 같은 서블릿 컨텍스트의 리소스를 검색할 때 사용하는 org.thymeleaf.templateresolver.ServletContextTemplateResolver implementation 입니다. 참고로 javax.servlet.ServletContext 오브젝트는 모든 자바 웹 어플리케이션에 존재하며 각 웹 어플리케이션의 리소스 루트 패스를 고려하여 리소스를 처리하는 객체 입니다.

추가로, 템플릿 리졸버에 몇 가지 설정을 할 수 있습니다.

먼저 템플릿 모드 입니다. 표준 템플릿 모드 중 하나를 예로 들어 보겠습니다.

```
templateResolver.setTemplateMode("XHTML");
```

XHTML은 ServletContextTemplateResolver의 기본적인 템플릿 모드지만, 코드를 작성할 때는 어떤 일이 벌어지고 있는지 명확하게 알 수 있도록 하는 것이 좋습니다.

```
templateResolver.setPrefix("/WEB-INF/templates/");
templateResolver.setSuffix(".html");
```

이 prefix와 surfix는 템플릿의 이름을 수정하여 템플릿에 사용될 실제 리소스의 이름을 템플릿 엔진에 전달하게 됩니다.

이러한 설정을 이용하면, "product/list" 라는 템플릿 이름은 다음 코드처럼 동작하게 될 것입니다.

```
servletContext.getResourceAsStream("/WEB-INF/templates/product/list.html")
```

추가적으로 템플릿 리졸버에 cacheTTMLs 어트리뷰트를 지정하여, 캐시의 유효 기간을 지정할 수 있습니다.

```
templateResolver.setCacheTTLMs(3600000L);
```

물론 템플릿은 설정된 TTL 전에라도, 캐시의 최대 크기를 초과하면 캐시에서 가장 오래 된 항목을 제거할 수 있습니다.

> 캐시의 동작과 용량을 정의하기 위해서는, 사용자에 의해 ICacheManager 인터페이스를 구현하거나, 단순히 StandardCacheManager 오브젝트에 설정된 기본 캐시 정보를 수정하면 됩니다.

템플릿 리졸버에 대해서는 나중에 자세히 다루겠습니다. 이제 템플릿 리졸버 오브젝트의 생성에 대해 알아 보겠습니다.

2.2.2. The Template Engine

org.thymeleaf.TemplateEngine 클래스 중 하나인 템플릿 엔진 오브젝트는 다음과 같은 코드로 생성됩니다.

```
templateEngine = new TemplateEngine();
templateEngine.setTemplateResolver(templateResolver);
```

아주 간단하지 않습니까? 이제 남은 것은 인스턴스를 생성하고 템플릿 리졸버를 할당하는 것뿐입니다.

템플릿 리졸버는 TemplateEngine 에 유일한 필수 파라미터입니다. 그 외에도 message resolver나 cache size와 같은 여러 파라미터들이 있긴 하지만, 이에 대해서는 나중에 알아 보도록 하겠습니다. 지금 단계에서 필요한 것은 이것뿐입니다.

템플릿 엔진이 준비 되었으니, 이제 Thymeleaf를 이용해 페이지를 만들어 보도록 하겠습니다.

3.TEXT 이용하기

3.1. 다국어로 첫 화면 구성하기

첫 번째 작업은 식료품 웹사이트의 첫 화면을 만드는 일입니다.

이 페이지의 최초 코드는 매우 심플합니다. 그저 제목과 인사말이 있을 뿐입니다.

다음은 /WEB-INF/templates/home.html 파일의 코드 입니다.

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-3.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all" href="../../css/gtvg.css"
th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <p th:text="#{home.welcome}">Welcome to our grocery store!</p>

  </body>

</html>
```

먼저 여기서 주목할 것은 이 파일이 XHTML이기 때문에 어떤 브라우저에서든 바로 표현된다는 것입니다. 이 소스는 어떤 non-XHTML 코드도 포함하고 있지 않으며, 브라우저들은 자신들이 이해하지 못하는 th:text 와 같은 어트리뷰트들은 무시해 버리게 됩니다. 또 well-formed DOCTYPE을 선언했기 때문에 브라우저는 이 페이지를 표준에 맞게 표시하게 됩니다.

다음으로, 이 소스는 또한 검증된 XHTML 코드 입니다. th:text와 같은 어트리뷰트를 정의한 Thymeleaf DTD를 선언했기 때문에 템플릿은 검증된 XHTML로 간주 됩니다. 게다가 일단 템플릿이 실행되면 모든 th:* 어트리뷰트가 제거되고 Thymeleaf가 자동으로 DOCTYPE의 DTD 선언절을 XHTML 1.0 Strict 로 변경할 것입니다. (나중에 DTD 변환 기능에 대해 알아 볼 것입니다)

Thymeleaf 네임스페이스는 th:* 어트리뷰트에 대해 선언하고 있습니다.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
```

그동안 템플릿의 유효성 및 well-formed 적용 여부에 대해 전혀 신경 쓰지 않았다면, 단순히 xmlns 네임스페이스 선언에서 XHTML 1.0 Strict DOCTYPE 표준을 지정 할 수도 있습니다.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html>

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all" href="../../css/gtvg.css"
th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <p th:text="#{home.welcome}">Welcome to our grocery store!</p>

  </body>

</html>
```

또한 이 코드는 Thymeleaf의 XHTML 모드에서 완벽하게 작동합니다. 비록 IDE 에 끔찍한 경고 메시지가 가득하겠지만, 유효성은 충분히 지키고 있습니다.

이제 템플릿의 매우 흥미로운 부분인 th:text 어트리뷰트에 대해 살펴 보겠습니다.

3.1.1. th:text 사용하기와 텍스트 외부화

'텍스트 외부화'란 여러 템플릿 파일들에서 템플릿 코드 조각을 추출하여 특정 파일을 별도로 보관하여 다른 언어 적용 시 쉽게 문자열을 대체할 수 있도록 하는 것을 의미합니다. 텍스트 외부화의 대표적인 예로 .properties 파일을 들 수 있습니다. 텍스트의 조각들을 외부화 하는 것을 보통 "메시지"라고 표현합니다.

메시지는 항상 각 메시지를 인식할 수 있는 키를 가지고 있는데, Thymeleaf에서는 #{...} 구문을 이용하여 텍스트가 특정 메시지와 대응하도록 합니다.

```
<p th:text="#{home.welcome}">Welcome to our grocery store!</p>
```

여기서 우리는 Thymeleaf Standard Dialect의 두 가지 기능을 발견할 수 있습니다.

- th:text 어트리뷰트에 지정된 value는 여기에 사용된 표현식의 실행 결과에 따라 "Welcome to our grocery store!"의 내용을 대체하게 됩니다.
- 표준 표현식(*Standard Expression Syntax*)에 의해 규정된 #{home.welcome} 표현식은 th:text 어트리뷰트에 사용될 메시지를 지정하며, 어떤 템플릿을 이용하든 해당 메시지는 home.welcome의 key 값에 대응하는 텍스트로 지정됩니다.

그렇다면 외부화된 텍스트는 어디에 위치할까요?

Thymeleaf는 외부화된 텍스트의 위치를 org.thymeleaf.messageresolver.IMessageResolver 구현체에 지정된 내용에 따라 설정할 수 있습니다. 보통 .properties 파일을 사용하는 것을 기초로 구현되어 있으나 원하는 경우 DB에서 메시지를 불러 오는 방식으로도 구현할 수 있습니다.

템플릿 엔진이 초기화 될 때 메시지 리졸버를 지정하지 않으면 org.thymeleaf.messageresolver.StandardMessageResolver 클래스에 구현된 Standard Message Resolver를 이용하게 됩니다.

이 Standard Message Resolver는 /WEB-INF/templates/home.html 이 위치한 폴더 내에 템플릿과 같은 이름을 가진 .properties 파일이 있다고 가정하여 메시지를 찾게 됩니다.

- /WEB-INF/templates/home_en.properties for English texts.
- /WEB-INF/templates/home_es.properties for Spanish language texts.
- /WEB-INF/templates/home_pt_BR.properties for Portuguese (Brazil) language texts.
- /WEB-INF/templates/home.properties for default texts (if locale is not matched).

다음은 home_es.properties 파일의 내용입니다.

```
home.welcome=¡Bienvenido a nuestra tienda de comestibles!
```

Thymeleaf이 템플릿을 처리하게 하는데 필요한 것은 모두 준비 되었습니다.

이제 Home Controller 를 만들어 보겠습니다.

3.1.2. Contexts

템플릿 처리 순서에 따라, HomeController 클래스로 앞서 봤던 IGTVController 인터페이스의 구현체를 만들어 보겠습니다.

```
public class HomeController implements IGTVGController {

    public void process(
            HttpServletRequest request, HttpServletResponse response,
            ServletContext servletContext, TemplateEngine templateEngine) {

        WebContext ctx = new WebContext(request, response, servletContext,
request.getLocale());
        templateEngine.process("home", ctx, response.getWriter());

    }

}
```

먼저 context를 만들어야 합니다. Thymeleaf의 Context는 org.thymeleaf.context.IContext 인터페이스를 구현한 객체 입니다. Contexts는 템플릿 엔진의 실행에 필요한 모든 데이터를 포함하고

있어야 합니다. 또한 Contexts는 외부화된 텍스트에 사용될 Locale 정보에 영향을 받게 됩니다.

```
public interface IContext {

    public VariablesMap<String,Object> getVariables();
    public Locale getLocale();
    ...

}
```

또 다음처럼 org.thymeleaf.context.IWebContext 인터페이스를 확장해 줍니다.

```
public interface IWebContext extends IContext {

    public HttpSerlvetRequest getHttpServletRequest();
    public HttpSession getHttpSession();
    public ServletContext getServletContext();

    public VariablesMap<String,String[]> getRequestParameters();
    public VariablesMap<String,Object> getRequestAttributes();
    public VariablesMap<String,Object> getSessionAttributes();
    public VariablesMap<String,Object> getApplicationAttributes();

}
```

Thymeleaf의 코어 라이브러리는 다음과 같은 인터페이스를 구현합니다.

- org.thymeleaf.context.Context implements IContext
- org.thymeleaf.context.WebContext implements IWebContext

Controller 코드에서 볼 수 있는 것처럼, WebContext는 우리가 사용하게 될 Contexts입니다.
실제로는 ServletContextTemplateResolver를 사용하기 위해서는 IWebContext를 구현하는 context를 사용해야 합니다.

```
WebContext ctx = new WebContext(request, servletContext, request.getLocale());
```

생성자의 세 인수 중 두개만 필수입니다. Locale은 지정하지 않는 경우 시스템의 기본 Locale 정보를 이용하기 때문입니다. (그렇지만 실제 어플리케이션을 만들 때는 절대 이렇게 하지 마세요)

인터페이스 정의에 따라, WebContext는 request 파라미터와 request, session, 어플리케이션의 어트리뷰트들을 가진 특수한 메소드를 제공합니다. 그러나 사실 WebContext는 이보다 조금 더 많은 기능을 가지고 있습니다.

- request 어트리뷰트를 Context 변수 맵에 추가하기
- 모든 request 파라미터를 param 으로 Context 변수에 추가하기
- 모든 세션 어트리뷰트를 session 으로 Context 변수에 추가하기
- 모든 서블릿 컨텍스트 어트리뷰트를 application 으로 Context 변수에 추가하기

Context와 WebContext를 포함한 모든 Context 객체는 IContext의 구현체로, 실행되기 직전 실행정보(execInfo)라고 불리는 특별한 변수들을 설정합니다. 이 변수에는 템플릿에서 사용될 수 있는 두 가지 데이터를 포함하고 있습니다.

- 템플릿 이름: 템플릿 엔진이 실행되기 전 ${execInfo.templateName}에 대응하는 템플릿을 조회

- 현재 일시(날짜/시간): 템플릿 엔진이 템플릿을 실행하기 시작한 시점에 해당하는 Calendar 객체

3.1.3. 템플릿 엔진의 실행

Context가 준비 되었으니, 템플릿 이름과 Context를 지정하여 템플릿을 실행하고 response writer에 전달하도록 다음과 같이 response를 지정해 보겠습니다.

```
templateEngine.process("home", ctx, response.getWriter());
```

스페인어 Locale을 이용한 결과는 다음과 같습니다.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type"/>
    <link rel="stylesheet" type="text/css" media="all" href="/gtvg/css/gtvg.css" />
  </head>

  <body>

    <p>¡Bienvenido a nuestra tienda de comestibles!</p>

  </body>

</html>
```

3.2. 텍스트와 변수에 대한 추가 정보

3.2.1. 언이스케이프 문자(Unescaped Text)

아주 단순한 형태의 홈페이지가 준비되었지만, 아직 충분하지는 않습니다. 다음과 같은 메시지가 있다고 가정해 볼까요?

```
home.welcome=Welcome to our <b>fantastic</b> grocery store!
```

이 메시지를 템플릿에 적용하면 다음과 같은 결과를 얻게 될 것입니다.

```
<p>Welcome to our &lt;b&gt;fantastic&lt;/b&gt; grocery store!</p>
```

우리가 기대한 결과와는 다른데, 그 이유는 <b> 태그가 escape 처리 되어 브라우저에 표시되기 때문입니다.

이는 th:text 어트리뷰트의 기본적인 동작 방식입니다. 만약 Thymeleaf가 XHTML 태그를 escape 처리하지 않기를 원한다면, th:utext 어트리뷰트를 사용하면 됩니다. (Unescaped text)

3.2.2. 변수 사용 및 출력

이제 홈페이지에 몇 가지 내용을 추가해 보겠습니다. 예를 들어 다음과 같이 인사말 아래에 날짜를 표시하고 싶다고 가정해 보겠습니다.

```
Welcome to our fantastic grocery store!

Today is: 12 july 2010
```

먼저 날짜 정보를 Context 변수에 추가 하기 위해 Controller를 수정해야 합니다.

```
public void process(
        HttpServletRequest request, HttpServletResponse response,
        ServletContext servletContext, TemplateEngine templateEngine) {

    SimpleDateFormat dateFormat = new SimpleDateFormat("dd MMMM yyyy");
    Calendar cal = Calendar.getInstance();

    WebContext ctx = new WebContext(request, servletContext, request.getLocale());
    ctx.setVariable("today", dateFormat.format(cal.getTime()));

    templateEngine.process("home", ctx, response.getWriter());

}
```

Context에 String 변수로 today를 추가했으니, 이제 템플릿에 표시할 수 있습니다.

```
<body>

  <p th:utext="#{home.welcome}">Welcome to our grocery store!</p>

  <p>Today is: <span th:text="${today}">13 February 2011</span></p>

</body>
```

보시는 것처럼, 태그 body의 내용을 대체하기 위해 새로운 코드도 th:text 어트리뷰트를 이용하고 있지만 #{...} 대신 ${...} 구문을 사용하는 약간의 차이가 있습니다. 이것은 변수 값을 표현하는 방식으로, OGNL(*Object-Graph Navigation Language*)의 표현 방식에 따라 Context 변수 맵에 의해 실행됩니다.

${today} 는 "today 라는 변수의 값을 구하라"라는 뜻입니다. 그리고 이런 표현은 $user.name 처럼 보다 복잡해 질 수 있는데, 이는 "user 객체를 구하고, 이 객체의 getName() 메소드를 호출하라"는 의미 입니다.

메시지, 표현식 이외에도 어트리뷰트 값을 사용하는 수많은 방법이 있습니다. 다음 장에서 이러한 모든 방법들을 알아 보도록 하겠습니다.

4. Standard Expression Syntax(표준 표현 구문)

식료품 가상 스토어를 만드는 걸 잠시 중단하고, "Thymeleaf Standard Expression Syntax"라는 Thymeleaf Standard Dialect의 가장 중요한 내용 중 하나에 대해 알아 보도록 하겠습니다.

앞서 우리는 메시지와 변수를 출력하는 두 가지 유효한 방법을 알아 보았습니다.

```
<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>

<p>Today is: <span th:text="${today}">13 February 2011</span></p>
```

그러나 우리가 아직 알지 못 하는 더 흥미로운 방법들이 있습니다.

다음은 Standard Expression에서 제공하는 기능에 대해 요약한 것입니다.

- 단순 표현식
  - 변수 출력: ${...}
  - 선택 변수 출력: *{...}

- 메시지 출력: #{...}
- URL 링크 출력: @{...}

- 리터럴
  - 문자열 리터럴: '' 사이에 텍스트 삽입
  - 넘버 리터럴: 1,2,3,4,...
  - 불린 리터럴: true, false
  - 리터럴 토큰: text
- 텍스트 연산
  - 문자열 연결: +
  - 리터럴 대체: |the name is ${name}|
- 산술 연산
  - 이항 연산자: +, -, *, /, %
  - 마이너스 기호(단항 연산자): -
- 불린 연산
  - 이항 연산자: and, or
  - 부울 부정(단항 연산자): !, not
- 비교와 항등
  - 비교: >, <, >=, <= (gt, lt, ge, le)
  - 항등 연산자: ==, != (eq, ne)
- 조건 연산
  - if-then: (if) ? (then)
  - if-then-else: (if) ? (then) : (else)
  - Default: (value) ?: (defaultvalue)

각 기능들은 모두 중첩으로 결합해 사용할 수 있습니다.

```
'User is of type ' + (${user.isAdmin()} ? 'Administrator' : (${user.type} ?:
'Unknown'))
```

4.1. 메시지

이미 알고 있는 것처럼, #{...} 메시지 표현을 이용하면 다음과 같이 링크를 걸 수 있습니다.

```
<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>
```

그러나, 여기에는 우리가 아직 예측하지 못 한 것이 있습니다. 메시지 텍스트가 고정되어 있지 않다면? 예를 들어 어플리케이션에 현재 접속한 사용자가 누구인지 확인하고 사용자의 이름으로 인사말을 건네고 싶다면 어떻게 되겠습니까?

```
<p>¡Bienvenido a nuestra tienda de comestibles, John Apricot!</p>
```

이를 위해서는 다음과 같이 메시지에 파라미터를 추가해야 합니다.

```
home.welcome=¡Bienvenido a nuestra tienda de comestibles, {0}!
```

파라미터는 java.text.MessageFormat의 표준 구문으로 지정되므로, MessageFormat 클래스의 API docs에 기술되어 있는 숫자나 날짜 포맷을 추가 할 수도 있습니다.

다음은 HTTP 세션에서 user라는 이름으로 파라미터의 값을 받아 오게 되는 경우를 예로 든 것입니다.

```
<p th:utext="#{home.welcome(${session.user.name})}">Welcome to our grocery store,
Sebastian Pepper!</p>
```

필요하다면, 콤마로 구분된 몇 가지 파라미터를 표시할 수도 있습니다. 이렇게 되면 메시지의 키 자체도 변수로 처리할 수 있습니다.

```
<p th:utext="#{${welcomeMsgKey}(${session.user.name})}">Welcome to our grocery store,
Sebastian Pepper!</p>
```

4.2. 변수

앞서 ${...} 표현식은 OGNL 표현식에 따라 context에 포함된 변수의 맵을 처리한다고 언급한 바 있습니다.

> ONGL 에 대해 더 자세한 정보는 에서 http://commons.apache.org/ognl/ 에서 찾아 볼 수 있습니다.

ONGL 구문으로 다음과 같이 표현할 수 있다는 것을 알고 있습니다.

```
<p>Today is: <span th:text="${today}">13 february 2011</span>.</p>
```

그리고 이 표현식은 다음의 실행 결과에 해당합니다.

```
ctx.getVariables().get("today");
```

하지만 OGNL은 다음과 같이 아주 강력한 표현식도 사용할 수 있게 해 줍니다.

```
<p th:utext="#{home.welcome(${session.user.name})}">Welcome to our grocery store,
Sebastian Pepper!</p>
```

이 표현식을 실행하여 아래와 같이 user.name을 구할 수 있습니다.

```
((User) ctx.getVariables().get("session").get("user")).getName();
```

getter 메소드 탐색은 OGNL이 제공하는 기능 중 하나일 뿐입니다. 아래 내용을 참고하세요.

```
/*
 * Access to properties using the point (.). Equivalent to calling property getters.
 */
${person.father.name}

/*
 * Access to properties can also be made by using brackets ([]) and writing
 * the name of the property as a variable or between single quotes.
```

```
 */
${person['father']['name']}

/*
 * If the object is a map, both dot and bracket syntax will be equivalent to
 * executing a call on its get(...) method.
 */
${countriesByCode.ES}
${personsByName['Stephen Zucchini'].age}

/*
 * Indexed access to arrays or collections is also performed with brackets,
 * writing the index without quotes.
 */
${personsArray[0].name}

/*
 * Methods can be called, even with arguments.
 */
${person.createCompleteName()}
${person.createCompleteNameWithSeparator('-')}
```

4.2.1. 기본 객체 출력

Context 변수에 OGNL 표현식을 대체할 때, 일부 객체는 더 높은 유연성을 위한 표현식을 사용할 수 있게 만들어져 있습니다. # 기호로 시작되는 다음 객체들은 각각 OGNL 표준을 따르게 됩니다.

기본 객체	설명
#ctx	the context object
#vars	the context variables
#locale	the context locale
#httpServletRequest	(only in Web Contexts) the HttpServletRequest object
#httpSession	(only in Web Contexts) the HttpSession object

따라서 다음과 같이 표현할 수 있습니다.

```
Established locale country: <span th:text="${#locale.country}">US</span>.
```

부록 A에서 이 객체들에 대한 모든 예제를 확인할 수 있습니다.

4.2.2. 유틸리티 객체 표현

기본 객체 외에, Thymeleaf는 일반적인 작업을 수행하는데 도움이 되는 유틸리티 객체 집합을 제공합니다.

유틸리티 객체	설명
#dates	java.util.Date 객체: 포맷팅, 컴포넌트 추출 등
#calendars	#dates와 유사하지만, Java.util.Calendar 객체를 사용함
#numbers	상수형 객체의 포맷팅
#strings	String 객체: contains, startsWith, prepeding/appending 등
#bools	불린 평가식
#arrays	arrays

#lists	lists
#sets	sets
#maps	maps
#aggregates	배열이나 콜렉션의 집계를 생성
#messages	#{...} 구문을 이용하는 것과 같은 방식으로, 외부화된 메시지를 얻기 위해 사용
#ids	반복되는 id 어트리뷰트를 처리(iteration의 결과 처럼)

부록 B에서 각 유틸리티 객체가 제공하는 기능들에 대해 확인할 수 있습니다.

4.2.3. 홈페이지에 날짜 표시 방법을 변경하기

유틸리티 객체들에 대해 알게 되었으므로, 이를 이용해 홈페이지에 날짜를 표시하는 방법을 변경해 보겠습니다. HomeController를 다음과 같이 변경합니다.

```
SimpleDateFormat dateFormat = new SimpleDateFormat("dd MMMM yyyy");
Calendar cal = Calendar.getInstance();

WebContext ctx = new WebContext(request, servletContext, request.getLocale());
ctx.setVariable("today", dateFormat.format(cal.getTime()));

templateEngine.process("home", ctx, response.getWriter());
```

간단히 이렇게 작성할 수도 있습니다.

```
WebContext ctx = new WebContext(request, servletContext, request.getLocale());
ctx.setVariable("today", Calendar.getInstance());

templateEngine.process("home", ctx, response.getWriter());
```

이제 View 계층에서 직접 날짜의 포맷팅을 수행하게 됩니다.

```
<p>Today is: <span th:text="${#calendars.format(today,'dd MMMM yyyy')}">13 February
2011</span></p>
```

4.3. 선택의 표현(asterisk 구문)

변수는 ${...} 표현식 뿐 아니라 *{...} 으로도 표시할 수 있습니다.

그러나 둘 사이에는 중요한 차이가 있습니다. asterisk 구문은 선택된 객체 뿐 아니라 모든 컨텍스트 변수 맵에서 표현식을 평가 합니다. 물론 선택을 수행하지 않는다면, $와 *의 구문은 동일하게 동작하게 됩니다.

그러면 객체의 선택 이라는 것은 어떤 의미 일까요? 바로 th:object 어트리뷰트입니다. 예제의 userprofile.html 파일을 이용해 설명하겠습니다.

```
<div th:object="${session.user}">
  <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="*{lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
</div>
```

이 코드는 다음과 같이 대체될 수 있습니다.

```
<div>
  <p>Name: <span th:text="${session.user.firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="${session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="${session.user.nationality}">Saturn</span>.</p>
</div>
```

물론, $와 *는 혼합하여 사용할 수 있습니다.

```
<div th:object="${session.user}">
  <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="${session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
</div>
```

객체를 선택하면, 해당 객체는 #object 표현 변수로 $표현식에서 사용할 수 있습니다.

```
<div th:object="${session.user}">
  <p>Name: <span th:text="${#object.firstName}">Sebastian</span>.</p>
  <p>Surname: <span th:text="${session.user.lastName}">Pepper</span>.</p>
  <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
</div>
```

이미 말했던 것처럼, 객체가 선택되지 않은 경우, $와 *표현식은 정확히 동일한 결과를 나타냅니다.

```
<div>
  <p>Name: <span th:text="*{session.user.name}">Sebastian</span>.</p>
  <p>Surname: <span th:text="*{session.user.surname}">Pepper</span>.</p>
  <p>Nationality: <span th:text="*{session.user.nationality}">Saturn</span>.</p>
</div>
```

4.4. URL 링크

URL은 매우 중요하기 때문에, 웹어플리케이션 템플릿의 최상위 요소로 정의되어 있습니다. 그리고 Thymeleaf Standard Dialect는 URL을 표현하기 위한 특별한 구문으로 @{...}을 제공합니다.

URLs 에는 다음 두 가지 종류가 있습니다.

- 절대 경로: http://www.thymeleaf.org
- 상대 경로
  - Page-relative: user/login.html
  - Context-relative: /itemdetails?id=3
  - Server-relative: ~/billing/processInvoice
  - Protocol-relative: //code.jquery.com/jquery-2.0.3.min.js

Thymeleaf는 어떤 상황에서도 절대 경로를 처리할 수 있지만, 상대 경로를 사용하기 위해서는 HTTP

request로부터 정보를 수신하거나 상대 경로를 생성하기 위해 필요하기 때문에 IWebContext 인터페이스의 구현한 context 객체가 필요합니다.

th:href 어트리뷰트를 이용한 새로운 구문을 살펴 보겠습니다.

```
<a href="details.html"
th:href="@{http://localhost:8080/gtvg/order/details(orderId=${o.id})}">view</a>

<a href="details.html" th:href="@{/order/details(orderId=${o.id})}">view</a>
```

여기에는 몇 가지 주의할 사항이 있습니다.

- th:href 어트리뷰트는 수정 어트리뷰트입니다. 일단 한번 처리되면, 이 어트리뷰트는 사용될 링크 URL을 연산하고, <a> 태그의 어트리뷰트에 이 URL을 설정하게 됩니다.
- orderId=${o.id}와 같이 URL 파라미터에 대한 표현식을 사용할 수 있습니다. 이때 필요한 URL 인코딩은 자동으로 수행됩니다.
- 여러 파라미터가 필요한 경우, 다음과 같이 콤마(쉼표)로 구분되어야 합니다. @{/order/process(execID=${execId}, execType='FAST')
- 상대 경로가 / 로 시작하는 경우, 자동으로 앞에 어플리케이션 컨텍스트명 을 붙여 주게 됩니다.
- 쿠키를 사용할 수 없거나, 확인할 수 없는 경우, Thymeleaf는 상대 경로 끝에 ";jsessionid=..."를 접미사로 붙여 세션이 유지될 수 있도록 합니다. 이를 *URL Rewriting* 이라고 부릅니다.
- th:href 태그는 템플릿에서 href 어트리뷰트가 고정 값을 갖도록 지정할 수도 있습니다(선택 사항). 이를 이용하면 프로토타이핑 목적으로 템플릿을 브라우저에서 바로 열었을 때도 링크가 동작할 수 있도록 할 수 있습니다.

메시지 구문과 함께 사용되는 경우처럼, URL은 다른 표현식의 처리 결과에 따릅니다.

```
<a th:href="@{${url}(orderId=${o.id})}">view</a>
<a th:href="@{'/details/'+${user.login}(orderId=${o.id})}">view</a>
```

4.4.1. 홈페이지 메뉴

링크 URL을 생성하는 방법을 배웠으니, 홈페이지에 사이트의 다른 페이지로 이동하는 작은 메뉴를 추가해 보겠습니다.

```
<p>Please select an option</p>
<ol>
  <li><a href="product/list.html" th:href="@{/product/list}">Product List</a></li>
  <li><a href="order/list.html" th:href="@{/order/list}">Order List</a></li>
  <li><a href="subscribe.html" th:href="@{/subscribe}">Subscribe to our
Newsletter</a></li>
  <li><a href="userprofile.html" th:href="@{/userprofile}">See User Profile</a></li>
</ol>
```

4.5. 리터럴

4.5.1. 텍스트 리터럴

텍스트 리터럴은 Single Quotation(작은 따옴표) 사이에 지정된 문자열을 의미합니다. 텍스트 리터럴은 어떤 문자든 포함할 수 있으나, 작은 따옴표를 표현하기 위해서는 ₩'와 같이 ₩를 붙여 escape 처리 해야 합니다.

```
<p>Now you are looking at a <span th:text="'working web application'">template
file</span>.</p>
```

4.5.2. 넘버 리터럴

넘버 리터럴은 보이는 그대로 숫자를 의미합니다.

```
<p>The year is <span th:text="2013">1492</span>.</p>
<p>In two years, it will be <span th:text="2013 + 2">1494</span>.</p>
```

4.5.3. 불린 리터럴

불린 리터럴은 true or false를 의미합니다.

```
<div th:if="${user.isAdmin()} == false"> ...
```

상기 예시와 OGNL/SpringEL 변수 표현식에서 false 리터럴을 표현한느 방식에 차이가 있다는 점을 확인할 수 있습니다. 그러므로 표현식의 처리 엔진이 이를 처리하도록 할 수도 있습니다.

```
<div th:if="${user.isAdmin() == false}"> ...
```

4.5.4. Null 리터럴

다음과 같이 사용됩니다.

```
<div th:if="${variable.something} == null"> ...
```

4.5.5. 리터럴 토큰

사실 숫자, 불린, 그리고 널 리터럴은 모두 리터럴 토큰의 특정 케이스 입니다.

이 토큰들은 Standard Expresssions(표준 표현식)을 좀 더 단순화 할 수 있도록 해 줍니다. 이 토큰들은 텍스트 리터럴과 정확히 같은 동작을 하지만 대소문자와 숫자, 대괄호와 구두점, 하이픈과 언더스코어만 포함할 수 있습니다. 공백이나 콤마등은 사용할 수 없습니다.

이 리터럴 토큰의 장점은 따옴표 처리를 하지 않아도 된다는 것입니다.

따라서 우리는 다음과 같은 코드를 아래 코드 대신 사용할 수 있습니다.

```
<div th:class="content">...</div>
```

```
<div th:class="'content'">...</div>
```

4.6. 텍스트 추가하기

리터럴이든 변수나 메시지 표현식의 처리 결과이든 관계 없이 + 연산자만 사용하면 쉽게 문자열을 추가할 수 있습니다.

```
th:text="'The name of the user is ' + ${user.name}"
```

4.7. 리터럴 대체

리터럴 대체는 리터럴들을 + 로 연결할 필요 없이 변수값을 포함하는 문자열의 포맷팅을 용이하게 해 줍니다. 이렇게 문자열을 대체할 때는 | 와 | 사이에 리터럴을 둘러 싸야 합니다.

```
<span th:text="|Welcome to our application, ${user.name}!|">
```

이는 다음과 같이 대체될 수 있습니다.

```
<span th:text="'Welcome to our application, ' + ${user.name} + '!'">
```

리터럴 대체는 다른 종류의 표현식과 결합하여 사용될 수 있습니다.

```
<span th:text="${onevar} + ' ' + |${twovar}, ${threevar}|">
```

주의: 리터럴 대체의 |...| 내부에는 반드시 변수 표현식(${...}) 만 사용 가능합니다. ('...')나 불린, 숫자 토큰, 조건식 등의 다른 리터럴은 사용할 수 없습니다.

4.8. 산술 연산

+, -, *, /, %와 같은 일부 산술 연산 또한 사용 가능합니다.

```
th:with="isEven=(${prodStat.count} % 2 == 0)"
```

이 연산자는 OGNL 변수 표현식 자체 내에서 적용될 수도 있습니다. 그리고 이런 경우 Thymeleaf Standard Expression 엔진 대신 OGNL 엔진을 이용해 결과가 처리 됩니다.

```
th:with="isEven=${prodStat.count % 2 == 0}"
```

또, div(/) 와 mod(%)와 같은 연산자의 별칭도 사용 가능합니다.

4.9. 비교와 항등

표현식 내의 값은 평소처럼 >, <, >=, <= 기호를 이용해 비교할 수 있습니다. 또한 ==와 != 연산자를 이용해 항등(또는 부정) 여부를 확인할 수 있습니다. XML에서는 <와 > 기호를 어트리뷰트값에 사용할 수 없기 때문에 <와 >로 대체해야 한다는 점을 주의하시기 바랍니다.

```
th:if="${prodStat.count} gt; 1"
th:text="'Execution mode is ' + ( (${execMode} == 'dev')? 'Development' :
'Production')"
```

gt (>), lt (<), ge (>=), le (<=), not (!), 그리고 eq (==), neq/ne (!=)와 같은 연산자의 별칭을 사용할 수도 있습니다.

4.10. 조건식

조건식은 조건을 평가한 결과에 따라 두 표현식 중 하나를 평가하는 것을 의미합니다.

다음 예시를 살펴 보겠습니다(또 다른 attribute modifier인 th:class를 소개합니다).

```
<tr th:class="${row.even}? 'even' : 'odd'">
  ...
</tr>
```

조건식의 세 부분(condition, then, else)은 각각 표현식이므로, 변수(${...}, #{...}), 메시지(#{...}), URLs(@{...}), 또는 리터럴('...') 등이 들어갈 수 있습니다.

조건식은 중첩해 사용할 수도 있습니다.

```
<tr th:class="${row.even}? (${row.first}? 'first' : 'even') : 'odd'">
  ...
</tr>
```

조건이 false 일때 null을 반환하는 경우 Else 표현은 생략할 수 있습니다.

```
<tr th:class="${row.even}? 'alt'">
  ...
</tr>
```

4.11. 기본값 표현(Elvis 연산자)

기본값 표현은 then 구문 없이 조건에 따른 값을 처리하는 특수한 방법입니다. 이는 Groovy와 같은 몇몇 언어에 존재하는 Elvis 연산자와 동일하여, 두 표현식을 사용하여 앞의 표현식이 null을 반환하게 되는 경우, 뒤의 표현식의 결과를 반환해 줍니다.

Profile 페이지에서 이 코드가 어떻게 동작하는지 알아 보겠습니다.

```
<div th:object="${session.user}">
  ...
  <p>Age: <span th:text="*{age}?: '(no age specified)'">27</span>.</p>
</div>
```

위에서 보는 것처럼 ?: 연산자 뒤에 기본값을 설정하여 사용합니다. 이 기본값은 *{age}에 대한 결과가 null 일 때만 적용됩니다. 따라서 이는 다음의 결과와 같습니다.

```
<p>Age: <span th:text="*{age != null}? *{age} : '(no age specified)'">27</span>.</p>
```

조건 값과 마찬가지로 괄호 안에 중첩해 사용할 수도 있습니다.

```
<p>Name: <span th:text="*{firstName}?: (*{admin}? 'Admin' :
#{default.username})">Sebastian</span>.</p>
```

4.12. 전처리

템플릿 표현 처리 기능 외에도, Thymeleaf는 전처리 표현을 가능하게 합니다.

그렇다면 전처리 라는 것은 무엇일까요? 이는 일반적인 표현식이 수행되기 전 다른 표현식을 먼저 실행하는 것을 말합니다. 이를 통해 최종적으로 표현식을 수정해 보여 줄 수 있습니다.

전처리 표현식은 일반적인 표현식과 유사하지만, 더블 언더스코어(__) 기호로 묶여 있다는 점이

다릅니다. (__${expression}__)

다음과 같이 언어 별로 정적 메소드를 호출하는 OGNL 표현을 포함한 I18n Messages_fr.properties 개체가 있다고 가정해 보겠습니다.

```
article.text=@myapp.translator.Translator@translateToFrench({0})
```

또 Messages_es.properties 도 동일하게 가정합니다.

```
article.text=@myapp.translator.Translator@translateToSpanish({0})
```

여기서 우리는 하나의 표현식이나 Locale 정보에 따른 결과에 대한 부분을 마크업으로 생성할 수 있습니다. 이에 따라, 전처리 될 표현식을 추출하면 Thymeleaf는 이를 수행할 것입니다.

```
<p th:text="${__#{article.text('textVar')}__}">Some text here...</p>
```

French Locale에 대한 전처리 과정은 다음과 같은 결과를 만들어 낸다는 것을 눈 여겨 보시기 바랍니다.

```
<p th:text="${@myapp.translator.Translator@translateToFrench(textVar)}">Some text here...</p>
```

전처리 문자 __ 는 어트리뷰트에서 ₩_₩_ 로 escape 처리하여 사용할 수 있습니다.

5. 어트리뷰트값 셋팅하기

이번 장에서는 태그의 바디에 내용을 설정하는 가능한 가장 단순한 방법으로 마크업 태그의 어트리뷰트에 값을 셋팅하는 방법에 대해 설명할 것입니다.

5.1. 여러 어트리뷰트에 값 셋팅하기

웹사이트에서 뉴스레터를 발송하고 사용자가 여기에 가입할 수 있게 하기 위해, /WEB-INF/templates/subscribe.html 템플릿에 폼을 설정했습니다.

```
<form action="subscribe.html">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe me!" />
  </fieldset>
</form>
```

이는 아주 훌륭해 보이지만, 사실 웹 어플리케이션이라기보다는 정적 XHTML 페이지처럼 보입니다. 먼저, 폼의 action 어트리뷰트는 템플릿 파일 내에 정적으로 링크가 걸려 있어 URL rewriting 기능을 제공할 수 없습니다. 둘째로, Submit 버튼의 value 어트리뷰트는 영어로만 표시되도록 만들어져 있는데, 이를 국제화할 필요가 있습니다.

th:attr 어트리뷰트를 입력하고, 태그에 설정된 어트리뷰트값을 변경할 수 있도록 하겠습니다.

```
<form action="subscribe.html" th:attr="action=@{/subscribe}">
  <fieldset>
    <input type="text" name="email" />
```

```
    <input type="submit" value="Subscribe me!" th:attr="value=#{subscribe.submit}"/>
  </fieldset>
</form>
```

이는 매우 간단한 개념입니다. th:attr 은 단순히 어트리뷰트에 값을 설정하는 표현에 불과합니다. 여기에 대응하는 Controller와 message 파일을 가지게 되면, 이 템플릿 파일의 실행 결과는 다음과 같을 것입니다.

```
<form action="/gtvg/subscribe">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="¡Suscríbeme!"/>
  </fieldset>
</form>
```

앞선 장에서 설명했던 것처럼, 새로운 어트리뷰트값 외에도 어플리케이션 콘텍스트의 이름이 자동으로 접두 처리되어 URL Base에 /gtvg/subscribe/ 로 설정된 것을 볼 수 있습니다.

하지만 한번에 하나 이상의 어트리뷰트를 설정하고 싶은 경우에는 어떻게 해야 할까요? XML 규칙에서는 한 태그에 중복된 어트리뷰트를 설정하는 것을 허용하지 않기 때문에 th:attr 은 콤마로 분리된 리스트로 할당되어야 합니다.

```
<img src="../../images/gtvglogo.png"
th:attr="src=@{/images/gtvglogo.png},title=#{logo},alt=#{logo}" />
```

필요한 메시지 파일이 주어지면 이 코드의 실행 결과는 다음과 같이 출력될 것입니다.

```
<input type="submit" value="Subscribe me!" th:attr="value=#{subscribe.submit}"/>
```

5.2. 특정 어트리뷰트에 값 셋팅하기

특정 어트리뷰트에 값을 셋팅한다고 하면, 아마 당신은 다음과 같이 생각할 지도 모릅니다.

```
<input type="submit" value="Subscribe me!" th:attr="value=#{subscribe.submit}"/>
```

그런데 이 마크업들은 참으로 보기 좋지 않습니다. (th:attr의) 어트리뷰트값 내부에 할당을 지정하는 것은 매우 실용적일 수 있지만, 템플릿을 만들 때 마다 매번 이렇게 하는 것은 그다지 고상한 방법이 아닙니다.

그리고 바로 이게 th:attr 이 템플릿에서 거의 사용되지 않는 이유 입니다. 일반적으로 특정 태그 어트리뷰트를 셋팅할 때는 th:attr 처럼 특정하지 않고 그냥 th:* 어트리뷰트를 이용하게 될 것입니다.

그러면 어떤 어트리뷰트가 버튼의 value 어트리뷰트 값을 셋팅하게 해 주는 Standard Dialect 일까요? 명백하게도, th:value 어트리뷰트입니다. 다음 예제를 보겠습니다.

```
<input type="submit" value="Subscribe me!" th:value="#{subscribe.submit}"/>
```

이건 좀 나아 보이는 군요. 그럼 form 태그의 action 어트리뷰트에도 똑같이 적용해 보겠습니다.

```
<form action="subscribe.html" th:action="@{/subscribe}">
```

앞서 home.html에 th:href 어트리뷰트를 입력했던 것을 기억하시는지요? 그것들 역시 같은 어트리뷰트입니다.

```
<li><a href="product/list.html" th:href="@{/product/list}">Product List</a></li>
```

아래와 같은 아주 많은 어트리뷰트들이 있고, 각 어트리뷰트들은 XHTML이나 HTML5의 어트리뷰트에 대응하게 됩니다.

th:abbr	th:accept	th:accept-charset
th:accesskey	th:action	th:align
th:alt	th:archive	th:audio
th:autocomplete	th:axis	th:background
th:bgcolor	th:border	th:cellpadding
th:cellspacing	th:challenge	th:charset
th:cite	th:class	th:classid
th:codebase	th:codetype	th:cols
th:colspan	th:compact	th:content
th:contenteditable	th:contextmenu	th:data
th:datetime	th:dir	th:draggable
th:dropzone	th:enctype	th:for
th:form	th:formaction	th:formenctype
th:formmethod	th:formtarget	th:frame
th:frameborder	th:headers	th:height
th:high	th:href	th:hreflang
th:hspace	th:http-equiv	th:icon
th:id	th:keytype	th:kind
th:label	th:lang	th:list
th:longdesc	th:low	th:manifest
th:marginheight	th:marginwidth	th:max
th:maxlength	th:media	th:method
th:min	th:name	th:optimum
th:pattern	th:placeholder	th:poster
th:preload	th:radiogroup	th:rel
th:rev	th:rows	th:rowspan
th:rules	th:sandbox	th:scheme
th:scope	th:scrolling	th:size
th:sizes	th:span	th:spellcheck
th:src	th:srclang	th:standby
th:start	th:step	th:style
th:summary	th:tabindex	th:target
th:title	th:type	th:usemap
th:value	th:valuetype	th:vspace

th:width	th:wrap	th:xmlbase
th:xmllang	th:xmlspace	

5.3. 동시에 하나 이상의 값을 셋팅하기

th:alt-title 과 th:lang-xmllang 라는 두 가지 아주 특별한 어트리뷰트는 동시에 두개의 어트리뷰트에 같은 값을 셋팅합니다.

- th:alt-title 은 alt 와 title 에 같은 값을 셋팅합니다.
- th:lang-xmllang 은 lang과 xml:lang 에 같은 값을 셋팅합니다.

GTVG 홈페이지에서 아래 코드를 이 어트리뷰트를 이용해 변경할 수 있습니다.

```
<img src="../../images/gtvglogo.png"
th:attr="src=@{/images/gtvglogo.png},title=#{logo},alt=#{logo}" />
```

물론 이렇게 해도 결과는 같습니다.

```
<img src="../../images/gtvglogo.png" th:src="@{/images/gtvglogo.png}"
th:title="#{logo}" th:alt="#{logo}" />
```

th:alt-title 어트리뷰트를 이용하면 이렇게 됩니다.

```
<img src="../../images/gtvglogo.png" th:src="@{/images/gtvglogo.png}" th:alt-
title="#{logo}" />
```

5.4. Appending & Prepending

Thymeleaf는 th:attr 과 같은 방법으로 동작하는 th:attrappend와 th:attrprepend 어트리뷰트를 제공합니다. 이 어트리뷰트는 기존 어트리뷰트값 에 Append(접미사) 나 prepend(접두사) 를 붙여 결과값을 변경하는 어트리뷰트입니다.

예를 들어, 특정 CSS 클래스가 사용자의 동작에 따라 사용되는 경우 Context 변수에 버튼에 추가될 CSS 클래스의 이름을 저장하고 싶을 수도 있습니다.

```
<input type="button" value="Do it!" class="btn" th:attrappend="class=${' ' + cssStyle}" />
```

cssStyle 변수에 warning 이라는 값을 셋팅하고 템플릿을 실행하면 다음과 같은 결과를 얻을 수 있습니다.

```
<input type="button" value="Do it!" class="btn warning" />
```

또한 Standard Dialect에는 th:classappend 라는 특유의 appending 어트리뷰트가 있습니다. 이 어트리뷰트는 element에 기존 CSS 클래스를 덮어 쓰지 않고 새로운 CSS를 추가할 때 사용합니다.

```
<tr th:each="prod : ${prods}" class="row" th:classappend="${prodStat.odd}? 'odd'">
```

(th:each 어트리뷰트에 대해 신경 쓰지 마세요. 이 반복 어트리뷰트에 대해서는 앞으로 다루게 될 것입니다)

5.5. 고정값 불린 어트리뷰트

몇몇 XHTML/HTML5 어트리뷰트는 매우 특별해서, 해당 element에 특정 고정값으로 존재하거나, 아예 값이 없습니다.

checked 를 예를 들어 보면 다음과 같습니다.

```
<input type="checkbox" name="option1" checked="checked" />
<input type="checkbox" name="option2" />
```

XHTML 표준에 따라 check 어트리뷰트에는 checked 외 다른 값은 허용되지 않습니다(HTML5의 규칙은 조금 더 느슨합니다). disabled, multiple, readonly, selected 와 같은 어트리뷰트도 동일합니다.

Standard Dialect는 조건을 평가하여 만약 true 면 고정값으로 설정되고, false 면 아무 값도 설정하지 않는 방식으로 어트리뷰트값을 설정할 수 있게 해 주는 어트리뷰트를 포함하고 있습니다.

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

Standard Dialect에서 제공하는 고정값 불린 어트리뷰트는 다음과 같습니다.

th:async	th:autofocus	th:autoplay
th:checked	th:controls	th:declare
th:default	th:defer	th:disabled
th:formnovalidate	th:hidden	th:ismap
th:loop	th:multiple	th:novalidate
th:nowrap	th:open	th:pubdate
th:readonly	th:required	th:reversed
th:scoped	th:seamless	th:selected

5.6. HTML5 친화적 어트리뷰트와 요소명 지원

좀 더 HTML5에 친화적이 되도록 템플릿에 프로세서를 적용해 완전히 다른 구문을 사용하는 것도 가능합니다.

```
<table>
   <tr data-th-each="user : ${users}">
      <td data-th-text="${user.login}">...</td>
      <td data-th-text="${user.name}">...</td>
   </tr>
</table>
```

data-{prefix}-{name} 구문은 HTML5에서 커스텀 어트리뷰트를 작성할 때, th:*와 같은 네임스페이스를 개발자의 도움 없이 사용하는 기본적인 방법입니다. Thymeleaf는 Standard Dialect 가 아닌 사용자 Dialect를 사용할 수 있도록 자동으로 이런 구문을 만들어 줍니다.

또 {prefix}-{name}는 커스텀 태그를 지정하는 구문으로, W3C Custom Elements specification을 준수합니다. 예를 들어 다음 장에서 배우게 될 th:block나 th-block 같은 엘리먼트에 사용할 수 있습니다.

중요: 향후 이 구문이 th:*를 대체할 것이라는 이야기는 아닙니다. th:*를 계속 사용해도 문제 없습니다.

6. 반복

지금까지 우리는 홈페이지와 회원 정보 페이지, 사용자가 뉴스레터에 가입할 수 있는 페이지를 만들었습니다. 하지만 우리 제품에 대해서는 어떻습니까? 방문자가 우리가 판매하는 것들에 대해 알 수 있게 해 주는 제품 목록을 만들 필요가 있지 않을까요? 당연히 Yes 입니다. 이제 그 작업을 시작해 보겠습니다.

6.1. 기초적인 반복문

/WEB-INF/templates/product/list.html 페이지의 상품 목록을 위해서는 테이블이 필요합니다. 각 상품은 행(row, <tr>)으로 표시되며, 템플릿에서 사용하기 위해 Template Row를 생성 해야 합니다. Template Row는 각 상품이 어떻게 보여질 것인지를 알 수 있게 해 주며, Thymeleaf는 각 상품에 대해 한번씩 Template Row를 반복하게 됩니다.

Standard Dialect는 정확히 이런 기능을 위해 th:each 라는 어트리뷰트를 제공합니다.

6.1.1. th:each 사용하기

상품 목록 페이지를 만들기 위해 서비스 계층에서 상품 목록을 수집하고 템플릿 Context에 추가하는 Controller가 필요합니다.

```
public void process(
        HttpServletRequest request, HttpServletResponse response,
        ServletContext servletContext, TemplateEngine templateEngine) {

    ProductService productService = new ProductService();
    List<Product> allProducts = productService.findAll();

    WebContext ctx = new WebContext(request, servletContext, request.getLocale());
    ctx.setVariable("prods", allProducts);

    templateEngine.process("product/list", ctx, response.getWriter());
}
```

그리고 템플릿에 상품 목록을 반복하게 하기 위해 th:each 를 사용합니다.

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

```
    <link rel="stylesheet" type="text/css" media="all"
          href="../../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <h1>Product list</h1>

    <table>
      <tr>
        <th>NAME</th>
        <th>PRICE</th>
        <th>IN STOCK</th>
      </tr>
      <tr th:each="prod : ${prods}">
        <td th:text="${prod.name}">Onions</td>
        <td th:text="${prod.price}">2.41</td>
        <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
      </tr>
    </table>

    <p>
      <a href="../home.html" th:href="@{/}">Return to home</a>
    </p>

  </body>

</html>
```

prod:${prods} 어트리뷰트값은 "${prods}를 수행한 결과에 있는 각 요소들을 prod 변수의 요소에 할당해 템플릿 조각을 반복하라" 라는 의미 입니다. 위 코드의 각 요소에 이름을 부여해 보겠습니다.

- ${prods} 는 *iterated expression* 반복문 또는 *iterated variable* 반복된 변수
- prod 는 *iteration variable* 또는 *iter variable* 반복 변수

prod 반복 변수는 내부에 <td> 와 같은 태그를 포함한 <tr> 요소 내부에만 사용 가능하다는 것을 주의하시기 바랍니다.

6.1.2. 반복 가능한 값

Thymeleaf에서 java.util.List 오브젝트(객체)만 반복해 사용할 수 있는 것은 아닙니다. 사실 th:each 어트리뷰트가 반복할 수 있도록 고안된 완벽한 오브젝트의 조합이 있습니다.

- java.util.Iterable 을 구현한 모든 오브젝트
- java.util.Map을 구현한 모든 오브젝트. 만약 iterating map 이라면 iter 변수는 java.util.Map.Entry

클래스여야 합니다.

- 모든 배열
- 오브젝트 자신을 포함한 단일 값 리스트로 간주될 수 있는 모든 오브젝트

6.2. 반복 상태 유지하기

th:each 를 이용할 때, Thymeleaf는 상태 변수(*the status variable*)라는 - 반복 상태를 추적하는데 아주 유용한 메커니즘을 제공합니다.

상태 변수들은 th:each 어트리뷰트와 함께 정의되며, 다음과 같은 정보를 포함하고 있습니다.

- 현재의 반복 순서(iteration index), 0부터 시작하는 index 속성(property)
- 현재의 반복 순서(iteration index), 1부터 시작하는 count 속성
- 반복된 변수에 있는 요소들의 총합, size 속성
- 각 반복에 사용되는 반복 변수, current 속성
- 현재 반복이 짝수 번째인지 홀수 번째인지, even/odd 불린 속성
- 현재 반복이 첫 번째인지 여부, first 불린 속성
- 현재 반복이 마지막인지 여부, last 불린 속성

앞선 예제에서 이 어트리뷰트들을 어떻게 사용할 수 있는지 알아 보겠습니다.

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
  </tr>
  <tr th:each="prod,iterStat : ${prods}" th:class="${iterStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
  </tr>
</table>
```

위에서 알 수 있듯이, 상태변수(이 예제에서는 iterStat)는 th:each의 어트리뷰트에 상태변수 자신의 이름이 콤마로 구분되어 정의되어 있습니다. 일단 상태변수가 설정되면, 이 변수는 th:each 어트리뷰트가 태그 내에 정의되어 있는 코드 블록 내부에서만 사용 가능합니다.

이제 템플릿의 연산 결과를 확인해 보겠습니다.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

  <head>
```

```
    <title>Good Thymes Virtual Grocery</title>
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type"/>
    <link rel="stylesheet" type="text/css" media="all" href="/gtvg/css/gtvg.css" />
  </head>

  <body>

    <h1>Product list</h1>

    <table>
      <tr>
        <th colspan="1" rowspan="1">NAME</th>
        <th colspan="1" rowspan="1">PRICE</th>
        <th colspan="1" rowspan="1">IN STOCK</th>
      </tr>
      <tr>
        <td colspan="1" rowspan="1">Fresh Sweet Basil</td>
        <td colspan="1" rowspan="1">4.99</td>
        <td colspan="1" rowspan="1">yes</td>
      </tr>
      <tr class="odd">
        <td colspan="1" rowspan="1">Italian Tomato</td>
        <td colspan="1" rowspan="1">1.25</td>
        <td colspan="1" rowspan="1">no</td>
      </tr>
      <tr>
        <td colspan="1" rowspan="1">Yellow Bell Pepper</td>
        <td colspan="1" rowspan="1">2.50</td>
        <td colspan="1" rowspan="1">yes</td>
      </tr>
      <tr class="odd">
        <td colspan="1" rowspan="1">Old Cheddar</td>
        <td colspan="1" rowspan="1">18.75</td>
        <td colspan="1" rowspan="1">yes</td>
      </tr>
    </table>

    <p>
      <a href="/gtvg/" shape="rect">Return to home</a>
    </p>

  </body>

</html>
```

반복 상태 변수가 완벽하게 동작해서, 홀수 번째의 줄에만 CSS 클래스가 적용되어 있는 것을 확인할

수 있습니다. (row 카운팅은 0부터 시작합니다)

> Thymeleaf 는 DTD 로 설정된 XHTML 1.0 Strict 표준에 따라, <a> 태그와 마찬가지로 자동으로 모든 <td> 태그에 colspan 과 rowspan 어트리뷰트를 해당 어트리뷰트의 기본 값으로 설정합니다. (템플릿 코드에 해당 어트리뷰트를 설정한 적이 없다는 것을 기억하세요) 이런 태그가 페이지를 표시하는 데 아무런 영향을 미치지 않으므로, 걱정하지 않아도 됩니다. 또 다른 예로, 우리가 HTML5(DTD 설정 없이)를 사용하고 있다면, 해당 태그들은 추가 되지 않을 것입니다.

반복 변수를 명시적으로 선언하지 않아도, Thymeleaf는 항상 반복문의 이름에 Stat을 접미사로 붙인 반복변수를 생성할 것입니다.

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
  </tr>
</table>
```

7. 조건문

7.1. 단순 조건문: “if”, “unless”

종종 템플릿 중 일부분이 특정 조건에서만 결과가 보여질 필요가 있습니다.

예를 들어, 상품 테이블에 상품평 개수를 표시하는 컬럼이 상품평이 존재하는 경우에만 보여 지고, 상품평 상세 보기 화면으로 이동하는 링크가 걸리도록 하길 원한다고 가정해 봅시다.

이 작업을 수행하기 위해 th:if 어트리뷰트를 사용할 것입니다.

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
```

```
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:if="${not #lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
</table>
```

봐야 할 코드가 너무 많은데, 일단 가장 중요한 라인에 집중해 보겠습니다.

```
<a href="comments.html"
   th:href="@{/product/comments(prodId=${prod.id})}"
   th:if="${not #lists.isEmpty(prod.comments)}">view</a>
```

이 코드에 대해서는 조금 설명이 필요한데요. 해당 상품에 상품평이 존재하는 경우에만 prodId 파라미터에 상품 id를 부여하고 /product/comments 라는 URL로 상품평 페이지를 링크 시킬 것입니다.

이 코드의 결과를 확인해 보겠습니다. (깔끔하게 보이도록 자동 입력된 rowspan 과 colspan 어트리뷰트를 제거 하였습니다)

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr>
    <td>Fresh Sweet Basil</td>
    <td>4.99</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Italian Tomato</td>
    <td>1.25</td>
    <td>no</td>
    <td>
      <span>2</span> comment/s
      <a href="/gtvg/product/comments?prodId=2">view</a>
    </td>
```

```
  </tr>
  <tr>
    <td>Yellow Bell Pepper</td>
    <td>2.50</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Old Cheddar</td>
    <td>18.75</td>
    <td>yes</td>
    <td>
      <span>1</span> comment/s
      <a href="/gtvg/product/comments?prodId=4">view</a>
    </td>
  </tr>
</table>
```

완벽합니다! 원했던 결과가 정확하게 표시되었습니다.

th:if 어트리뷰트는 불린 조건에서만 실행되지는 않습니다. th:if 어트리뷰트는 이보다 더 큰 호환성을 가지고 있어서, 다음 규칙에서 특정 구문이 true인지 판단합니다.

- 값이 Null이 아닌 경우
  - 불린 값이 true 인 경우
  - 숫자 값이 0이 아닌 경우
  - 문자 값이 0이 아닌 경우
  - 문자열이 "false"나 "off", "no"가 아닌 경우
  - 불린이나, 숫자, 문자나 문자열이 아닌 경우
- 값이 Null인 경우, th:if 는 false 를 반환함

또 th:if의 반대 급부로 th:unless 가 있습니다. 앞선 예제에서 OGNL 표현인 not 대신 th:unless를 이용할 수 있습니다.

```
<a href="comments.html"
   th:href="@{/comments(prodId=${prod.id})}"
   th:unless="${#lists.isEmpty(prod.comments)}">view</a>
```

7.2. Switch 문

자바의 switch 구문을 사용하는 것처럼 th:switch / th:case 어트리뷰트를 이용해 조건에 따라 내용을

표시하는 방법도 있습니다.

아래 예문은 당신이 예상하는 그대로 작동합니다.

```
<div th:switch="${user.role}">
  <p th:case="'admin'">User is an administrator</p>
  <p th:case="#{roles.manager}">User is a manager</p>
</div>
```

th:case가 true로 평가되는 즉시 다른 th:case 어트리뷰트는 false로 평가된다는 점에 주의하시기 바랍니다.

default를 지정할 때는 th:case="*" 를 사용합니다.

```
<div th:switch="${user.role}">
  <p th:case="'admin'">User is an administrator</p>
  <p th:case="#{roles.manager}">User is a manager</p>
  <p th:case="*">User is some other thing</p>
</div>
```

8. 템플릿 레이아웃

8.1. 템플릿 조각 include 하기

8.1.1. 템플릿 조각을 정의하고 참조하기

종종 푸터나 헤더, 메뉴등과 같이 템플릿에 다른 템플릿 조각을 삽입하고자 할 때가 있습니다.

이를 수행하기 위해서는 Thymeleaf에 포함할 템플릿 조각들을 정의해야 하는데, th:fragment 어트리뷰트를 이용해 처리할 수 있습니다.

이제 예제(식료품 판매 웹사이트) 사이트에 기본 카피라이트 푸터를 추가해 보겠습니다.

먼저 다음과 같은 코드를 포함하는 /WEB-INF/templates/footer.html 파일을 만듭니다.

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

  <body>

    <div th:fragment="copy">
      &copy; 2011 The Good Thymes Virtual Grocery
    </div>

  </body>

</html>
```

위 코드를 copy 라는 이름의 조각으로 정의하여, th:include 나 th:replace 어트리뷰트 중 하나를 이용해 쉽게 홈페이지에 추가할 수 있습니다.

```
<body>

  ...

  <div th:include="footer :: copy"></div>

</body>
```

조각을 포함하는 어트리뷰트들은 매우 단순하며, 아래와 같이 세 가지 형식이 있습니다.

- templatename::domselector 또는 templatename::[domselector] 는 DOM Selector에 규정된 templatename의 처리 결과를 템플릿 조각으로 포함합니다.
  - domselector 는 간단히 템플릿 조각의 이름이 될 수 있으니, 위의 예에서 footer::copy 와 같이 templatename::fragmentname 으로 간단하게 규정할 수 있다는 것을 상기하세요.

> DOM selector 구문은 XPath expressions 이나 CSS selector 와 유사합니다. 이 구문에 대한 더 자세한 정보는 Appendix C 를 참조 하시기 바랍니다.

- templatename 은 templatename 으로 명명된 템플릿을 완전히 포함합니다.

> th:include / th:replace 태그는 Template Resolver 에 의해 해석되며, 현재 사용중인 TemplateEngine 을 이용해 처리된다는 점에 유의하시기 바랍니다.

- ::domselector / this::domselector 는 같은 템플릿에 조각을 삽입합니다.

templatename 과 domselector는 아래 예처럼 모든 expressions과 함께 사용할 수 있습니다(심지어 조건문 처리도 가능합니다).

```
<div th:include="footer :: (${user.isAdmin}? #{footer.admin} :
#{footer.normaluser})"></div>
```

템플릿 조각은 어떤 th:* 어트리뷰트도 포함할 수 있습니다. 이 어트리뷰트는 템플릿 조각이 대상 템플릿에 삽입될 때(th:include / th:replace 어트리뷰트와 함께 사용될 때) 평가될 것입니다. 그리고 대상 템플릿에 정의된 컨텍스트 변수의 영향을 받게 됩니다.

> 템플릿 조각을 이용할 때 가장 큰 장점은, Thymeleaf 이 다른 템플릿에 템플릿 조각을 삽입하는 능력을 유지한 채 페이지의 각 템플릿 조각들이 완전하고 검증된 XHTML 구조를 가지고 브라우저에서 완벽하게 display 되도록 만들 수 있다는 점입니다.

8.1.2. th:fragment 없이 템플릿 조각 참조하기

반면 DOM Selector의 덕분에 th:fragment 어트리뷰트 를 사용하지 않고도 템플릿 조각을 포함할 수 있습니다. 이는 Thymeleaf이 전혀 사용되지 않은 다른 어플리케이션의 마크업 코드로도 사용될 수 있습니다.

```
...
<div id="copy-section">
  &copy; 2011 The Good Thymes Virtual Grocery
</div>
...
```

CSS Selector와 유사한 방법으로 위 코드의 id를 참조하는 방법으로 쉽게 템플릿 조각을 사용할 수 있습니다.

```
<body>

  ...

  <div th:include="footer :: #copy-section"></div>

</body>
```

8.1.3. th:include 와 th:replace의 차이점

th:include 가 부모 태그안에 템플릿 조각의 내용을 포함시키는 반면, th:replace 는 부모 태그를 템플릿 조각의 내용으로 대체해 버립니다.

아래 예제를 보겠습니다.

```
<footer th:fragment="copy">
  &copy; 2011 The Good Thymes Virtual Grocery
</footer>
```

그리고 아래와 같이 부모 태그에 두 가지 방법으로 템플릿 조각을 포함해 보겠습니다.

```
<body>

  ...

  <div th:include="footer :: copy"></div>
  <div th:replace="footer :: copy"></div>

</body>
```

이 코드의 실행 결과는 다음과 같습니다.

```
<body>

  ...

  <div>
    &copy; 2011 The Good Thymes Virtual Grocery
  </div>
  <footer>
    &copy; 2011 The Good Thymes Virtual Grocery
  </footer>

</body>
```

th:replace 의 별칭으로 th:substituteby 어트리뷰트를 사용할 수도 있지만, th:replace 어트리뷰트를 사용할 것을 추천합니다. th:substituteby 어트리뷰트는 이후 버전에서는 더 이상 사용되지 않을 수도 있습니다.

8.2. 파라미터를 사용하는 템플릿 조각 Signature[1]

템플릿 조각을 사용할 때, function과 유사한 메커니즘을 만들기 위해 th:fragment 어트리뷰트로

[1] 메소드의 동작에 필요한 파라미터를 정의하는 일. 이하 '조각 서명'

템플릿 조각을 정의하여 파라미터를 규정할 수 있습니다.

```
<div th:fragment="frag (onevar,twovar)">
   <p th:text="${onevar} + ' - ' + ${twovar}">...</p>
</div>
```

th:include 나 th:replace 로 템플릿 조각을 호출할 때 두 가지 구문 중 하나를 사용해야 합니다.

```
<div th:include="::frag (${value1},${value2})">...</div>
<div th:include="::frag (onevar=${value1},twovar=${value2})">...</div>
```

두 번째 방법에서 순서는 중요하지 않습니다.

```
<div th:include="::frag (twovar=${value2},onevar=${value1})">...</div>
```

8.2.1. 조각 서명 없이 템플릿 조각 지역 변수 사용하기

다음과 같이 조각 서명 없이 템플릿 조각이 정의되어 있다 해도 무방합니다.

```
<div th:fragment="frag">
   ...
</div>
```

두 번째 구문을 이용해 템플릿 조각에 변수를 설정해 호출할 수 있습니다.

```
<div th:include="::frag (onevar=${value1},twovar=${value2})">
```

사실 이 것은 th:include 에 th:with 를 조합해 이용하는 것과 동일합니다.

```
<div th:include="::frag" th:with="onevar=${value1},twovar=${value2}">
```

조각 서명을 했든 아니든 템플릿 조각에 지역 변수를 규정하는 것이 컨텍스트를 Zero로 초기화 하는 원인은 아닙니다. 템플릿 조각들은 여전히 모든 컨텍스트 변수에서 현재와 마찬가지로 접근할 수 있습니다.

8.2.2. 템플릿 내 조건 처리를 위한 th:assert

th:assert 어트리뷰트는 콤마(,)로 구분된 표현식 목록으로 규정될 수 있습니다. 표현식은 평가 결과에 따라 true를 반환하거나, 예외를 발생 시켜야 합니다.

```
<div th:assert="${onevar},(${twovar} != 43)">...</div>
```

이것을 이용하면 다음과 같이 파라미터의 유효성을 쉽게 검사할 수 있습니다.

```
<header th:fragment="contentheader(title)"
th:assert="${!#strings.isEmpty(title)}">...</header>
```

8.3. 템플릿 조각 제거하기

상품 목록 템플릿의 마지막 버전을 다시 보겠습니다.

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:unless="${#lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
</table>
```

이 코드는 템플릿으로 적절해 보이지만, 정적 HTML일 뿐 적당한 프로토타입이 될 수는 없습니다.

브라우저에서 제대로 보이기는 하지만, 각 행이 테스트용 데이터를 가지고 있는 테이블에 불과하기 때문입니다. 그저 단순하기만 할 뿐 프로토타입이라고 하기에는 무리가 있습니다. 상품을 추가하기 위해서는 행을 늘려야만 합니다.

다음과 같이 몇 가지를 추가해 보겠습니다.

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:unless="${#lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
  <tr class="odd">
    <td>Blue Lettuce</td>
    <td>9.55</td>
    <td>no</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr>
    <td>Mild Cinnamon</td>
    <td>1.99</td>
    <td>yes</td>
    <td>
```

```
      <span>3</span> comment/s
      <a href="comments.html">view</a>
    </td>
  </tr>
</table>
```

이제 세 개의 상품을 가지게 되었으니, 프로토타입으로서 훨씬 더 낫습니다. 그렇지만 이 템플릿을 Thymeleaf에 실행시켰을 때 어떤 결과가 일어나게 될까요?

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr>
    <td>Fresh Sweet Basil</td>
    <td>4.99</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Italian Tomato</td>
    <td>1.25</td>
    <td>no</td>
    <td>
      <span>2</span> comment/s
      <a href="/gtvg/product/comments?prodId=2">view</a>
    </td>
  </tr>
  <tr>
    <td>Yellow Bell Pepper</td>
    <td>2.50</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Old Cheddar</td>
    <td>18.75</td>
    <td>yes</td>
    <td>
      <span>1</span> comment/s
      <a href="/gtvg/product/comments?prodId=4">view</a>
    </td>
  </tr>
  <tr class="odd">
    <td>Blue Lettuce</td>
    <td>9.55</td>
    <td>no</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr>
    <td>Mild Cinnamon</td>
    <td>1.99</td>
    <td>yes</td>
    <td>
      <span>3</span> comment/s
      <a href="comments.html">view</a>
```

```
    </td>
  </tr>
</table>
```

마지막 두 행은 그저 테스트용 행이었습니다. 그러나 첫 번째 행만 반복이 적용되었을 뿐이므로, Thymeleaf가 다른 두 행을 제거해야 할 이유는 없습니다.

따라서 템플릿이 실행되는 동안 두 행을 제거하는 방법이 필요합니다. 이를 위해 th:remove 어트리뷰트를 두번째와 세번째 <tr> 태그에 적어 줍니다.

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
    <td th:text="${prod.name}">Onions</td>
    <td th:text="${prod.price}">2.41</td>
    <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
    <td>
      <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
      <a href="comments.html"
         th:href="@{/product/comments(prodId=${prod.id})}"
         th:unless="${#lists.isEmpty(prod.comments)}">view</a>
    </td>
  </tr>
  <tr class="odd" th:remove="all">
    <td>Blue Lettuce</td>
    <td>9.55</td>
    <td>no</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr th:remove="all">
    <td>Mild Cinnamon</td>
    <td>1.99</td>
    <td>yes</td>
    <td>
      <span>3</span> comment/s
      <a href="comments.html">view</a>
    </td>
  </tr>
</table>
```

일단 실행되면, 모두 예상했던 대로 보일 것입니다.

```
<table>
  <tr>
    <th>NAME</th>
    <th>PRICE</th>
    <th>IN STOCK</th>
    <th>COMMENTS</th>
  </tr>
  <tr>
    <td>Fresh Sweet Basil</td>
    <td>4.99</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
```

```
    </td>
  </tr>
  <tr class="odd">
    <td>Italian Tomato</td>
    <td>1.25</td>
    <td>no</td>
    <td>
      <span>2</span> comment/s
      <a href="/gtvg/product/comments?prodId=2">view</a>
    </td>
  </tr>
  <tr>
    <td>Yellow Bell Pepper</td>
    <td>2.50</td>
    <td>yes</td>
    <td>
      <span>0</span> comment/s
    </td>
  </tr>
  <tr class="odd">
    <td>Old Cheddar</td>
    <td>18.75</td>
    <td>yes</td>
    <td>
      <span>1</span> comment/s
      <a href="/gtvg/product/comments?prodId=4">view</a>
    </td>
  </tr>
</table>
```

어트리뷰트의 값들과 이 값들이 의미하는 바는 무엇인지 알아 보겠습니다. th:remove 는 속성값에 따라 다음과 같은 세 가지 다른 방식으로 동작합니다.

all	태그와 자식 요소 모두를 제거한다.
body	태그는 남기고 자식 요소를 모두 제거한다.
tag	태그를 제거하되, 자식 요소는 제거하지 않는다.
all-but-first	태그 내의 자식요소 중 첫 번째 요소를 남기고 모두 제거한다.
none	속성값에 표현식으로 동적 평가하여, true이면 제거, false면 유지한다.

all-but-first 속성값은 프토토타이핑 할 때 th:remove='all'을 덜 쓰게 만들어 줍니다.

```
<table>
  <thead>
    <tr>
      <th>NAME</th>
      <th>PRICE</th>
      <th>IN STOCK</th>
      <th>COMMENTS</th>
    </tr>
  </thead>
  <tbody th:remove="all-but-first">
    <tr th:each="prod : ${prods}" th:class="${prodStat.odd}? 'odd'">
      <td th:text="${prod.name}">Onions</td>
      <td th:text="${prod.price}">2.41</td>
      <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
      <td>
        <span th:text="${#lists.size(prod.comments)}">2</span> comment/s
        <a href="comments.html"
           th:href="@{/product/comments(prodId=${prod.id})}"
           th:unless="${#lists.isEmpty(prod.comments)}">view</a>
```

```
      </td>
    </tr>
    <tr class="odd">
      <td>Blue Lettuce</td>
      <td>9.55</td>
      <td>no</td>
      <td>
        <span>0</span> comment/s
      </td>
    </tr>
    <tr>
      <td>Mild Cinnamon</td>
      <td>1.99</td>
      <td>yes</td>
      <td>
        <span>3</span> comment/s
        <a href="comments.html">view</a>
      </td>
    </tr>
  </tbody>
</table>
```

th:remove 어트리뷰트는 all, tag, body, all-but-first, none 중 하나의 String 값을 반환하는 한 어떤 Thymeleaf 표준 표현식이라도 사용 가능합니다.

이는 다음과 같이 조건에 따라 제거할 수 있다는 것을 의미합니다.

```
<a href="/something" th:remove="${condition}? tag : none">Link text not to be
removed</a>
```

th:remove 는 null을 none 으로 인식할 수 있도록 고안되었습니다. 따라서 다음 예제도 제대로 동작합니다.

```
<a href="/something" th:remove="${condition}? tag">Link text not to be removed</a>
```

${조건}이 false인 경우, null이 반환되며, 아무 것도 제거하지 않게 될 것입니다.

9. 지역 변수

Themeleaf 는 템플릿의 특정 조각으로 정의된 변수를 템플릿 조각의 내부 평가식에 사용할 수 있는 지역 변수로 호출할 수 있습니다.

이에 대해서는 이미 상품 목록 페이지에서 prod 반복 변수에서 본 적이 있습니다.

```
<tr th:each="prod : ${prods}">
    ...
</tr>
```

prod 변수는 <tr> 태그의 묶음 내에서만 사용 가능합니다.

- th:each 보다 낮은 우선순위에서 실행되는 모든 th:* 어트리뷰트에서 사용 가능합니다. (th:each 어트리뷰트 다음에 실행된다는 뜻입니다)
- <td>와 같은 모든 <tr> 태그의 자식 요소에서 사용 가능합니다.

Thymeleaf는 반복문 없이 지역 변수를 선언하는 방법을 제공해 줍니다. th:with 어트리뷰트로, 속성값을 지정하는 방법은 다음과 같습니다.

```
<div th:with="firstPer=${persons[0]}">
    <p>The name of the first person is <span th:text="${firstPer.name}">Julius Caesar</span>.</p>
</div>
```

th:with 가 실행되면, firstPer 변수가 지역 변수로 생성되고 다음에 오는 컨텍스트에 변수 맵으로 추가됩니다. 따라서 컨텍스트에서 어떤 변수든 평가식으로 넘길 수 있지만, 이때 컨텍스트는 반드시 태그 내부에 있어야 합니다.

다중 할당 구문을 이용해 여러 변수를 한번에 정의할 수도 있습니다.

```
<div th:with="firstPer=${persons[0]},secondPer=${persons[1]}">
    <p>The name of the first person is <span th:text="${firstPer.name}">Julius Caesar</span>.</p>
    <p>But the name of the second person is <span th:text="${secondPer.name}">Marcus Antonius</span>.</p>
</div>
```

th:with 어트리뷰트는 동일한 속성에 정의된 변수를 재활용할 수 있게 해 줍니다.

```
<div th:with="company=${user.company + ' Co.'},account=${accounts[company]}">...</div>
```

이를 식료품 판매 홈페이지에 적용해 보겠습니다. 앞서 날짜 형식으로 날짜를 표기하는 코드를 떠올려 보시기 바랍니다.

```
<p>Today is: <span th:text="${#calendars.format(today,'dd MMMM yyyy')}">13 february 2011</span></p>
```

만약 로케일 설정에 따라 "dd MMMM yyyy"로 표현하고 싶다면 어떻게 해야 할까요? 예를 들어 우리는 home_en.properies 에 다음과 같은 메시지를 추가해야 할 수도 있습니다.

```
date.format=MMMM dd'','' yyyy
```

또 이렇게도 쓸 수 있습니다.

```
date.format=dd ''de'' MMMM'','' yyyy
```

이제 th:with 를 이용해 변수 내에 지역화된 날짜 형식을 불러와 th:text 표현식에서 사용해 보겠습니다.

```
<p th:with="df=#{date.format}">
  Today is: <span th:text="${#calendars.format(today,df)}">13 february 2011</span>
</p>
```

쉽고 깔끔합니다.

그리고 th:with 는 th:text 보다 높은 우선순위를 가지고 있기 때문에, span 태그에 모두 집어 넣어 처리할 수도 있습니다.

```
<p>
  Today is: <span th:with="df=#{date.format}"
th:text="${#calendars.format(today,df)}">13 february 2011</span>
</p>
```

따로 언급한 적 없이 바로 우선순위에 대해 이야기 하니 혼란스러울 수 있겠지만 걱정할 필요 없습니다. 이는 다음 장에서 배우게 될 것입니다.

10. 어트리뷰트 우선순위

예를 들어, 다음과 같이 한 태그 내에 두 개 이상의 th:* 어트리뷰트를 작성한다면 어떻게 될까요?

```
<ul>
  <li th:each="item : ${items}" th:text="${item.description}">Item description
here...</li>
</ul>
```

우리가 원하는 결과를 얻기 위해서는 th:text 보다 th:each 어트리뷰트가 먼저 실행되어야 할 것입니다. 그러나 DOM(Document Object Model) standard 는 작성된 태그의 어트리뷰트의 순서에 해당하는 정보를 제공하지 않기 때문에 우선순위 메커니즘은 어트리뷰트 자체에서 정의되어야 제대로 작동하게 될 것입니다.

따라서 모든 Thymeleaf 어트리뷰트는 태그 내에서 실행될 때의 우선순위가 설정되어 있으며, 이는 다음과 같습니다.

Order	Feature	Attributes
1	Fragment inclusion	th:include
2	Fragment iteration	th:each
3	Conditional evaluation	th:if th:unless th:switch th:case
4	Local variable definition	th:object th:with
5	General attribute modification	th:attr th:attrprepend th:attrappend
6	Specific attribute modification	th:value, th:href, th:src, etc.
7	Text (tag body modification)	th:text

		th:utext
8	Fragment specification	th:fragment
9	Fragment removal	th:remove

이 우선순위 메커니즘은 어트리뷰트의 위치가 뒤바뀌어 있다고 해도 반복된 템플릿 조각들이 모두 같은 결과를 반환한다는 것을 의미합니다. (비록 가독성이 떨어질 수는 있겠지만)

```
<ul>
  <li th:text="${item.description}" th:each="item : ${items}">Item description
here...</li>
</ul>
```

11. 주석과 블록

11.1. 표준 HTML/XML 주석

표준 HTML/XML 주석인 <!-- ... --> 는 Thymeleaf 템플릿의 어느 위치에서나 사용 가능합니다. Thymeleaf 나 브라우저는 주석 내부에 어떤 것이든 처리하지 않으며 단순히 결과물에 복사할 것입니다.

```
<!-- User info follows -->
<div th:text="${...}">
  ...
</div>
```

11.2. Thymeleaf 파서-레벨 주석 블록

아래 파서-레벨 주석 블록은 Thymeleaf이 파싱할 때 제거되는 코드 입니다.

```
<!--/* This code will be removed at thymeleaf parsing time! */-->
```

Thymeleaf 는 <!--/* 와 */--> 사이에 있는 모든 것을 완전히 제거할 것입니다. 이 주석 블록은 정적으로 템플릿을 열었을 때(브라우저로 그냥 열었을 때) Thymeleaf가 템플릿을 처리할 때 어느 부분이 제거되는지 알 수 있도록 합니다.

```
<!--/*-->
  <div>
    you can see me only before thymeleaf processes me!
  </div>
<!--*/-->
```

프로토타이핑할 때 수많은 <tr> 코드를 가진 테이블을 매우 편리하게 처리할 수 있습니다.

```
<table>
  <tr th:each="x : ${xs}">
    ...
  </tr>
  <!--/*-->
```

```
  <tr>
    ...
  </tr>
  <tr>
    ...
  </tr>
  <!--*/-->
</table>
```

11.3. 프로토타입 전용 주석 블록

Thymeleaf는 Thymeleaf에서 템플릿을 처리할 때만 일반적인 마크업으로 인식하고, 프로토타입의 경우처럼 정적으로 템플릿을 열었을 때만 주석으로 처리하도록 하는 특별한 주석 블록을 제공합니다.

```
<span>hello!</span>
<!--/*/
  <div th:text="${...}">
    ...
  </div>
/*/-->
<span>goodbye!</span>
```

Thymeleaf의 파싱 시스템은 단순히 <!--/*/ 와 /*/--> 마커를 제거하지만, 그 내용은 제거하지 않기 때문에 이 내용 부분이 주석으로 인식되지 않습니다. 따라서 템플릿을 실행할 때 Thymeleaf는 실제로 다음과 같은 처리를 하게 될 것입니다.

```
<span>hello!</span>

  <div th:text="${...}">
    ...
  </div>

<span>goodbye!</span>
```

파서-레벨의 주석 블록 기능은 dialect와 관계 없이 동작한다는 것을 유의하시기 바랍니다.

11.4. th:block 태그 합성하기

th:block 은 Thymeleaf의 Standard Dialects에 포함된 유일한 Element 프로세서(속성이 아닌) 입니다.

th:block 은 템플릿 개발자가 자신이 원하는 속성을 규정할 수 있는 단순한 어트리뷰트 컨테이너입니다. Thymeleaf는 이 속성들을 실행하고 흔적 없이 제거할 것입니다.

이는 각 엘리먼트를 위해 1개 이상의 <tr>이 필요한 반복 테이블을 만들 때 유용합니다.

```
<table>
  <th:block th:each="user : ${users}">
    <tr>
       <td th:text="${user.login}">...</td>
       <td th:text="${user.name}">...</td>
    </tr>
    <tr>
       <td colspan="2" th:text="${user.address}">...</td>
    </tr>
  </th:block>
</table>
```

프로토타입 전용 주석 블록과 함께 사용하면 특히 유용합니다.

```
<table>
  <!--/*/ <th:block th:each="user : ${users}"> /*/-->
  <tr>
    <td th:text="${user.login}">...</td>
    <td th:text="${user.name}">...</td>
  </tr>
  <tr>
    <td colspan="2" th:text="${user.address}">...</td>
  </tr>
  <!--/*/ </th:block> /*/-->
</table>
```

이 방법이 HTML 유효성을 준수하고 있다는 점과 브라우저에서 정적으로 열어도 잘 작동한다는 것에 주목하시기 바랍니다.

12. Inlining

12.1. Text Inlining

Standard Dialect가 태그 어트리뷰트를 이용할 때 필요한 대부분의 것을 제공한다고는 하지만, HTML 텍스트에 직접 표현식을 쓰는 게 더 나을 때도 있을 것입니다.

예를 들어 다음과 같이 쓰기 보다,

```
<p>Hello, <span th:text="${session.user.name}">Sebastian</span>!</p>
```

이렇게 쓰는 게 더 나을 수도 있습니다.

```
<p>Hello, [[${session.user.name}]]!</p>
```

[[...]] 사이에 작성된 표현식은 Thymeleaf에서 인라인으로 고려된 표현이며, th:text 어트리뷰트에서 유효한 어떤 종류의 표현도 사용 가능합니다.

Inlining 이 작동하기 위해서는 th:inline 어트리뷰트를 이용하도록 활성화 해야 합니다. 여기에는 세 가지 값 또는 모드가 있습니다. (text, javascript, none)

```
<p th:inline="text">Hello, [[${session.user.name}]]!</p>
```

th:inline 태그에는 inline 표현을 쓸 수 없기 때문에 어떤 식으로든 부모 태그에 포함되어야 합니다.

```
<body th:inline="text">

  ...

  <p>Hello, [[${session.user.name}]]!</p>

  ...

</body>
```

이렇게 하면 th:text 어트리뷰트를 사용했을 때보다 코드가 간결해 지는데 왜 처음부터 이런 방식을 사용하지 않았는지 궁금할 지도 모릅니다. 하지만 HTML 파일을 직접 브라우저에서 열었을 때 인라인 표현이 그대로 드러나기 때문에, 아마도 프로토타입으로는 더 이상 사용할 수 없을 거라는 점에 주의해야 합니다.

인라인을 썼을 때와 그렇지 않을 때의 차이는 너무나 명확합니다.

```
Hello, Sebastian!
```

(인라인 사용 전)

```
Hello, [[${session.user.name}]]!
```

(인라인 사용 후)

12.2. 스크립트 인라인(Javascript and Dart)

Thymeleaf는 인라인 호환성을 위해 "scripting" 모드 시리즈를 제공하여 몇 가지 스크립트 언어로 작성된 날짜를 통합할 수 있게 합니다.

현재 제공되는 스크립팅 모드는 javascript (th:inline="javascript") 와 dart (th:inline="dart") 입니다.

스크립트 인라인을 작성하는 첫번째 방법은 표현식의 값을 스크립트에 기술하는 것입니다.

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var username = /*[[${session.user.name}]]*/ 'Sebastian';

    ...
/*]]>*/
</script>
```

/*[[...]]*/ 구문은 Thymeleaf가 표현식을 평가하도록 지시합니다. 여기에는 몇 가지 의미가 더 있습니다.

- 자바스크립트의 주석이므로(/* ... */), 브라우저에서 정적으로 열었을 때는 해당 구문이 무시됩니다.
- 인라인 코드 다음에 있는 'Sebastian'은 페이지가 정적으로 보여질 때 처리됩니다.
- Thymeleaf는 표현식을 처리하고 결과를 삽입할 때 표현식 뒷부분의 코드를 제거합니다.

따라서 이 코드의 처리 결과는 다음과 같습니다.

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var username = 'John Apricot';

    ...
/*]]>*/
</script>
```

물론 주석 처리를 하지 않아도 똑 같은 결과를 얻을 수 있지만, 정적으로 페이지를 열었을 때 스크립트 오류가 발생합니다.

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var username = [[${session.user.name}]];

    ...
/*]]>*/
</script>
```

이 평가식은 String에 국한하지 않고 지능적으로 작동합니다. Thymeleaf는 Javascipt/Dart 구문으로 작성된 다음과 같은 종류의 오브젝트에 대해 올바르게 작동합니다.

- Strings
- Numbers
- Booleans
- Arrays
- Collections
- Maps
- Beans(getter/setter 메소드와 함께 제공된 오브젝트)

다음과 같은 코드를 예로 들어 보겠습니다.

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var user = /*[[${session.user}]]*/ null;

    ...
/*]]>*/
</script>
```

${session.user} 구문은 User Object를 평가하여 javascript 구문으로 적절히 대체할 것입니다.

```
<script th:inline="javascript">
/*<![CDATA[*/
    ...

    var user = {'age':null,'firstName':'John','lastName':'Apricot',
                'name':'John Apricot','nationality':'Antarctica'};

    ...
/*]]>*/
</script>
```

12.2.1. 코드 추가

자바스크립트 인라인을 이용할 때는 /*[+...+]*/ 형태의 특수한 주석 구문 사이에 코드를 추가할 수 있는 추가적인 기능을 제공합니다. 이에 따라 Thymeleaf는 템플릿을 처리할 때 자동으로 해당 코드를 주석에서 제외할 것입니다.

```
var x = 23;

/*[+

var msg  = 'This is a working application';

+]*/

var f = function() {
    ...
```

이 코드는 다음과 같이 처리 됩니다.

```
var x = 23;

var msg  = 'This is a working application';

var f = function() {
...
```

이 주석 코드 사이에 표현식을 넣을 수도 있 습니다.

```
var x = 23;

/*[+

var msg  = 'Hello, ' + [[${session.user.name}]];

+]*/

var f = function() {
...
```

12.2.2. 코드 제거

또한 Thymeleaf가 /*[- */ 와 /* -]*/ 사이의 코드를 제거하게 할 수도 있습니다.

```
var x = 23;

/*[- */

var msg  = 'This is a non-working template';

/* -]*/

var f = function() {
...
```

13. 검증과 Doctypes

13.1. 템플릿 검증

앞서 언급한 것처럼, Thymeleaf는 템플릿을 처리 하기 전 유효성을 검증할 수 있도록 VALIDXML과 VALIDXHTML이라는 즉시 사용 가능한(out-of-the-box) 두 가지 표준 템플릿 모드를 제공합니다. 이 모드들은 well-formed XML 뿐 아니라, DTD에 규정된 바에 따라 적절하게 기술된 템플릿에도 필요합니다.

VALIDXHTML 모드를 사용할 때 아래와 같이 DOCTYPE 절을 선언하면, th:* 태그가 DTD에 규정되어 있지 않기 때문에 유효성 오류가 발생 합니다.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

이는 너무나 당연한 일이지만, W3C가 Thymeleaf의 기능들을 W3C 표준으로 등록해야 할 명백한 이유가 없기 때문입니다. 따라서 이를 해결하기 위해 DTD를 변경해야 합니다.

Thymeleaf는 XHTML 표준의 원본 DTD를 복제한 후 th:* 어트리뷰트를 Standard Dialect로 추가한 DTD 파일 셋을 제공합니다. 이게 그 동안 템플릿에 이 DTD를 사용했던 이유 입니다.

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
```

SYSTEM 이라는 식별자는 템플릿을 검증할 때 Thymeleaf 파서가 DHTML 1.0 Strict DTD 파일을 사용하도록 지시합니다. 그리고 http 라는 것은 단순히 식별자일 뿐입니다. 로컬에 내장된 Thymeleaf의 jar 파일을 읽어 들이기 때문에 이에 대해 걱정할 필요는 없습니다.

> 이 DOCTYPE 선언은 완벽히 검증된 것으로, 만약 프로토타입으로 이용할 때 브라우저에서 정적으로 템플릿을 열면 Standard mode 로 처리 될 것입니다.

XHTML을 지원하면서 Thymeleaf 처리가 가능한 DTD 선언들은 다음과 같습니다.

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-transitional-thymeleaf-
4.dtd">
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-frameset-thymeleaf-4.dtd">
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml11-thymeleaf-4.dtd">
```

또한, IDE(통합개발도구/i.e. eclipse)에서 원활히 작동되도록 하기 위해 템플릿 검증 모드를 사용하지 않더라도 html 태그에 th 네임스페이스를 등록할 필요가 있습니다.

```
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:th="http://www.thymeleaf.org">
```

13.2. DOCTYPE 변환

템플릿에 적절한 DOCTYPE은 다음과 같습니다.

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
```

그러나 웹 어플리케이션이 클라이언트의 브라우저에 이 DOCTYPE으로 된 XHTML을 전송하는 것은 적절하지 않습니다. 그 이유는 다음과 같습니다.

- PUBLIC 이 아니어서 W3C Validator로 검증할 수 없습니다.
- 템플릿이 처리된 후에는 모든 th:* 어트리뷰트가 제거되기 때문에 애초에 필요 없습니다.

따라서 Thymeleaf는 DOCTYPE 변환 메커니즘을 가지고 있으며, 이를 이용해 Thymeleaf에 특화된 XHTML DOCTYPE을 표준 DOCTYPE으로 자동 변환하게 됩니다.

예를 들어 템플릿이 XHTML 1.0 Strict DTD를 사용한다고 가정하겠습니다.

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
    ...
</html>
```

Thymeleaf가 템플릿을 처리한 후 XHTML의 결과는 다음과 같이 변환됩니다.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
    ...
</html>
```

이 변환을 위해 특별히 해야 할 일은 없습니다. Thymeleaf가 자동으로 처리할 것이기 때문입니다.

14. 식료품 판매용 사이트에 페이지 추가하기

이제 Thymeleaf의 사용 방법에 대해 많이 알게 되었으므로, 주문 관리를 위한 몇 가지 새로운 페이지를 식료품 판매용 사이트(예제 사이트)에 추가해 보겠습니다.

XHTML 코드에 주목할 것이므로, 해당 컨트롤러에 대해 더 알고 싶다면 번들 예제 소스 코드를 확인해 주시기 바랍니다.

14.1. 주문 목록

/WEB-INF/templates/order/list.html 에 주문 목록 페이지를 생성하겠습니다.

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

  <head>

    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body>

    <h1>Order list</h1>

    <table>
      <tr>
        <th>DATE</th>
        <th>CUSTOMER</th>
        <th>TOTAL</th>
        <th></th>
      </tr>
      <tr th:each="o : ${orders}" th:class="${oStat.odd}? 'odd'">
        <td th:text="${#calendars.format(o.date,'dd/MMM/yyyy')}">13 jan 2011</td>
        <td th:text="${o.customer.name}">Frederic Tomato</td>
        <td th:text="${#aggregates.sum(o.orderLines.{purchasePrice *
amount})}">23.32</td>
        <td>
          <a href="details.html" th:href="@{/order/details(orderId=${o.id})}">view</a>
        </td>
      </tr>
    </table>

    <p>
```

```
      <a href="../home.html" th:href="@{/}">Return to home</a>
    </p>

  </body>

</html>
```

OGNL을 제외하고는 별다른 것이 없습니다.

```
<td th:text="${#aggregates.sum(o.orderLines.{purchasePrice * amount})}">23.32</td>
```

이 코드는 각 주문(OrderLine Object)에 대해 purchasePrice 와 amount 속성(이와 매칭되는 getPurchasePrice()와 getAmount() 메소드를 이용하여)을 곱해 결과값을 반환하고, 이를 #aggregates.sum(...) function을 이용해 총 주문금액을 산출합니다.

14.2. 주문 상세 정보

이제 주문 상세 페이지에 대해 알아 보겠습니다. 여기에서는 애스테리스크(*) 구문을 자주 사용할 것입니다.

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

  <head>
    <title>Good Thymes Virtual Grocery</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" media="all"
          href="../../../css/gtvg.css" th:href="@{/css/gtvg.css}" />
  </head>

  <body th:object="${order}">

    <h1>Order details</h1>

    <div>
      <p><b>Code:</b> <span th:text="*{id}">99</span></p>
      <p><b>Date:</b> <span th:text="*{#calendars.format(date,'dd MMM yyyy')}">13 jan
2011</span></p>
    </div>

    <h2>Customer</h2>

    <div th:object="*{customer}">
      <p><b>Name:</b> <span th:text="*{name}">Frederic Tomato</span></p>
      <p><b>Since:</b> <span th:text="*{#calendars.format(customerSince,'dd MMM
yyyy')}">1 jan 2011</span></p>
    </div>

    <h2>Products</h2>

    <table>
      <tr>
        <th>PRODUCT</th>
        <th>AMOUNT</th>
        <th>PURCHASE PRICE</th>
      </tr>
      <tr th:each="ol,row : *{orderLines}" th:class="${row.odd}? 'odd'">
        <td th:text="${ol.product.name}">Strawberries</td>
        <td th:text="${ol.amount}" class="number">3</td>
```

```
        <td th:text="${ol.purchasePrice}" class="number">23.32</td>
      </tr>
    </table>

    <div>
      <b>TOTAL:</b> <span th:text="*{#aggregates.sum(orderLines.{purchasePrice *
amount})}">35.23</span>
    </div>

    <p>
      <a href="list.html" th:href="@{/order/list}">Return to order list</a>
    </p>

  </body>

</html>
```

중첩 오브젝트 외에는 그다지 새로운 것이 없습니다.

```
<body th:object="${order}">

  ...

  <div th:object="*{customer}">
    <p><b>Name:</b> <span th:text="*{name}">Frederic Tomato</span></p>
    ...
  </div>

  ...
</body>
```

사실 *{name} 은 다음과 동일합니다.

```
<p><b>Name:</b> <span th:text="${order.customer.name}">Frederic Tomato</span></p>
```

15. 추가 설정

15.1. 템플릿 리졸버

Thymeleaf로 더 좋은 웹사이트를 만들기 위해 ServletContextTemplateResolver를 구현한 ITemplateResolver 를 선택해 Servlet Context에서 템플릿을 리소스로 획득할 수 있도록 했습니다.

ITemplateResolver를 구현해 템플릿 리졸버를 만들 수 있는 기능을 제공하는 것 외에도 Thymeleaf는 다음과 같은 세 가지 다른 기능을 제공합니다.

- org.thymeleaf.templateresolver.ClassLoaderTemplateResolver 템플릿을 클래스로더 리소스로 처리

```
return
Thread.currentThread().getContextClassLoader().getResourceAsStream(templateName);
```

- org.thymeleaf.templateresolver.FileTemplateResolver 템플릿을 파일시스템의 파일로 처리

```
return new FileInputStream(new File(templateName));
```

- org.thymeleaf.templateresolver.UrlTemplateResolver 템플릿을 URL(로컬 파일이 아닌 것 포함)로

처리

```
return (new URL(templateName)).openStream();
```

미리 제공되는 ITemplateResolver 의 구현체들은 동일한 설정 파라미터 집합을 가지고 있습니다.

- 접두어 및 접미어(앞서 살펴본 Prefix and Suffix)

```
templateResolver.setPrefix("/WEB-INF/templates/");
templateResolver.setSuffix(".html");
```

- 템플릿 별칭: 템플릿에 이름을 정해 파일명을 직접 사용하지 않는 방법. 템플릿 별칭과 prefix/suffix가 함께 존재하는 경우 별칭이 접두사/접미사보다 먼저 처리됩니다.

```
templateResolver.addTemplateAlias("adminHome","profiles/admin/home");
templateResolver.setTemplateAliases(aliasesMap);
```

- 템플릿을 불러 들일 때 적용될 인코딩

```
templateResolver.setEncoding("UTF-8");
```

- 특정 템플릿에 Default 모드 템플릿 및 패턴 설정

```
// Default is TemplateMode.XHTML
templateResolver.setTemplateMode("HTML5");
templateResolver.getXhtmlTemplateModePatternSpec().addPattern("*.xhtml");
```

- 특정 템플릿의 캐시 여부와 관계 없이 Default 모드 템플릿 캐시 및 패턴 설정

```
// Default is true
templateResolver.setCacheable(false);
templateResolver.getCacheablePatternSpec().addPattern("/users/*");
```

- 템플릿 리졸버에서 파싱한 템플릿 캐시 정보의 TTL 밀리-초 설정

  설정하지 않는 경우 캐시는 LRU에 의해 제거될 것입니다. (캐시가 다 차면 오래된 자료부터 제거)

```
// Default is no TTL (only LRU would remove entries)
templateResolver.setCacheTTLMs(60000L);
```

또한 템플릿 엔진은 몇 가지 템플릿 리졸버를 설정하여 템플릿 리졸버가 선택적으로 동작하게 할 수 있습니다. 만약 첫번째 템플릿 리졸버가 템플릿을 처리할 수 없는 경우 두 번째 리졸버에게 요청하게 됩니다.

```
ClassLoaderTemplateResolver classLoaderTemplateResolver = new
ClassLoaderTemplateResolver();
classLoaderTemplateResolver.setOrder(Integer.valueOf(1));

ServletContextTemplateResolver servletContextTemplateResolver = new
```

```
ServletContextTemplateResolver();
servletContextTemplateResolver.setOrder(Integer.valueOf(2));

templateEngine.addTemplateResolver(classLoaderTemplateResolver);
templateEngine.addTemplateResolver(servletContextTemplateResolver);
```

여러 템플릿 리졸버가 적용되었을 때, Thymeleaf가 해당 템플릿을 처리하는데 의미가 없는 템플릿 리졸버를 빨리 판단해 제거할 수 있도록하여 퍼포먼스를 높일 수 있도록 각 템플릿 리졸버의 패턴을 정하는 것이 좋습니다. 이는 필수적인 것은 아니지만 최적화를 위해서는 필요합니다.

```
ClassLoaderTemplateResolver classLoaderTemplateResolver = new
ClassLoaderTemplateResolver();
classLoaderTemplateResolver.setOrder(Integer.valueOf(1));
// This classloader will not be even asked for any templates not matching these
patterns
classLoaderTemplateResolver.getResolvablePatternSpec().addPattern("/layout/*.html");
classLoaderTemplateResolver.getResolvablePatternSpec().addPattern("/menu/*.html");

ServletContextTemplateResolver servletContextTemplateResolver = new
ServletContextTemplateResolver();
servletContextTemplateResolver.setOrder(Integer.valueOf(2));
```

15.2. 메시지 리졸버

샘플 웹사이트에 메시지 리졸버 구현체를 명시적으로 지정하지 않았는데, 이는 앞서 설명했던 것처럼 org.thymeleaf.messageresolver.StandardMessageResolver 오브젝트를 이용해 구현된다는 의미입니다.

앞서 설명한 방법대로 이 StandardMessageResolver는 템플릿과 같은 이름을 가진 메시지 파일을 조회하는 것으로, Thymeleaf Core 가 제공하는 메시지 리졸버 구현체지만 org.thymeleaf.messageresolver.IMessageResolver 인터페이스를 구현하여 직접 만들 수도 있습니다.

> thymeleaf-spring3 통합 패키지는 MessageSource 오브젝트를 이용해 메시지를 외부화하는 Spring 기본 사상에 맞는 IMessageResolver 구현체를 제공합니다.

템플릿 엔진에 하나 이상의 메시지 리졸버를 추가하려면 어떻게 해야 할까요? 간단합니다.

```
// For setting only one
templateEngine.setMessageResolver(messageResolver);

// For setting more than one
templateEngine.addMessageResolver(messageResolver);
```

또 하나의 템플릿에 여러 메시지 리졸버를 할당하고 싶다면 템플릿 리졸버에 했던 것과 같은 방법을 사용하면 됩니다. 첫번째 메시지 리졸버가 특정 메시지를 처리하지 못한다면 다음 메시지 리졸버를 조회하게 될 것입니다.

15.3. 로깅

Thymeleaf는 로깅에 상당히 많은 주의를 쏟고 있으며, 항상 로깅 인터페이스로 유용한 정보를 최대한 제공하기 위해 노력하고 있습니다.

로깅 라이브러리로는 slf4j 를 이용합니다. 이는 log4j와 같은 사용자 어플리케이션에 로깅 구현체를 개발하기 위한 중간 다리 역할을 합니다.

Thymeleaf 클래스는 사용자가 원하는 수준의 로깅 레벨에 따라 TRACE, DDEBUG, INFO 레벨의 정보를 로그로 남기게 될 것입니다. 반면 일반적인 로깅은 사용자의 다양한 요구사항에 따라 각각 설정할 수 있는 TemplateEngine 클래스와 연동하여 세 가지 특별한 로거를 이용할 것입니다.

- org.thymeleaf.TemplateEngine.CONFIG 은 라이브러리가 초기화 되는 동안 상세한 설정 정보를 반환합니다.
- org.thymeleaf.TemplateEngine.TIMER 는 각 템플릿이 처리되는데 소요된 총 시간에 대한 정보를 반환합니다. (벤치마킹에 유용합니다)
- org.thymeleaf.TemplateEngine.cache 는 캐시에 대한 구체적인 정보를 제공하는 로거들에 대한 접두사 입니다. 캐시 로거들은 사용자에 의해 설정되고 변경될 수 있지만, 기본적으로 다음과 같습니다.
  - org.thymeleaf.TemplateEngine.cache.TEMPLATE_CACHE
  - org.thymeleaf.TemplateEngine.cache.FRAGMENT_CACHE
  - org.thymeleaf.TemplateEngine.cache.MESSAGE_CACHE
  - org.thymeleaf.TemplateEngine.cache.EXPRESSION_CACHE

log4j를 이용해 Thymeleaf의 로깅 인프라를 설정하는 예제는 다음과 같습니다.

```
log4j.logger.org.thymeleaf=DEBUG
log4j.logger.org.thymeleaf.TemplateEngine.CONFIG=TRACE
log4j.logger.org.thymeleaf.TemplateEngine.TIMER=TRACE
log4j.logger.org.thymeleaf.TemplateEngine.cache.TEMPLATE_CACHE=TRACE
```

16. 템플릿 캐시

Thymeleaf는 DOM 프로세싱 엔진과 프로세서(로직의 적용이 필요한 노드의 유형 중 하나)의 집합에 의해 동작하기 때문에 사용자 데이터와 DOM트리를 조합하여 기대하는 결과가 나올 수 있도록 문서의 DOM 트리를 수정합니다.

또 DOM트리가 템플릿을 처리하기 전에 파싱된 템플릿을 저장하는 cache를 포함(기본 설정)하여 템플릿 파일을 파싱하고 읽어 들입니다. 웹어플리케이션에서 아주 유용한 이 기능은 다음과 같은 컨셉으로 기반으로 만들어졌습니다.

- Input/Output 은 어떤 어플리케이션에서든지 항상 가장 느린 부분입니다. 이에 비해 인메모리 프로세스는 매우 빠릅니다.

- 기존 인메모리 DOM트리를 복제하는 것은 템플릿 파일을 읽어 들이고 파싱한 후 DOM 오브젝트 트리를 생성하는 것보다 훨씬 빠릅니다.
- 웹어플리케이션은 보통 소수의 템플릿을 가지고 있습니다.
- 템플릿의 파일 크기는 그다지 크지 않으며, 어플리케이션이 동작하는 동안에는 수정되지 않습니다.

이와 같은 이유로, 대량의 메모리를 낭비하지 않고도 어플리케이션에서 가장 많이 사용되는 템플릿을 캐싱하는 것이 가능하다고 구상하였습니다. 또한 실제로는 절대 변경되지 않을 작은 파일 조각들의 input/output 에 소요되는 많은 시간을 아낄 수 있도록 해 줄 것입니다.

그럼 이 캐시를 어떻게 조작할 수 있을까요?

먼저 우리는 이미 템플릿 리졸버에서 특정 템플릿에 대한 캐시를 가능 여부를 설정하는 법에 대해 학습한 바 있습니다.

```
// Default is true
templateResolver.setCacheable(false);
templateResolver.getCacheablePatternSpec().addPattern("/users/*");
```

또한 StandardCacheManager 구현체로 직접 구현한 인스턴스인 Cache Manager 오브젝트를 이용해 설정을 변경할 수도 있습니다.

```
// Default is 50
StandardCacheManager cacheManager = new StandardCacheManager();
cacheManager.setTemplateCacheMaxSize(100);
...
templateEngine.setCacheManager(cacheManager);
```

캐시를 설정하는 더 많은 정보는 org.thymeleaf.cache.StandardCacheManager 에 대한 JAVADOC API 문서를 참조 하시기 바랍니다.

템플릿 캐시에서 제외할 항목에 대해서도 수동으로 제어할 수 있습니다.

```
// Clear the cache completely
templateEngine.clearTemplateCache();

// Clear a specific template from the cache
templateEngine.clearTemplateCacheFor("/users/userList");
```

이하 부록 및 원문 정보는 Thymeleaf 공식 홈페이지를 참조하시기 바랍니다.

http://www.thymeleaf.org/doc/html/Using-Thymeleaf.html