Thymeleaf

Tutorial: Using Thymeleaf (ja)

Japanese translation by: Mitsuyuki Shiiba (@bufferings)
Document version: 20141222 - 22 December 2014
Project version: 2.1.4.RELEASE
Project web site: http://www.thymeleaf.org

1 Thymeleafの紹介

1.1 Thymeleafって何？

ThymeleafはJavaのテンプレートエンジンライブラリです。XML/XHTML/HTML5で書かれたテンプレートを変換して、アプリケーションのデータやテキストを表示することができます。

ウェブアプリケーション内のXHTML/HTML5を扱う方が得意ですが、どんなXMLファイルでも処理できますし、ウェブアプリケーションでもスタンドアローンアプリケーションでも使用可能です。

Thymeleafのメインゴールは、テンプレート作成のための優雅で整形式の方法を提供することです。そのため、テンプレート内にロジックを記述する方法ではなく、事前定義されたロジックの実行を *DOM(Document Object Model)* 上でXMLタグ・属性によって指定する方法を基本としています。

このアーキテクチャのおかげで、パースしたファイルを賢くキャッシュして実行時のI/O処理を最小限に抑えることができるので、テンプレートを高速に処理することが可能となっています。

さらに、Thymeleafは最初からXMLとウェブ標準を念頭に置いてデザインされているので、必要に応じて完全にバリデーションされた状態のテンプレートを作成することもできます。

1.2 Thymeleafはどんな種類のテンプレートを処理できるの？

Thymeleafは6種類のテンプレートを処理することができます。これをテンプレートモードと呼びます:

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

Legacy HTML5 以外は整形式XMLです。*Legacy HTML5* モードでは閉じていないタグ・値がない属性・引用符で囲まれていない属性が許容されていますが、Thymeleafはこのモードのファイルを最初に整形式XMLに変換します。それでもHTML5としては正しい状態です(そして実際こちらがHTML5を書くのに推奨されている方法です)[1]。

また、バリデーションはXMLとXHTMLのみで使用可能なことに注意してください。

ただ、Thymeleafが処理できるテンプレートのタイプはこれだけではありません。テンプレートを「パースする方法」と結果を「書き込む方法」を指定することで、ユーザーは独自のモードを定義することができます。Thymeleafは、DOMツリーとして表現することができるものであれば何でも(XMLかどうかに関係なく)テンプレートとして効率よく処理することができます。

1.3 ダイアレクト: スタンダードダイアレクト

Thymeleafは非常に拡張性の高いテンプレートエンジンです(実際「テンプレートエンジンフレームワーク」と呼んだほうがいいかもしれません)。Thymeleafでは、処理対象のDOMノードと、そのDOMノードをどのように処理するかを完全に定義することができます。

DOMノードにロジックを適用するものを「プロセッサ」と呼びます。そして、プロセッサー式 — と、いくつかの特別な生成物 — のことをダイアレクトと呼びます。Thymeleafでは「スタンダードダイアレクト」というそのまますぐに使えるコアライブラリを提供していて、大半のユーザーにとってはこれで十分です。

このチュートリアルでカバーしているのはスタンダードダイアレクトです。以降のページで学ぶ全ての属性や文法は特に明記していなくても、このダイアレクトに定義してあります。

もちろん、ライブラリの拡張機能を利用して独自の処理ロジックを定義したい、など(スタンダードダイアレクトを拡張することも含めて)独自のダイアレクトを作りたい場合があるかもしれません。テンプレートエンジンは複数のダイアレクトを同時に使用できます。

> 公式の **thymeleaf-spring3** と **thymeleaf-spring4** 連携パッケージはどちらも「**Spring**スタンダードダイアレクト」と呼ばれるダイアレクトを定義しています。これは、ほぼスタンダードダイアレクトと同じで、そこに**Spring Framework**用の便利機能を少しだけ適用しています(例えば、**Thymeleaf**標準の**OGNL**の代わりに**Spring**式言語を使用するなど)。ですので、**Spring MVC**を使用するような場合でも時間の無駄にはなりません。ここで学ぶことは全て、**Spring**アプリケーションを作成する際にも役立つでしょう。

Thymeleafのスタンダードダイアレクトはどのテンプレートモードでも使用できますが、特にウェブ向けのテンプレートモードに適しています(**XHTML**と**HTML5**モード)。**HTML5**の他に具体的には以下の**XHTML**仕様をサポート・動作確認しています: **“XHTML 1.0 Transitional”**、**“XHTML 1.0 Strict”**、**“XHTML 1.0 Frameset”**、そして **“XHTML 1.1”** です。

スタンダードダイアレクトの大半のプロセッサは「属性プロセッサ」です。属性プロセッサを使用すると、**XHTML/HTML5**テンプレートファイルは処理前であってもブラウザで正しく表示することができます。単純にその属性が無視されるからです。例えば、タグライブラリを使用した**JSP**だとブラウザで直接表示できない場合がありますが:

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

Thymeleafスタンダードダイアレクトでは同様の機能をこのように実現します:

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

ブラウザで正しく表示できるだけでなく、(任意ですが)**value**属性を指定することもできます(この場合の **“James Carrot”** の部分です)。プロトタイプを静的にブラウザで開いた場合にはこの値が表示され、**Thymeleaf**でテンプレートを処理した場合には `${user.name}` の評価結果値で置き換えられます。

必要な場合には、全く同じファイルをデザイナーとデベロッパーが触ることができるので、静的なプロトタイプをテンプレートに変換する労力を減らすことができます。この機能のことを「ナチュラルテンプレーティング」と呼びます。

1.4 全体のアーキテクチャ

Thymeleafのコアは**DOM**処理エンジンです。具体的にいうと —標準の**DOM API**ではなく— 高性能の独自**DOM**実装によってテンプレートのインメモリツリー表現を生成します。その後、そのインメモリツリー上でノードを走査してプロセッサを実行し**DOM**を変更します。**DOM**の変更は現在の設定や、テンプレートに渡されるコンテキストと呼ばれるデータセットに従います。

ウェブドキュメントはオブジェクトツリーとして表現されることが本当によくあるので、**DOM**テンプレート表現の使用はウェブアプリケーションにとても適しています(実際にブラウザは**DOM**ツリーによってメモリ上でウェブページを表現します)。また、多くのウェブアプリケーションで、使用するテンプレート数は数十個程度である、そのテンプレートが大きなサイズではない、アプリケーションの実行中に通常は変更されない、という考えに基づいて**Thymeleaf**はテンプレートの**DOM**ツリーのインメモリキャッシュを利用しています。これによって多くのテンプレート処理で(必要だとしても)ほんの少しの**I/O**しか必要なくなるので、本番環境での実行を速くすることができます。

> このチュートリアルの後ろの方にキャッシュについてと、高速な処理のために**Thymeleaf**がどのようにメモリとリソースを最適化しているかについて説明した章がありますので詳細はそちらを参照してください。

しかし、制約もあります: このアーキテクチャではテンプレート処理に他のアプローチよりも多くのメモリスペースが必要

になります。つまり、(ウェブドキュメントとは対照的な)大きなサイズのデータXMLの作成には使わない方が良いということです。大まかには(といってもJVMのメモリサイズによりますが)1テンプレートを処理するのに数十メガバイトが必要になるXMLファイルを処理する場合は、おそらくThymeleafを使わない方が良いでしょう。

> ここで、データXMLに対してだけこの制約について考えているのは、ウェブのXHTML/HTML5に関しては、そんなに大きなサイズのドキュメントは作成しないからです。ブラウザもDOMツリーを生成するので、そんなことをすると固まってしまいますもんね。

1.5 次に進む前に読むことをお勧めします...

Thymeleafは特にウェブアプリケーションに適しています。そしてウェブアプリケーションには標準というものがあります。みんながこの標準についてよく知っているべきなのですが、ほとんどの人が知りません。たとえウェブアプリケーションの仕事を何年もやっている人であってもです。

HTML5の出現によって、今日のウェブ標準はかつてないほどに混乱しています... 「XHTMLからHTMLに戻るの？」「XMLシンタックスはなくなるの？」「XHTML2.0はどこにいったの？」

ということでこのチュートリアルでは先に進む前に、Thymeleafのウェブサイトの次の記事を読むことを強くお勧めします:
"From HTML to HTML (via HTML)" http://www.thymeleaf.org/doc/articles/fromhtmltohtmlviahtml.html

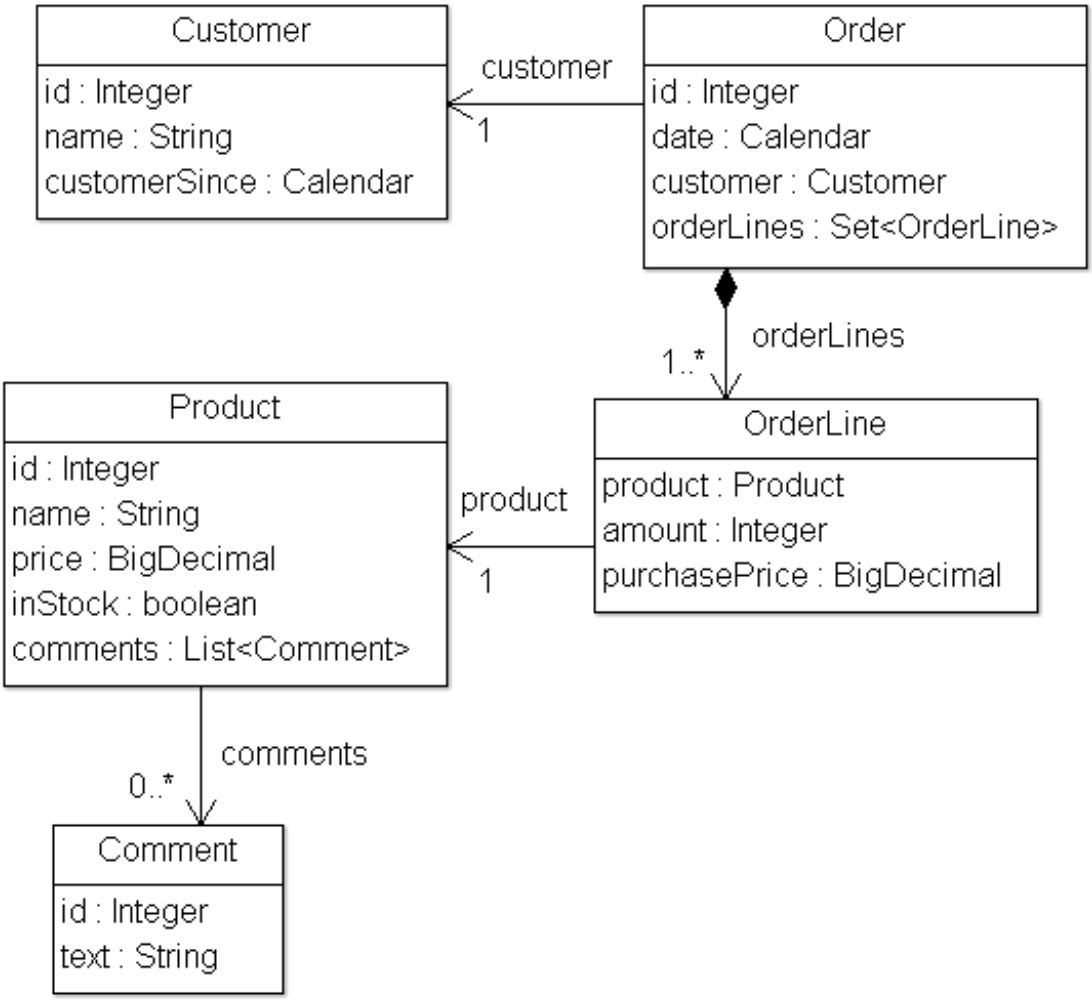
2 The Good Thymes Virtual Grocery(グッドタイムス仮想食料品店)

2.1 食料品店用のウェブサイト

Thymeleafのテンプレート処理のコンセプトを分かりやすく説明するために、このチュートリアルではデモアプリケーションを使用します。デモアプリケーションはプロジェクトのウェブサイトからダウンロードできます。

このアプリケーションは架空の仮想食料品店のウェブサイトで、様々な**Thymeleaf**の機能の例をお見せするのに十分なシナリオが用意されています。

アプリケーションにはとてもシンプルなモデルエンティティが必要でしょう: `Products` は `Orders` を作成することによって `Customers` に販売されます。さらにこの `Products` について `Comments` も管理しましょう:



Example application model

とてもシンプルなサービスレイヤも作りましょう。次のようなメソッドを持つ `Service` オブジェクトです:

```
public class ProductService {

    ...

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

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

}
```

最後に、リクエスト**URL**に応じて**Thymeleaf**に処理を委譲するフィルタをウェブレイヤに作成しましょう:

```
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);
    }

}
```

`IGTVGController` インターフェイスは次のようになります:

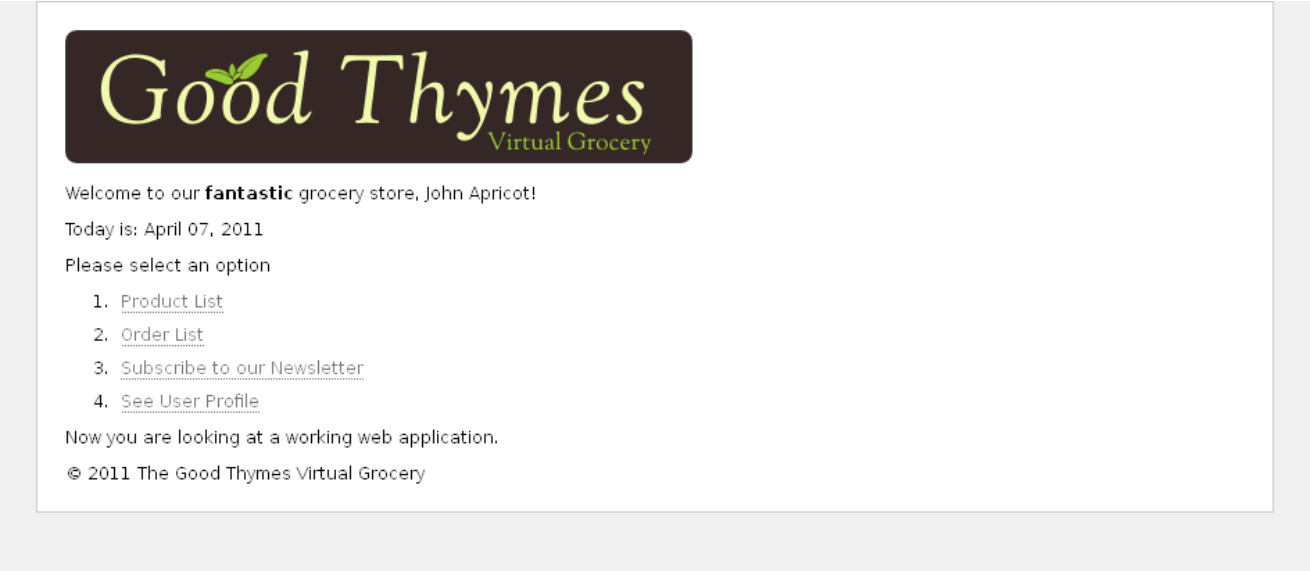
```
public interface IGTVGController {

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

}
```

これで `IGTVGController` の実装を作成すれば良いだけです。データをサービスから受け取って `TemplateEngine` オブジェクトを使用してテンプレートを処理します。

最終的にはこのようになりますが:



Example application home page

まずはテンプレートエンジンの初期化について見てみましょう。

2.2 テンプレートエンジンの作成と設定

フィルタの ***process(...)*** メソッドの中に次のような文があります:

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

これは、**Thymeleaf**を使用するアプリケーションにおいて最も重要なオブジェクトの中の一つである `TemplateEngine` インスタンスの作成と設定を ***GTVGApplication*** クラスが担っているということです。

ここでは `org.thymeleaf.TemplateEngine` を次のように初期化しています:

```
public class GTVGApplication {


    ...
    private static TemplateEngine templateEngine;
    ...


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


    private static void initializeTemplateEngine() {

        ServletContextTemplateResolver templateResolver =
            new ServletContextTemplateResolver();
        // XHTML is the default mode, but we 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");
        // Template cache TTL=1h. If not set, entries would be cached until expelled by LRU
        templateResolver.setCacheTTLMs(3600000L);

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

    }

    ...

}
```

もちろん `TemplateEngine` オブジェクトを初期化するのには様々な方法がありますが、今はこの数行のコードで十分です。

テンプレートリゾルバー

テンプレートリゾルバーからスタートしましょう:

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

テンプレートリゾルバーはThymeleafのAPIである `org.thymeleaf.templateresolver.ITemplateResolver` を実装しています:

```
public interface ITemplateResolver {

    ...

    /*
     * 文字列名(templateProcessingParameters.getTemplateName())によってテンプレートを解決します。
     * このテンプレートリゾルバーで解決できない場合は null を返します。
     */
    public TemplateResolution resolveTemplate(
            TemplateProcessingParameters templateProcessingParameters);

}
```

テンプレートリゾルバーは、どうやってテンプレートにアクセスするかを決定する役割を担っています。GTVGアプリケー

ションの場合は `org.thymeleaf.templateresolver.ServletContextTemplateResolver` 実装を使用して ***Servlet Context*** からテンプレートファイルを取得します: **Java**の全てのウェブアプリケーションにはアプリケーションレベルの `javax.servlet.ServletContext` というオブジェクトが存在し、それによってウェブアプリケーションのルートをリソースパスのルートとしてリソースを解決することができます。

テンプレートリゾルバーにはいくつかのパラメータを設定することができます。まず、標準的なものとして、テンプレートモードがあります:

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

XHTMLは `ServletContextTemplateResolver` のデフォルトテンプレートモードですが意図を明らかにするために書いておくのは良い習慣ですね。

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

prefix と ***suffix*** は文字通り、テンプレート名から実際のリソース名を作り出すために使用されます。

この設定を使用すると ***"product/list"*** というテンプレート名は次の内容と同じになります:

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

任意ですが ***cacheTTLMs*** でテンプレートキャッシュの生存期間を指定することもできます:

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

もちろん**TTL**以内であってもキャッシュのサイズが最大値に達した場合は古いエントリーから削除されます。

> キャッシュの振る舞いやサイズは `ICacheManager` インターフェイスの実装によって定義されます。または、単純にデフォルトで設定されている `StandardCacheManager` を修正しても良いです。

テンプレートリゾルバーについてのより詳細な説明は後ほど行います。今はテンプレートエンジンオブジェクトの生成について見てみましょう。

テンプレートエンジン

テンプレートエンジンオブジェクトとは ***org.thymeleaf.TemplateEngine*** のことです。現在の例ではこのようにエンジンを作成しています:

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

かなりシンプルですよね。インスタンスを作成してテンプレートリゾルバーをセットするだけです。

`TemplateEngine` に必須のパラメータはテンプレートリゾルバーだけです。もちろん他にも色々な設定があります**(**メッセージリゾルバーやキャッシュサイズなど**)**が、それについては後ほど説明します。今はこれだけで十分です。

これでテンプレートエンジンの準備ができました。では、**Thymeleaf**を使用したページの作成に進みましょう。

3 テキストを使う

3.1 複数言語でウェルカム

私たちの食料品店用の最初のタスクはホームページ作成です。

最初のバージョンは非常にシンプルです: タイトルとウェルカムメッセージだけです。 `/WEB-INF/templates/home.html` は以下のようになります:

```
<!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>

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

  </body>

</html>
```

最初に見て欲しいのは、このファイルがどんなブラウザでも正しく表示できる**XHTML**であるということです。理由は、**XHTML**にあるタグしか使っていないからです**(**そしてブラウザは `th:text` のような知らない属性は無視します**)**。また整形式の `DOCTYPE` 宣言を持っているので互換モードではなくスタンダードモードで表示されます。

次に、このファイルは `th:text` のような属性を定義した**Thymeleaf**の**DTD**を指定しているので「妥当な」**XHTML**でもあります[2]。さらに、テンプレートが処理されると**(**全ての `th:*` 属性が取り除かれますが**)**、**Thymeleaf**は自動的に `DOCTYPE` 内の**DTD**定義を標準的な `XHTML 1.0 Strict` のものに置き換えます**(**この**DTD**変換機能に関しては後の章で説明します**)**。

thymeleaf名前空間も `th:*` として定義されています。

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

もしテンプレートの妥当性や、整形式であるかどうかを全く気にしないのであれば単純に標準の `XHTML 1.0 Strict DOCTYPE` を指定すればよく、**xmlns**名前空間の定義も不要であることに気をつけてください:

```
<!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>
```

...こう書いてもXHTMLモードのThymeleafは問題なく処理することができます(IDEの警告で残念な感じになると思いますけど)。

バリデーションに関してはOKですね。ではテンプレートに関する本当に面白い部分に進みましょう: `th:text` 属性を見て行きましょう。

th:text とテキストの外部化

テキストの外部化とは、テンプレートコードのフラグメント(断片)をテンプレートファイルの外に取り出すことです。それによって、テンプレートから切り離された別のファイル(通常は `.properties` ファイル)の中でフラグメントを管理することができ、また、簡単に他の言語で書かれた文字列に置き換えることができます(このことを多言語対応、または *i18n* と呼びます)。外部化されたテキストのフラグメントのことを通常は "メッセージ" と呼びます。

メッセージは、そのメッセージを特定するためのキーを持っており、Thymeleafは `#{...}` という構文を使用してテキストとメッセージの紐付けを行います:

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

ここでは実際、Thymeleafスタンダードダイアレクトの2つの異なる機能を使用しています:

- `th:text` 属性: この属性は値の式を評価した結果をタグのボディに設定します。ここでは、コード内の **"Welcome to our grocery store!"** というテキストを置換します。
- `#{home.welcome}` 式: 「スタンダード式構文」に規定されています。ここでは、テンプレートを処理する全てのロケールで `home.welcome` キーに対応するメッセージを取得して `th:text` 属性で使用するということを意味します。

ふむ。では外部化されたテキストはどこにあるのでしょうか？

Thymeleafでは外部化テキストの場所は `org.thymeleaf.messageresolver.IMessageResolver` を実装することで自由に設定できます。通常は `.properties` ファイルを使用する実装になっていますが、独自実装を作成することも可能です。例えばメッセージをDBから取得することも可能です。

ところで、私たちのテンプレートエンジンには初期化の時にメッセージリゾルバーを指定していません。これは `org.thymeleaf.messageresolver.StandardMessageResolver` クラスによって実装された「スタンダードメッセージリゾルバー」を使用していますよ、ということです。

スタンダードメッセージリゾルバーは `/WEB-INF/templates/home.html` というテンプレートに対してテンプレートと同じフォルダ内で、同じ名前のファイルで拡張子が **.properties** のファイルの中からメッセージを探します。

- `/WEB-INF/templates/home_en.properties` が英語用。

- `/WEB-INF/templates/home_es.properties` がスペイン語用。
- `/WEB-INF/templates/home_pt_BR.properties` がポルトガル語(ブラジル)用。
- `/WEB-INF/templates/home.properties` がデフォルト用(ロケールが一致しない場合)。

`home_es.properties` ファイルを見てみましょう:

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

これでThymeleafのテンプレート処理に必要なことは全て終わりました。ではHomeコントローラーを作成しましょう。

コンテキスト

テンプレートを処理するために `HomeController` クラスを作成します。前述の `IGTVGController` インターフェイスを実装します:

```
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());

    }

}
```

まずはコンテキストの作成について見てみましょう。Thymeleafのコンテキストは `org.thymeleaf.context.IContext` インターフェイスを実装したオブジェクトです。コンテキストはテンプレートエンジンの実行に必要な全てのデータを変数のマップとして持ち、また、外部化メッセージで使用されるロケールへの参照を持っています。

```
public interface IContext {

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

}
```

このインターフェイスの拡張として `org.thymeleaf.context.IWebContext` というインターフェイスがあります:

```
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();

}
```

Thymeleafのコアライブラリはそれぞれの実装を提供しています:

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

コントローラーのコードを見ていただければ分かるように、ここでは `WebContext` を使用しています。というか、そうしなければなりません。 `ServletContextTemplateResolver` が `IWebContext` の実装を必要とするからです。

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

3つの引数のうち2つだけが必須です。ロケールに何も指定しなかったらシステムのデフォルトロケールが使用されます(実際のアプリケーションでは絶対に指定した方がよいですが)。

インターフェイスの定義から `WebContext` はリクエストパラメータ、リクエスト属性、セッション属性、アプリケーション属性を取得するメソッドを持っていることが分かりますが、実際のところ `WebContext` はもう少し色々とやっています:

- 全てのリクエスト属性をコンテキスト変数マップに追加。
- 全てのリクエストパラメータを持つ `param` というコンテキスト変数を追加。
- 全てのセッション変数を持つ `session` というコンテキスト変数を追加。
- 全てのサーブレットコンテキスト属性を持つ `application` というコンテキスト変数を追加。

実行直前に全てのコンテキストオブジェクト(`IContext` の実装)に対して特別な変数が設定されます。 `Context` と `WebContext` のどちらもその対象です。この変数は実行情報(`execInfo`)と呼ばれます。この変数はテンプレートで使用される2つのデータを持っています。

- テンプレート名(`${execInfo.templateName}`): エンジンの実行時に指定される名前です。これは、処理するテンプレート名と一致します。
- 現在日時(`${execInfo.now}`): テンプレートエンジンが現在のテンプレートの処理を開始した日時を示す `Calendar` オブジェクトです。

テンプレートエンジンの実行

コンテキストオブジェクトが準備できたので、あとはテンプレートエンジンを実行するだけです。テンプレート名とコンテキストとレスポンスライターを渡してレスポンスへの書き込みを行います:

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

ではスペイン語ロケールを使用して結果を見てみましょう:

```
<!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 テキストと変数に関するその他のこと

エスケープなしのテキスト

私たちのホームページの最もシンプルなバージョンは準備できましたが、もしメッセージが次のようなものだったらどうしましょう...

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

今のままでテンプレートを実行するとこのようになります:

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

これは本当に欲しい結果ではありません。 `<b>` タグがエスケープされてブラウザに表示されてしまっています。

これは **th:text** 属性のデフォルトの振る舞いです。**Thymeleaf**で**XHTML**タグをエスケープせずに表示したいのであれば、違う属性を使用しなければなりません: `th:utext` **("unescaped text"**用**):**

```
<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>
This will output our message just like we wanted it:
<p>Welcome to our <b>fantastic</b> grocery store!</p>
```

変数の使用と表示

さて、私たちのホームページについてもう少し見てみましょう。例えば、ウェルカムメッセージに次のようなデータを表示したいかもしれません:

```
Welcome to our fantastic grocery store!

Today is: 12 july 2010
```

まずはじめに、コントローラーを修正してコンテキスト変数に日付を追加します:

```
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, response, servletContext, request.getLocale());
    ctx.setVariable("today", dateFormat.format(cal.getTime()));

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

}
```

`String` 型の**today**変数をコンテキストに追加したので、テンプレートで表示することができるようになりました:

```
<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>
```

見ての通りここでも `th:text` 属性を使用しています(タグのボディを置換したいので、これで問題ありません)。ですが構文が少し違いますね。 `#{...}` 式ではなく `${...}` 式を使っています。これが変数用の式です。 ***OGNL (Object-Graph Navigation Language)*** と呼ばれる言語の式でコンテキスト変数マップに対して処理を行います。

この `${today}` は単純に「**today**という名前の変数を取得する」という意味ですが、もっと複雑なこともできます(例えば `${user.name}` は「**user**変数を取得してその `getName()` メソッドを呼び出す」という意味になります)。

属性には様々な値を設定することができます: メッセージ、変数式...などなど。次の章では、どのようなものが指定できるかを全て見ていきましょう。

4 スタンダード式構文

私たちの仮想食料品店の開発は少し休憩して、**Thymeleaf**スタンダードダイアレクトの中でもっとも重要なものの一つについて学んでいきましょう: 「**Thymeleaf**スタンダード式構文」です。

この構文を使って表現された**2**タイプの属性値を既に見てきました: メッセージ式と変数式です:

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

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

ですが、まだ知らないタイプがたくさんあります。また、知っているものにももっと面白い部分があります。初めにスタンダード式の機能の概要を見てみましょう。

- 単純式:
  - 変数式: `${...}`
  - 選択変数式: `*{...}`
  - メッセージ式: `#{...}`
  - リンク**URL**式: `@{...}`
- リテラル
  - テキストリテラル: `'one text'`, `'Another one!'`,...
  - 数値リテラル: `0`, `34`, `3.0`, `12.3`,...
  - 真偽値リテラル: `true`, `false`
  - **Null**リテラル: `null`
  - リテラルトークン: `one`, `sometext`, `main`,...
- テキスト演算子:
  - 文字列結合: `+`
  - リテラル置換: `|The name is ${name}|`
- 算術演算子:
  - バイナリ演算子: `+`, `-`, `*`, `/`, `%`
  - マイナス符号 (単項演算子): `-`
- 論理演算子:
  - 二項演算子: `and`, `or`
  - 論理否定演算子 (単項演算子): `!`, `not`
- 比較と等価:
  - 比較演算子: `>`, `<`, `>=`, `<=` (`gt`, `lt`, `ge`, `le`)
  - 等価演算子: `==`, `!=` (`eq`, `ne`)
- 条件演算子:
  - **If-then:** `(if) ? (then)`
  - **If-then-else:** `(if) ? (then) : (else)`
  - **Default:** `(value) ?: (defaultvalue)`

これら全ての機能は、結合したりネストしたりすることができます:

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

4.1 メッセージ

ご存知の通り `#{...}` メッセージ式は次のように書いて:

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

...これとリンクすることができます:

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

でも、まだ考えていないことが一つあります: メッセージテキストが完全に静的ではない場合はどうしましょうか？例えば、アプリケーションは誰がサイトに訪れているかをいつでも知っているとして、その人の名前を呼んで挨拶文を出したい場合にはどのようにすればいいのでしょうか？

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

つまり、メッセージにパラメータを持たせる必要があるということです。こんなふうに:

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

パラメータは `java.text.MessageFormat` の標準構文に従って指定します。つまり、そのクラスのAPIドキュメントにあるように、数値や日付にフォーマットを指定することもできるということです。

HTTPセッションに持っている `user` という属性をパラメータとして指定するには次のように記述します:

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

必要に応じて複数のパラメータをカンマ区切りで指定することも可能です。実際のところ、メッセージキー自体も変数から取得することができます:

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

4.2 変数

既に述べたように `${...}` 式は、実際にはコンテキスト内の変数マップ上で実行されるOGNL(Object-Graph Navigation Language)式です。

> OGNL構文や機能についての詳細はOGNL Language Guideを参照してください: http://commons.apache.org/ognl/

OGNL構文から次のようなことが分かります。以下の内容は:

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

...実際には次の内容と同等です:

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

ただし、**OGNL**ではもっとパワフルな表現が可能です。こんな風に:

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

...これは、実際には次の処理を実行することでユーザー名を取得します:

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

ですが、**Getter**メソッドのナビゲーションは**OGNL**の機能の**1**つにすぎません。もっと見てみましょう:

```
/*
 * ポイント(.)を使用したプロパティへのアクセス。プロパティのGetterを呼び出すのと同じです。
 */
${person.father.name}

/*
 * プロパティへのアクセスは角括弧([])にプロパティ名を指定することでも可能です。
 * プロパティ名の指定は変数でも、シングルクォートで囲まれた文字列でも可能です。
 */
${person['father']['name']}

/*
 * オブジェクトがマップの場合、ドットも括弧も同様に get(...) メソッドを呼び出します。
 */
${countriesByCode.ES}
${personsByName['Stephen Zucchini'].age}

/*
 * 配列やコレクションに対するインデックスを使用したアクセスも同様に角括弧を使用します。
 * インデックスをクォートなしで書きます。
 */
${personsArray[0].name}

/*
 * メソッド呼び出しが可能です。引数ありでも可能です。
 */
${person.createCompleteName()}
${person.createCompleteNameWithSeparator('-')}
```

式の基本オブジェクト

コンテキスト変数に対して**OGNL**式で評価をする際に、より柔軟に記述できるようにいくつかのオブジェクトを用意しています。これらのオブジェクトの参照は**(OGNL**標準に従って**)** `#` シンボルで始まります:

- `#ctx` : コンテキストオブジェクト。
- `#vars`: コンテキスト変数。
- `#locale` : コンテキストロケール。
- `#httpServletRequest` : (ウェブコンテキストのみ) `HttpServletRequest` オブジェクト。
- `#httpSession` : (ウェブコンテキストのみ) `HttpSession` オブジェクト。

次のようなことができます:

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

詳細は Appendix A を参照して下さい。

式のユーティリティオブジェクト

基本オブジェクト以外にも、式の中の共通のタスクを手助けするためのユーティリティオブジェクトがあります。

- `#dates` : `java.util.Date` オブジェクト用のユーティリティメソッド: フォーマット、コンポーネントの抽出など。
- `#calendars` : `#dates` に似ていますが `java.util.Calendar` オブジェクト用です。
- `#numbers` : 数値オブジェクト用のユーティリティメソッド。
- `#strings` : `String` オブジェクト用のユーティリティメソッド: **contains, startsWith, prepending/appending,** など。
- `#objects` : オブジェクト一般のユーティリティメソッド。
- `#bools` : 真偽値評価用のユーティリティメソッド。
- `#arrays` : 配列用のユーティリティメソッド。
- `#lists` : リスト用のユーティリティメソッド。
- `#sets` : セット用のユーティリティメソッド。
- `#maps` : マップ用のユーティリティメソッド。
- `#aggregates` : 配列やコレクション上での集約処理用ユーティリティメソッド。
- `#messages` : **#{...}** と同様に、変数式内での外部化メッセージを取り扱うためのユーティリティメソッド。
- `#ids` : **(**例えば、イテレーション結果などの**)**繰り返し処理内で**id**属性を取り扱うためのユーティリティメソッド。

それぞれのユーティリティオブジェクトの詳細については Appendix B を参照してください。

私たちのホームページ内の日付を再フォーマット

ユーティリティオブジェクトについて学んだので、それを使って私たちのホームページ内の日付表示を変えてみましょう。
次のように `HomeController` で処理する代わりに:

```
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());
```

...次のように書いて:

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

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

...ビュー側でフォーマットすることができます:

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

4.3 選択したものに対する式 (アスタリスク構文)

変数式は `${...}` だけでなく `*{...}` でも書くことができます。

重要な違いは、アスタリスク構文はコンテキスト変数マップに対してではなく、選択されたオブジェクトに対して評価をする式であるということです。選択されたオブジェクトがない場合は、ダラー構文もアスタリスク構文も全く同じになります。

オブジェクトの選択とはどういうことでしょうか？ `th:object` のことです。では、ユーザープロフィールページ (`userprofile.html`) で使ってみましょう:

```
  <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>
```

これは次と全く同じです:

```
<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>
```

もちろん、ダラー構文とアスタリスク構文は共存可能です:

```
<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>
```

ダラー構文内で `#object` 式変数を使用して選択されているオブジェクトを参照することもできます:

```
<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>
```

繰り返しになりますが、オブジェクトが選択されていない場合はダラー構文とアスタリスク構文は全く同じ意味になります。

```
<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 リンクURL

その重要性から、URLはウェブアプリケーションテンプレートにおけるファーストクラスオブジェクトであり「Thymeleafスタンダードダイアレクト」にも特別な構文が用意されています。 `@` 構文です: `@{...}`

URLにはいくつかのタイプがあります:

- 絶対URL: `http://www.thymeleaf.org`
- 相対URL:
  - ページ相対URL: `user/login.html`
  - コンテキスト相対URL: `/itemdetails?id=3` (サーバー内のコンテキスト名は自動的に付与されます)
  - サーバー相対URL: `~/billing/processInvoice` (同じサーバー内の異なるコンテキスト(= application)のURLを呼び出すことができます。)
  - プロトコル相対URL: `//code.jquery.com/jquery-2.0.3.min.js`

Thymeleafでは絶対URLはどんな場合でも使用できますが、相対URLを使用する場合は `IWebContext` を実装したコンテキストオブジェクトが必要です。そのコンテキストオブジェクトを使用して、相対リンクを生成するための情報をHTTPリクエスト内から取得します。

ではこの新しい構文を使ってみましょう。 `th:href` 属性で使用します:

```
<!-- Will produce 'http://localhost:8080/gtvg/order/details?orderId=3' (plus rewriting) -->
<a href="details.html"
   th:href="@{http://localhost:8080/gtvg/order/details(orderId=${o.id})}">view</a>

<!-- Will produce '/gtvg/order/details?orderId=3' (plus rewriting) -->
<a href="details.html" th:href="@{/order/details(orderId=${o.id})}">view</a>

<!-- Will produce '/gtvg/order/3/details' (plus rewriting) -->
<a href="details.html" th:href="@{/order/{orderId}/details(orderId=${o.id})}">view</a>
```

いくつか注意点:

- `th:href` は属性変更用の属性です: リンクURLを生成し `<a>` タグのhref属性にセットします。
- URLパラメータを指定することができます(`orderId=${o.id}` の部分です)。自動的にURLエンコーディングされます。
- 複数のパラメータを指定する場合はカンマ区切りで指定できます
  `@{/order/process(execId=${execId},execType='FAST')}`
- URLパス内でも変数式は使用可能です `@{/order/{orderId}/details(orderId=${orderId})}`
- `/` で始まる相対URL(`/order/details`)に対しては、自動的にアプリケーションコンテキスト名を前に付けます。
- クッキーが使用できない場合、またはまだ分からない場合は `";jsessionid=..."` を相対URLの最後につけてセッションをキープできるようにすることがあります。これは *URL Rewriting* と呼ばれていますが、Thymeleafでは全てのURLに対してサーブレットAPIの `response.encodeURL(...)` のメカニズムを使用して独自リライトフィルタを追加することができます。
- `th:href` タグを使用する場合、(任意ですが)静的な `href` 属性をテンプレートに同時に指定することができます。そうすることでプロトタイプ用途などで直接テンプレートをブラウザで開いた場合でもリンクを有効にすることができます。

メッセージ構文(`#{...}`)のときと同様に、URL構文でも他の式の評価結果が使用可能です。

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

私たちのホームページ用のメニュー

リンクURLの作成方法がわかったので、ホームにサイト内の他のページへの小さなメニューを加えてみましょうか。

```
<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>
```

サーバールート相対URL

追加のシンタックスを使用して、(コンテキストルート相対URLの代わりに)サーバールート相対URLを作成することができます。 `@{~/path/to/something}` のように指定することで、同じサーバーの異なるコンテキストへのリンクを作成することができます。

4.5 リテラル

テキストリテラル

テキストリテラルはシングルクォートで囲まれた文字列です。どんな文字でも大丈夫ですが、シングルクォート自体は `\'` のようにエスケープしてください。

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

数値リテラル

数値リテラルは数字そのままです。

```
<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>
```

真偽値リテラル

真偽値リテラルは `true` と `false` です:

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

ここで注意して欲しいのは、 `== false` が括弧の外側にあるということです。この場合はThymeleaf自身が処理します。もし括弧の中にある場合は、OGNL/SpringELのエンジンが処理を担当します。

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

nullリテラル

`null` リテラルも使用可能です:

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

リテラルトークン

数値、真偽値、**null**リテラルは実は「リテラルトークン」の特定のケースなのです。

このリテラルトークンはスタンダード式を少しだけシンプルにしてくれます。テキストリテラル(`'...'`)と全く同様の動きをしますが次の文字しか使用できません: 文字(`A-Z` and `a-z`)、数字(`0-9`)、括弧(`[` と `]`), ドット (`.`), ハイフン (`-`) アンダースコア (`_`)。ですので、空白文字やカンマ等は使用できません。

この利点は何でしょうか？それはトークンはクォートで囲む必要がないという点です。ですので、次のように書く代わりに:

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

こう書くことができます:

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

4.6 テキストの追加

テキストは `+` 演算子で追加できます。文字列リテラルであっても、値やメッセージ式の評価結果であっても大丈夫です:

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

4.7 リテラル置換

リテラル置換を使用すると複数の変数から文字列を作成するフォーマットが簡単になります。 `'...' + '...'` のようにリテラルを追加する必要がありません。

リテラル置換を使用する場合は、縦棒(`|`)で囲みます:

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

これは以下の内容と同じです:

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

リテラル置換は他の式と組み合わせて使用することができます:

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

注意点: リテラル置換(`|...|`)内で使用可能なのは、変数式(`${...}`)だけです。他のリテラル(`'...'`)や真偽値/数値トークンや条件式などは使用できません。

4.8 算術演算子

いくつかの算術演算子が使用可能です: `+`, `-`, `*`, `/`, `%`

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

この演算子はOGNL変数式の中でも使用可能なことに注意して下さい(その場合はThymeleafスタンダード式エンジンの代わりにOGNLによって計算されます)。

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

いくつかの演算子には文字列エイリアスもあります: `div` (`/`), `mod` (`%`)

4.9 比較演算子と等価演算子

式の中の値は `>`, `<`, `>=`, `<=` シンボルで比較できます。また、 `==` と `!=` 演算子で等価性を確認できます。ただし、XMLの属性値には `<` と `>` を使用すべきではないと策定されていますので、代わりに `<` と `>` を使用すべきです。

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

文字列エイリアスもあります: `gt` (`>`), `lt` (`<`), `ge` (`>=`), `le` (`<=`), `not` (`!`), `eq` (`==`), `neq` / `ne` (`!=`)。

4.10 条件式

「条件式」は条件(それ自体が別の式です)を評価した結果によって、2つのうちのどちらかの式を評価することを意味します。

例を見てみましょう(今回は `th:class` という「属性変更子」を使用しますね):

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

条件式の3つのパーツ全て(`condition`, `then` and `else`)がそれぞれ式になっています。つまり、変数(`${...}`, `*{...}`)やメッセージ(`#{...}`)や、URL(`@{...}`)やリテラル(`'...'`)を使うことができるということです。

条件式は括弧で囲むことでネスト可能です:

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

Else式は省略可能です。その場合、条件がfalseのときにはnull値が返されます。

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

4.11 デフォルト式 (エルビス演算子)

「デフォルト式」は「then」のない特別な条件式です。Groovyなどの「エルビス演算子」と同じです。2つの式を指定して最初の式がnullを返した場合にのみ2番目の式の値が評価されます。

実際にユーザープロフィールページを見てみましょう:

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

演算子は `?:` です。年齢(`*{age}`)がnullの場合にのみラベル(今回はリテラル値)を表示します。つまり、以下の内容と同じです:

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

括弧で囲むことでネスト可能です:

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

4.12 プリプロセッシング

ここまで見てきた式に加えて、Thymeleafは「プリプロセッシング」式を提供します。

プリプロセッシングとはどういうことでしょうか？それは、通常の式よりも先に評価されるということです。それによって、最終的に実行される実際の式の変更をすることができます。

プリプロセッシング式は普通の式と全く同じように書くことができますが、二重のアンダースコアで囲まれています(`__${expression}__`)。

i18nの `Messages_fr.properties` のエントリに言語特有のスタティックメソッドを呼び出すようなOGNL式が含まれているとしましょう:

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

... `Messages_es.properties` の対応する部分:

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

ロケールに応じた式を評価してマークアップを作成する必要があるので、まずは(プリプロセッシングで)式を選択して、その次にThymeleafにそれを実行させます:

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

フランス語ロケールの場合のプリプロセッシングは次と同等になります:

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

プリプロセッシング用文字列 `__` は属性の中では `\_\_` とエスケープします。

5 属性値を設定する

この章ではThymeleafでどのようにしてマークアップタグ内の属性値を設定(または変更)するかを説明します。タグのボディの内容を設定する機能の次に必要な基本機能かもしれません。

5.1 任意の属性に値を設定する

私たちのウェブサイトでニュースレターを発行するとしましょう。ユーザーが購読できるようにしたいので `/WEB-INF/templates/subscribe.html` テンプレートにフォームを設置します:

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

これで全然問題ないように見えます。しかし実際はこのファイルはウェブアプリケーションのテンプレートというよりは静的なXHTMLに見えます。まず、action属性がこのテンプレートファイル自身への静的リンクなので、URLを書き換える方法がありません。次に、submitボタンのvalue属性は英語で表示されますが多言語対応したいですよね。

ということで `th:attr` 属性を使いましょう。これで、タグの中の属性値を変更することができます。

```
<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>
```

コンセプトは非常に直感的です: `th:attr` には単純に属性に値を代入する式を書きます。対応するコントローラーやメッセージファイルを作成することによって、想定通りの処理結果が得られます:

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

新しい属性の値が使用されていることに加えて `/gtvg/subscribe` のURLには、既に説明したようにアプリケーションコンテキスト名が自動的に付け加えられています。

同時に複数の属性に値を設定したい場合はどうしたらよいでしょうか？XMLでは1つのタグの中に同じ属性を2つ以上書くことはできませんので `th:attr` にカンマ区切りのリストを指定します:

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

メッセージファイルを用意すれば、このような出力になります:

```
<img src="/gtgv/images/gtvglogo.png" title="Logo de Good Thymes" alt="Logo de Good Thymes" />
```

5.2 特定の属性に値を設定する

ここまでで、次のような書き方はすごく汚いなぁと思っているかもしれませんね:

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

属性の中で値を設定するというのはとても実用的ではありますが、常にそうしないといけないというのはエレガントではありません。

ですよね。なので実際のところ `th:attr` 属性はテンプレート内ではほとんど使われません。通常は `th:*` 属性を使用します。この属性を使用すると(`th:attr` のような任意の属性ではなく)特定のタグ属性に値を設定することができます。

ではスタンダードダイアレクトでボタンの `value` 属性に値を設定するにはどのような属性を使用すればいいのでしょうか？これはかなり分かりやすいと思います。 `th:value` です。では見てみましょう:

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

この方が全然良いですよね！同様に `form` タグの `action` 属性も見てみましょう:

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

`th:href` 属性を `home.html` で使用したのを覚えていますか？これも同じです:

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

このような属性が非常にたくさん用意されていて、それぞれが特定の**XHTML**や**HTML5**のタグを対象にしています:

|———————+———————+———————|| 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 複数の値を同時に設定する

ここでは**2**つのちょっと特別な属性を紹介します。 `th:alt-title` と `th:lang-xmllang` です。**2**つの属性に同じ値を同時に指定することができます。具体的には:

- `th:alt-title` は `alt` と `title` を設定します。
- `th:lang-xmllang` は `lang` と `xml:lang` を設定します。

私たちの**GTVG**ホームページで次のように書いている部分は:

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

このように書くこともできますし:

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

このように書くこともできます:

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

5.4 前後に追加する

`th:attr` と同じように任意の属性に対して作用するものとして、**Thymeleaf**には `th:attrappend` と `th:attrprepend` 属性があります。既存の属性値の前や後ろに評価結果を付け加えるための属性です。

例えばあるボタンに対して、ユーザーが何をしたかによって異なる**CSS**クラスを追加**(**設定ではなく追加**)**したい場合が考えられます。これは簡単です:

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

`cssStyle` 変数に `"warning"` という値を設定してテンプレートを処理すると次の結果が得られます:

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

スタンダードダイアレクトには**2**つの特別な属性追加用の属性があります: `th:classappend` と `th:styleappend` です。**CSS** クラスや ***style*** の一部を既存のものを上書きせずに追加します:

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

(`th:each` 属性のことは心配しないでください。「繰り返し用の属性」として後ほど説明します。**)**

5.5 固定値ブール属性

XHTML/HTML5属性の中には、決まった値を持つか、その属性自体が存在しないかのどちらか、という特別な属性があります。

例えば `checked` です:

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

XHTML標準では `checked` 属性には `"checked"` という値しか設定できません**(HTML5**では少し緩いですが**)**。 `disabled`, `multiple`, `readonly` と `selected` も同様です。

これらの属性に対して条件の結果によって値を設定するための属性を、スタンダードダイアレクトでは提供しています。条件の評価結果が**true**の場合はその固定値が設定され、**false**の場合は属性自体が設定されません:

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

スタンダードダイアレクトには次のような固定値ブール属性があります:

|———————-+———————-+———————-|| 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フレンドリーな属性や要素名のサポート

よりHTML5フレンドリーな書き方もできます。これは全く異なる構文になります。

```
<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}` 構文は、 `th:*` などの名前空間を使用せずに独自属性を書くためのHTML5での標準的な方法です。Thymeleafでは、(スタンダードダイアレクトだけでなく)全てのダイアレクトでこの構文を使用することができます。

`{prefix}-{name}` という形式で独自タグを指定するための構文もあります。これは *W3C Custom Elements specification* (より大きな *W3C Web Components spec* の一部です)に準拠しています。例えば `th:block` 要素(または `th-block`)で使用することができますが、これについては後述します。

重要: この構文は名前空間を使用した `th:*` に追加された機能であって、置き換えるものではありません。将来的に名前空間構文を非推奨にする意図は全くありません。

6 繰り返し処理

ここまでホームページとしてユーザープロフィールページと、ニュースレター購読ページを作ってきました。ですが、商品についてはどうでしょう？訪問者に、私たちの商品を知ってもらうための商品一覧ページを作るべきではないでしょうか？えぇ、明らかに**Yes**ですね。ではそうしましょう。

6.1 繰り返し処理の基礎

`/WEB-INF/templates/product/list.html` ページに商品一覧を掲載するためにテーブルが必要です。**1**行**(** `<tr>` 要素**)**に**1**商品ずつ表示したいので、テンプレートの中に「テンプレート行」**(**各商品がどのように表示されるかを示す行**)**を作って、それを**Thymeleaf**で商品ごとに繰り返す必要があります。

スタンダードダイアレクトにはそのための属性があります。 `th:each` です。

th:each を使用する

商品一覧ページのコントローラーはサービスレイヤから商品一覧を取得してテンプレートコンテキストにそれを追加します:

```
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());
}
```

では商品リストを繰り返し処理するために `th:each` を使いましょう:

```
<!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>
```

上記の `prod : ${prods}` 属性値は「 `${prods}` の評価結果の各要素に対して、その要素を**prod**という変数に詰めて、 このテンプレートのフラグメントを繰り返し処理する」という意味になります。呼び名を決めておきましょう。

- ここでは `${prods}` のことを「被繰り返し式」または「被繰り返し変数」と呼びます。[3]
- ここでは `prod` のことを「繰り返し変数」と呼びます。

繰り返し変数 `prod` は `<tr>` 要素の内部だけで使用できることに注意してください。(`<td>` のような内部のタグでも使用可能です)

繰り返し処理が可能な値

Thymeleafの繰り返し処理で使用可能なのは `java.util.List` だけではありません。実際に `th:each` ではオブジェクト一式が「繰り返し可能」だと見なされます。

- `java.util.Iterable` を実装しているオブジェクト
- `java.util.Map` を実装しているオブジェクト。マップを繰り返し処理する場合の繰り返し変数は `java.util.Map.Entry` のクラスになります。
- 配列
- その他のオブジェクトは、そのオブジェクト自身のみを要素として持つ、**1**要素だけのリストのように扱われます。

6.2 繰り返しステータスの保持

`th:each` を使用する際に、繰り返し処理中のステータスを知るための便利なメカニズムがThymeleafにはあります:「ステータス変数」です。

ステータス変数は `th:each` 属性の中で定義され、次の内容を保持しています:

- `index` プロパティ: 0始まりの現在の「繰り返しインデックス」
- `count` プロパティ: 1始まりの現在の「繰り返しインデックス」
- `size` プロパティ: 被繰り返し変数の全要素数
- `current` プロパティ: 繰り返し中の「繰り返し変数」
- `even/odd` 真偽値プロパティ: 現在の繰り返し処理が偶数か奇数か
- `first` 真偽値プロパティ: 現在の繰り返し処理が最初かどうか
- `last` 真偽値プロパティ: 現在の繰り返し処理が最後かどうか

ではどのように使用するのかを前回の例で見てみましょう:

```
<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>
```

ご覧のとおり `th:each` 属性の中で、繰り返し変数の後ろにカンマで区切って名前を書いてステータス変数(この例では `iterStat`)を定義します。繰り返し変数と同様、ステータス変数も `th:each` 属性を持っているタグによって定義されたフラグメントの内部でのみ使用可能です。

それでは、テンプレートの処理結果を見てみましょう:

```
<!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>
```

繰り返しステータス変数は完璧に動いていますね。 `odd` CSSクラスが奇数行のみに適用されています(行番号は0から始まります)。

> colspanとrowspan属性が `<td>` タグに追加されていますが、これは `<a>` のshape属性と同様に、選択されている *XHTML 1.0 Strict* 標準のDTDに従ってThymeleafが自動的に追加します。 *XHTML 1.0 Strict* 標準では、これらの値が属性のデフォルト値として策定されています(テンプレートでは値を設定していないことに注意してください)。ページの表示には影響はないので、このことを気にする必要は全然ありません。例えば、HTML5(にはDTDがありませんが)を使用していたら、この属性は決して追加されません。

ステータス変数を明示的に指定しない場合は、繰り返し変数の後ろに `Stat` をつけた変数名をThymeleafはいつでも作成し

ます:

```
<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 条件の評価

7.1 単純な条件: “if” と “unless”

特定の条件が満たされる場合にのみ、フラグメントを表示したい場合があるでしょう。

例えば、商品テーブルの各商品に対してコメント数を表示するカラムを用意する場合を想像してみてください。もしコメントがあれば、その商品のコメント詳細ページへのリンクを貼りたいです。

この場合 `th:if` 属性を使用します:

```
<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>
```

結構沢山のことをやっているので、重要な行にフォーカスしましょう:

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

実際、ほとんど説明することはないですね: 商品の `id` を `prodId` パラメータに設定してコメントページ (`/product/comments`)へのリンクを作成します。でもそれは商品にコメントがついている場合だけです。

では、結果のマークアップを見てみましょう(見やすくするために、デフォルト属性の `rowspan` と `colspan` は取り除いています):

```
<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>
```

カンペキ！まさに欲しかったものです。

`th:if` 属性は ***boolean*** 条件のみを評価するわけではないことに注意して下さい。もう少し幅広いのです。次のようなルールに従って指定された式を `true` と評価します:

- 値が **null** ではない場合:
  - **boolean**の `true`
  - **0**以外の数値
  - **0**以外の文字
  - **“false”** でも **“off”** でも **“no”** でもない文字列
  - 真偽値でも、数値でも、文字でも文字列でもない場合
- **(**値が **null** の場合は **th:if** は **false** と評価します**).**

また、 `th:if` には反対の意味で対になるものがあります。 `th:unless` です。先ほどの例で、**OGNL**式の `not` を使用する代わりに、これを使用することもできます。

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

7.2 スイッチ文

Javaにおける *switch* 構造と同じように使用して、コンテンツを条件毎に表示する方法もあります: `th:switch` / `th:case` 属性のセットです。

ご想像通りの動きをします:

```
<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>
```

一つの `th:case` 属性が `true` と評価されるとすぐに、同じスイッチコンテキスト内の他の全ての `th:case` 属性は `false` と評価されることに注意してください。

デフォルトオプションは `th:case="*"` で指定します:

```
<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 テンプレートレイアウト

8.1 テンプレートフラグメントのインクルード

フラグメントの定義と参照

他のテンプレートのフラグメントを別のテンプレートにインクルードしたいという場合がよくあります。よく使われるのはフッターやヘッダー、メニューなどです。

そうするためにThymeleafではインクルード可能なフラグメントを定義する必要があります。定義には `th:fragment` 属性を使用します。

私たちの食料品店の全てのページに標準的なコピーライトフッターを追加したいとしましょう。 `/WEB-INF/templates/footer.html` ファイルにこのようなコードを定義します:

```
<!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>
```

このコードは `copy` と呼ばれるフラグメントを定義しており、私たちのホームページで `th:include` または `th:replace` 属性のどちらかを使用して簡単にインクルードすることができます:

```
<body>

  ...

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

</body>
```

これらのインクルード属性の構文は両方ともとても直感的です。そのフォーマットには3種類あります:

- `"templatename::domselector"` またはそれと同等の `templatename::[domselector] templatename` という名前のテンプレート内にある、DOMセレクターで指定されたフラグメントをインクルードします。
  - `domselector` はフラグメント名でも大丈夫なので上記の例の `footer :: copy` のように単に `templatename::fragmentname` を指定することもできることに注意してください。

> DOMセレクター構文はXPath表現やCSSセレクターと似ています。詳しくは Appendix C を参照してください。

- `"templatename" templatename` という名前のテンプレート全体をインクルードします。

> `th:include` / `th:replace` タグで使用されるテンプレート名は現在テンプレートエンジンで使用されているテンプレートリゾルバーによって解決可能でなければならないことに注意してください。

- `::domselector"` or `"this::domselector"` 同じテンプレート内のフラグメントをインクルードします。

上記の例の `templatename` と `domselector` には両方とも式を指定することができます(条件式でも大丈夫です！):

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

フラグメントにはどんな `th:*` 属性でも含めることができます。これらの属性は対象テンプレート(`th:include` / `th:replace` 属性が書かれたテンプレートのことです)にそのフラグメントがインクルードされるときに1度評価されます。フラグメント内の属性は、対象テンプレート内のコンテキスト変数を参照することができます。

> フラグメントに対するこのアプローチの大きな利点は、完全かつ妥当なXHTML構造によって、ブラウザで完全に表示できるフラグメントを書くことができるという点です。Thymeleafを使って他のテンプレートにインクルードすることができるのに、です。

`th:fragment` を使用せずにフラグメントを参照する

さらに、DOMセレクターのパワーのお陰で、 `th:fragment` 属性を使わなくてもフラグメントをインクルードすることができます。全くThymeleafのことを知らない別のアプリケーションのマークアップコードでさえもインクルードすることができます。

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

このフラグメントを単に `id` 属性によってCSSセレクターに似た方法で参照することができます。

```
<body>

  ...

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

</body>
```

`th:include` と `th:replace` の違い

では、 `th:include` と `th:replace` の違いって何でしょうか？ `th:include` はホストタグの中にフラグメントの中身をインクルードする一方で `th:replace` は実際にホストタグをフラグメントで置換します。ですので、このようなHTML5フラグメントに対して:

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

...ホストとなる `<div>` タグを2個書いてインクルードしてみます:

```
<body>

  ...

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

</body>
```

...するとこのような結果になります:

```
<body>

  ...

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

</body>
```

`th:substituteby` 属性は `th:replace` 属性に対するエイリアスとして使用できますが、後者を推奨します。`th:substituteby` は将来のバージョンで非推奨になるかもしれないことに注意してください。

8.2 パラメータ化可能なフラグメントシグネチャ

テンプレートフラグメントを、より「関数のような」メカニズムで作成するために `th:fragment` で定義されたフラグメントは、パラメータを持つことができます:

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

`th:include` や `th:replace` からこのフラグメントを呼び出す場合には、以下の2つのどちらかの構文を使用します:

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

後者の場合は、順番が重要ではないことに注意してください:

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

フラグメントシグネチャなしでのフラグメントローカル変数

シグネチャなしでフラグメントが定義されている場合でも:

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

上記の後者の構文を使うことができます(後者の構文だけです):

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

実際のところ、これは `th:include` と `th:with` を組み合わせて使ったのと同じことです:

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

注意 シグネチャの有無に関わらず、フラグメントに対するローカル変数のこの仕様によってコンテキストが実行前に空になるというようなことはありません。この場合でもフラグメントは呼び出し元のテンプレートと同じように、全てのコンテキスト変数にアクセスすることができます。

テンプレート内でのアサーションのための th:assert

`th:assert` 属性に対して、全ての評価が **true** になるはずの式をカンマ区切りのリストで指定すると、もしそうならない場合には例外を投げます。

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

これを使うと、フラグメントシグネチャで簡単にパラメータをバリデートすることができます:

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

8.3 テンプレートフラグメントの削除

私たちの商品リストテンプレートの最新バージョンをもう一度見てみましょう:

```
<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>
```

このコードはテンプレートとしては全然問題ありませんが、静的ページ**(Thymeleaf**で処理をせずに直接ブラウザで開いた場合**)**としては良いプロトタイプではなさそうです。

なぜでしょうか？ブラウザで完全に表示できはしますが、テーブルにはモックデータの**1**行しかないからです。プロトタイプとして単純にリアルさが足りません**... 2**つ以上の商品を表示するほうが良かったですね。複数行必要ですよね。

ということで、追加しましょう:

```
<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>
```

よし、これで**3**商品になったので、プロトタイプとしてはこのほうが全然良いです。でも**...Thymeleaf**で処理したらどうなるでしょうか？:

```
<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>
```

最後の2行がモック行です！あぁ、そりゃそうです: 繰り返し処理は最初の行にしか適用されませんので、Thymeleafが他の2行を削除する理由がありません。

テンプレート処理をする際にこの2行を削除する手段が必要です。 `th:remove` 属性を2つ目と、3つ目の `<tr>` に使用しましょう:

```
<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>
```

テンプレートを処理すると、正しく動くように戻りましたね:

```
<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>
```

この属性に対する `all` という値はどうなっているのでしょう？何を意味するのでしょうか？はい、実際のところ `th:remove` はその値によって、**5**つの異なる振る舞いをします:

- `all`: この属性を含んでいるタグとその全ての子の両方を削除します。
- `body`: この属性を含んでいるタグは削除せずに、全ての子を削除します。
- `tag`: この属性を含んでいるタグは削除しますが、子は削除しません。
- `all-but-first`: 最初の子以外の全ての子を削除します。
- `none` : 何もしません。この値は、動的な評価の場合に有用です。

`all-but-first` 値は何の役に立つのでしょう？それはプロトタイプに書く `th:remove="all"` を減らしてくれます:

```
<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` 属性は許可された文字列値(`all` , `tag` , `body` , `all-but-first` または `none`)を返すのであればどんな「Thymeleafスタンダード式」でも指定することができます。

つまり、削除に条件を適用することもできるということです:

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

また、 `th:remove` は `null` を `none` と同義の別名とみなすため、次のような場合は上記の例と全く同じ動きをします:

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

この場合、 `${condition}` がfalseの場合、 `null` が返されるので、何も削除されません。

9 ローカル変数

Thymeleafではテンプレートの特定のフラグメントに対して定義され、そのフラグメント内でのみ評価可能な変数のことを「ローカル変数」と呼びます。

既に見たことのある例を挙げると、商品リストページの繰り返し変数 `prod` がそれにあたります。

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

`prod` 変数は `<tr>` タグの間だけで有効です。具体的には:

- そのタグの中で `th:each` より優先順位が下の `th:*` 属性全てで使用することができます(優先順位が下の属性とは `th:each` より後に実行される属性という意味です)。
- `<tr>` タグの子要素、例えば `<td>` など、でも使用可能です。

Thymeleafには、繰り返し処理以外でもローカル変数を定義する方法があります。 `th:with` 属性です。その構文は属性値の代入の構文に似ています:

```
<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` が処理されると、 `firstPer` 変数がローカル変数として作成されコンテキストの変数マップに追加されます。そして、コンテキスト内で最初から定義されている他の変数と同様に評価可能になります。ただし、 `<div>` タグの間だけです。

複数の変数を同時に設定したい場合は、普通に複数の代入をする構文を使用することができます:

```
<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` 属性ではその属性内で定義された変数の再利用ができます:

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

それでは私たちの食料品店のホームページで使ってみましょう！日付をフォーマットして出しているコードを覚えていますか？

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

では、実際にこの `"dd MMMM yyyy"` をロケールに合わせたい場合はどうしましょうか？例えば `home_en.properties` に次のようなメッセージを追加したいかもしれません:

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

そして、同様に `home_es.properties` には次のように:

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

さて、 `th:with` を使って、ローカライズされた日付フォーマットを変数に入れて、それを `th:text` 式で使ってみましょう:

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

綺麗で簡単ですね。実は、 `th:with` は `th:text` よりも高い優先順位を持っていますので全部を `span` タグに書くこともできます:

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

優先順位？それまだ知らない！と思ったかもしれませんね。心配しないでください、次の章は優先度についてです。

10 属性の優先順位

同じタグの中に複数の `th:*` 属性を書いた場合には何が起こるのでしょうか？例えば:

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

もちろん、期待した結果を得るためには `th:each` 属性が `th:text` より先に実行されて欲しいですよね。ですが、**DOM(Document Object Model)**標準ではタグの中の属性が書かれている順番には特に意味を持たせていないので、私たちの想定通りに動作することを保証するためには、属性自身に「優先順位」というメカニズムを持たせなければなりません。

ですので、**Thymeleaf**の全ての属性は数値の優先順位を定義しています。その値によってタグの中で実行される順番が決まります。この順番は次の通りです:

順番	機能	属性
1	フラグメントのインクルード	`th:include` `th:replace`
2	フラグメントの繰り返し	`th:each`

3	条件の評価	`th:if`
		`th:unless`
		`th:switch`
		`th:case`

4	ローカル変数の定義	`th:object` `th:with`

5	一般的な属性の変更	`th:attr`
		`th:attrprepend`
		`th:attrappend`

6	特定の属性の変更	`th:value`
		`th:href`
		`th:src`
		`...`

7	テキスト (タグボディの変更)	`th:text` `th:utext`
8	フラグメントの定義	`th:fragment`
9	フラグメントの削除	`th:remove`

この優先順位のメカニズムがあるので、上記の繰り返しのフラグメントで属性の位置を入れ替えても全く同じ結果を得ることができます(少し読みにくくなりますけどね)。

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

11. コメントとブロック

11.1. 標準的なHTML/XMLコメント

標準的なHTML/XMLコメント `<!-- ... -->` はThymeleafテンプレート内のどこでも使用することができます。このコメントの中にあるものは全てThymeleafにもブラウザにも処理されずに、一字一句そのまま単純に結果にコピーされます:

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

11.2. Thymeleafパーサーレベルのコメントブロック

パーサーレベルのコメントブロックはThymeleafがそれをパースする際にテンプレートから削除されます。こんな感じです:

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

Thymeleafは `<!--/*` と `*/-->` の間にあるもの全てを完全に削除するので、このコメントブロックは「テンプレートが静的に開かれた場合にだけ内容を表示する」という用途のために使用することもできます。Thymeleafで処理すると削除されます:

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

これは、例えばたくさんの `<tr>` を持ったテーブルのプロトタイプを作成する際にとても便利かもしれません:

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

11.3. Thymeleafプロトタイプのみのコメントブロック

Thymeleafにはテンプレートが静的に(例えばプロトタイプとして)開かれた場合にはコメントになり、テンプレートとして実行された場合には通常のマークアップとして扱われる特別なコメントブロックがあります。

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

Thymeleafのパースシステムは単純に `<!--/*/` と `/*/-->` のマーカーを削除しますが、コンテンツは削除しないので、そのコンテンツがアンコメントされて残ります。ですので、テンプレートを実行するときには **Thymeleaf**からは実際このように見えます:

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

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

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

パーサーレベルコメントブロックと同様、この機能はダイアレクトからは独立した機能です。

11.4. 擬似的な `th:block` タグ

`th:block` は、**Thymeleaf**のスタンダードダイアレクトに唯一含まれている要素プロセッサ(属性プロセッサではなく)です。

`th:block` は、テンプレート開発者が好きな属性を指定することができるという、ただの属性コンテナにすぎません。**Thymeleaf**は属性を実行して、次に単純にそのブロックを跡形もなく消してしまいます。

ですので例えば、繰り返しを使用したテーブルで各要素に対して1つ以上の `<tr>` が必要な場合に有用でしょう:

```
<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>
```

そしてプロトタイプのみのコメントと組み合わせると特に有用です:

```
<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>
```

この解決策によって、テンプレートが(`<table>` 内で禁止されている `<div>` ブロックを書く必要がなく)妥当なHTMLになっていることに注意してください。また、プロトタイプとしてブラウザで静的に開かれても問題ありません！

12 インライン処理

12.1 テキストのインライン処理

必要なものはほぼ全てスタンダードダイアレクトのタグ属性で実現できますが、HTMLテキストの中に直接式を書きたいというシチュエーションもあります。例えば、こう書くよりは:

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

このように書きたいかも知れません:

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

`[[...]]` の中の式はThymeleafでインライン処理される式と見なされ、 `th:text` 属性で使用することができる式ならどんな種類のものでも使用できます。

インライン処理を動作させるためには `th:inline` 属性を使用してアクティブにしなければなりません。 `th:inline` 属性には3つの値またはモードを指定することができます (`text` と `javascript` と `none`)。

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

`th:inline` を持つタグはインライン式を含んでいるタグ自体である必要はなく、親タグであっても構いません:

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

   ...

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

   ...

</body>
```

そして今こう思っているかもしれません:「どうして最初からこれをしなかったの？この方が `th:text` 属性よりコードが少なくてすむじゃない！」 あぁ、気をつけてくださいね。インライン処理はとても面白いと思ったかもしれませんが、インライン用に書かれた式は静的に開いた場合にはそのままHTMLの中に表示される、ということを覚えておいてください。つまりその場合、プロトタイプとしてはたぶんもう使えないのです！

インライン処理を使用していないフラグメントを静的にブラウザで表示した場合:

```
Hello, Sebastian!
```

そして、インライン処理を使用した場合:

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

ということです。

12.2 スクリプトのインライン処理 (JavaScript と Dart)

Thymeleafのインライン処理機能には「スクリプト」モードがあります。いくつかのスクリプト言語で書かれたスクリプト内にデータを組み込むことができます。

現在のスクリプトモードは `javascript` (`th:inline="javascript"`) と `dart` (`th:inline="dart"`) です。

スクリプトのインライン処理でできることの1つ目は、スクリプト内に式の値を書くことです:

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

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

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

`/*[[...]]*/` 構文は、中の式を評価するようにThymeleafに伝えます。しかし、いくつかポイントがあります:

- Javascriptコメント(`/*...*/`)になっているので、ブラウザで静的にページを開いた場合にはこの式は無視されます。
- インライン式の後ろのコード(`'Sebastian'`)は、静的にページを開いた場合には表示されます。
- Thymeleafは式を実行して結果を挿入しますが、同時にこの行のインライン式の後ろにある全てのコードを削除します(静的に開いた際には表示される部分です)。

ですので、実行結果はこのようになります:

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

    var username = 'John Apricot';

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

コメント化しなくても大丈夫なのですが、それだと静的に読み込んだ場合にスクリプトのエラーになるでしょう:

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

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

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

この評価は賢いので、文字列以外も使用できることに注意してください。Thymeleafは次の種類のオブジェクトをJavascript/Dart構文に正しく書くことができます:

- Strings
- Numbers
- Booleans
- Arrays
- Collections
- Maps
- Beans (objects with *getter* and *setter* methods)

例えば、このようなコードがあったとしましょう:

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

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

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

この `${session.user}` 式によって `User` オブジェクトが評価され、**Thymeleaf**によって正しく**Javascript**構文に変換されます:

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

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

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

コードを追加する

Javascriptインライン処理で使用できるもう**1**つの機能は、 `/*[+...+]*/` という特別なコメント構文で挟まれたコードをインクルードするという機能です。これを使用すると、**Thymeleaf**はテンプレート処理時に自動的にそのコードをアンコメントします:

```
var x = 23;

/*[+

var msg  = 'This is a working application';

+]*/

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

は、このように処理されます:

```
var x = 23;

var msg  = 'This is a working application';

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

このコメントの中には式を含めることができ、**Thymeleaf**で評価されます:

```
var x = 23;

/*[+

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

+]*/

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

コードを削除する

Thymeleafでは `/*[- */` と `/* -]*/` という特別なコメントの間に挟むことでコードを削除することもできます:

```
var x = 23;

/*[- */

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

/* -]*/

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

13 バリデーションとDoctype

13.1 テンプレートをバリデートする

前述の通り、Thymeleafには処理前にテンプレートをバリデートする2つの標準テンプレートモードがあります: `VALIDXML` と `VALIDXHTML` です。これらのモードの場合はテンプレートは「整形式のXML」である(常にそうあるべきですが)というだけでなく、実際に指定された `DTD` に従って妥当である必要があります。

問題は、以下のように `DOCTYPE` 節を含んでいるテンプレートに対して `VALIDXHTML` モードを使用する場合です:

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

これは、バリデーションエラーになるでしょう。 `th:*` タグが `DTD` に存在しないからです。当然ですよね、W3CがThymeleafの機能を標準に入れるわけがないですよね。でも、じゃあどうしましょうか？ `DTD` を変更することによって解決します。

ThymeleafにはXHTML標準のオリジナルをコピーした `DTD` が含まれていて、それらの `DTD` では、スタンダードダイアレクトの全ての `th:*` 属性が使用できるようなっています。そういった理由で、これまでテンプレート内で次のようにしていたのです:

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

`SYSTEM` 識別子はThymeleafパーサーに対して、Thymeleafが用意した特別な `XHTML 1.0 Strict DTD` ファイルを解決して、テンプレートをバリデートするときにそれを使用するように指示します。 `http` の部分に関しては心配しないでください、これはただの識別子であって `DTD` ファイルはThymeleafのjarファイルからローカルで読み込まれます。

> このDOCTYPE宣言は完全に妥当なので、ブラウザで静的にこのテンプレートをプロトタイプとして開いた場合は「標準モード」でレンダリングされることに注意してください。

対応しているXHTML全てに対してThymeleafが用意した `DTD` 定義の一式を記載します:

```
<!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">
```

また、バリデートモードを使用していない場合でも、IDEが幸せになるように `th` ネームスペースを `html` タグに定義しておいてあげると良いです。

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

13.2 Doctype変換

テンプレートに次のように `DOCTYPE` を持つことは良いのですが:

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

この `DOCTYPE` を持った**XHTML**ドキュメントをウェブアプリケーションからクライアントブラウザに送るのは次のような理由から良くありません:

- `PUBLIC` ではないので(`SYSTEM DOCTYPE` なので)、**W3C**のバリデーターでバリデートすることができない。
- 処理後には全ての `th:*` タグはなくなるので、この宣言は不要。

そのため、**Thymeleaf**には「**DOCTYPE** 変換」のメカニズムがあり、自動的に**Thymeleaf**用の**XHTML** `DOCTYPE` を標準の `DOCTYPE` に変換します。

例えば、テンプレートが ***XHTML 1.0 Strict*** で次のような場合:

```
<!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>
```

Thymeleafでテンプレートを処理すると、結果の**XHTML**は次のようになります:

```
<!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>
```

この変換の実行に関しては何もする必要はありません: **Thymeleaf**が自動的に面倒をみてくれます。

14 食料品店用のページをいくつか追加

Thymeleafの使い方について、もうたくさん知っているので、注文管理のための新規ページをいくつか追加することができます。

XHTMLコードにフォーカスしますが、対応するコントローラーを見てみたい場合はバンドルされたソースコードをチェックしてください。

14.1 注文リスト

注文リストページを作成しましょう `/WEB-INF/templates/order/list.html`:

```
<!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>
```

驚くようなことは何もありません。ちょっとした**OGNL**マジックくらいですね:

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

ここでやっているのは、注文の中の各注文行(`OrderLine` オブジェクト)で `purchasePrice` と `amount` プロパティを(対応する `getPurchasePrice()` と `getAmount()` メソッドを呼び出して)掛けあわせて結果を数値のリストに返し、`#aggregates.sum(...)` 関数で集計して注文の合計金額を取得するという処理です。

きっとOGNLのパワーが好きになったでしょう。

14.2 注文詳細

次は、注文詳細ページです。アスタリスク構文を多用します:

```
<!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>
```

ここでも本当に新しいことはありません。このネストされたオブジェクト選択くらいですね:

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

  ...

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

  ...
</body>
```

この `*{name}` は実際には次と同等です:

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

15 設定についてもう少し

15.1 テンプレートリゾルバー

グッドタイムス仮想食料品店では `ITemplateResolver` 実装の `ServletContextTemplateResolver` を選び、テンプレートをサーブレットコンテキストからリソースとして取得しました。

`ITemplateResolver` を実装して独自のテンプレートリゾルバーを作成する以外にもThymeleafにはそのまま使用可能な実装が3つあります:

- `org.thymeleaf.templateresolver.ClassLoaderTemplateResolver` テンプレートをクラスローダーリソースとして解決します:

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

- `org.thymeleaf.templateresolver.FileTemplateResolver` テンプレートをファイルシステムのファイルとして解決します:

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

- `org.thymeleaf.templateresolver.UrlTemplateResolver` テンプレートをURL(ローカル以外でも大丈夫)として解決します:

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

最初からバンドルされている `ITemplateResolver` 実装には、全て同じ設定パラメータを指定することができます。 そのパラメータには次のようなものがあります:

- Prefix と suffix の設定(もう見たことありますね):

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

- テンプレートエイリアス設定。これを使用するとファイル名と一致しないテンプレート名を使用することができます。suffix/prefixとエイリアスが両方指定されている場合はエイリアスがprefix/suffixより前に適用されます:

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

- テンプレートを読み込む際のエンコーディング設定:

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

- デフォルトテンプレートモード設定と、特定のテンプレートに他のモードを指定するためのパターン設定:

```
// デフォルトは TemplateMode.XHTML
templateResolver.setTemplateMode("HTML5");
templateResolver.getXhtmlTemplateModePatternSpec().addPattern("*.xhtml");
```

- テンプレートキャッシュのデフォルトモード設定と、特定のテンプレートにキャッシュするかしないかを指定するための

パターン設定:

```
// デフォルトは true
templateResolver.setCacheable(false);
templateResolver.getCacheablePatternSpec().addPattern("/users/*");
```

- このテンプレートリゾルバーでパースされたテンプレートキャッシュエントリーのTTLをミリセカンドで設定。設定されていない場合は、キャッシュからエントリーが削除されるのはLRU(最大キャッシュサイズを超えた際に一番古いキャッシュエントリーが削除される)のみになります。

```
// デフォルトはTTL指定なし（LRUのみがエントリーを削除）
templateResolver.setCacheTTLMs(60000L);
```

また、テンプレートエンジンには複数のテンプレートリゾルバーを指定することもできます。その場合、テンプレート解決のためにテンプレートリゾルバーは順番付けされ、最初のリゾルバーがテンプレートを解決できない場合には、次のリゾルバーに問い合わせる、といった流れで処理を行います:

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

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

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

複数のテンプレートリゾルバーが適用されている場合には、それぞれのテンプレートリゾルバーにパターンを指定することをお勧めします。そうすることでThymeleafはテンプレートに対して、対象外のリゾルバーを素早く無視することができるので、パフォーマンスが良くなります。必須ということではなく、最適化のためのお勧めです:

```
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 メッセージリゾルバー

私たちの食料品店アプリケーションでは、明示的にメッセージリゾルバーの実装を指定していません。前述の通り、この場合は `org.thymeleaf.messageresolver.StandardMessageResolver` オブジェクトが使用されています。

この `StandardMessageResolver` は、既に説明した通りテンプレートと同じ名前のメッセージファイルを探しますが、実際のところThymeleafのコアがそのまま使えるように用意している唯一のメッセージリゾルバーです。ですがもちろん、`org.thymeleaf.messageresolver.IMessageResolver` を実装すればあなた独自のメッセージリゾルバーを作成することができます。

> Thymeleaf + Spring 連携パッケージでは `IMessageResolver` の実装が提供されていて、そのリゾルバーは標準的なSpringの方法で、 `MessageSource` オブジェクトを使用して外部化されたメッセージを取得します。

テンプレートエンジンにメッセージリゾルバーを1つ(または複数)指定したい場合はどうすれば良いでしょうか？簡単です:

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

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

でも、どうして複数のメッセージリゾルバーを指定したいのでしょうか？テンプレートリゾルバーと同じ理由ですね: メッセージリゾルバーは順番付けされて、最初のリゾルバーがあるメッセージを解決できなければ、次のリゾルバーに問い合わせて、その次は3番目に・・・となります。

15.3 ロギング

Thymeleafはロギングにもかなり気を使っていて、いつでもロギングインターフェイスを通して最大限の有用な情報を提供しようとしています。

ロギングライブラリは `slf4j` を使用しています。 `slf4j` は実際にアプリケーションで使用しているどんなロギング実装(例えば `log4j`)に対してもブリッジとして振る舞います。

Thymeleafのクラスは、詳細レベルに合わせて `TRACE` と `DEBUG` と `INFO` レベルのログを出力します。そして一般的なロギングとは別に、 **TemplateEngine** クラスに関連付けられた3つの特別なロガーがあり目的に合わせて個別に設定をすることができます。

- `org.thymeleaf.TemplateEngine.CONFIG` は、ライブラリの初期化時に設定の詳細を出力します。
- `org.thymeleaf.TemplateEngine.TIMER` は、それぞれのテンプレートを処理する際にかかった時間を出力します(ベンチマークに便利です！)。
- `org.thymeleaf.TemplateEngine.cache` は、キャッシュに関する特定の情報を出力するロガーのプレフィックスになっています。ユーザーがキャッシュロガーの名前を設定することができるので、名前は変わりうるのですが、デフォルトでは:
  - `org.thymeleaf.TemplateEngine.cache.TEMPLATE_CACHE`
  - `org.thymeleaf.TemplateEngine.cache.FRAGMENT_CACHE`
  - `org.thymeleaf.TemplateEngine.cache.MESSAGE_CACHE`
  - `org.thymeleaf.TemplateEngine.cache.EXPRESSION_CACHE`

Thymeleafのロギングインフラのための設定例は `log4j` を使用する場合、次のようになります:

```
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 テンプレートキャッシュ

ThymeleafはDOM処理エンジンと一連のプロセッサ — ロジックを適用する必要があるノードのタイプごとに1つ — のおかげで動いています。プロセッサは、ドキュメントのDOMツリーとデータを結びつけることによって期待する結果を作成するために、DOMツリーに変更を加えます。

また、 — デフォルトで — パースしたテンプレートをキャッシュする機能があります。テンプレートファイルを読み込んでパースした結果の、処理前のDOMツリーをキャッシュします。これは特に以下のようなコンセプトに基づいて作成されているウェブアプリケーションに役立ちます:

- Input/Output が、いつでもどんなアプリケーションにとっても最も遅い部分である。インメモリ処理の方が全然速い。
- インメモリのDOMツリーをクローンする方が、テンプレートファイルを読み込んでパースして新しいDOMツリーを生成するよりも断然速い。
- ウェブアプリケーションは通常数十個のテンプレートしか使わない。
- テンプレートファイルは小-中程度のサイズであって、アプリケーションの実行中には変更されない。

このことから、ウェブアプリケーションで最も使用されているテンプレートをキャッシュすることで、大量のメモリを無駄に使うこともなくうまくいきそうですし、実際には決して変更されない少ないファイルのIO処理に費やされるたくさんの時間を節約することができると考えられます。

このキャッシュをどのようにコントロールすることができるのでしょうか？まず、テンプレートリゾルバーで有効/無効の切り替えをすることができ、特定のテンプレートにだけ適用することもできるということは学びましたね。

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

また、独自の「キャッシュマネージャ」を作成することで設定を変更することもできます。デフォルトの `StandardCacheManager` 実装のインスタンスを使用することも可能です。

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

キャッシュの設定についてのより詳しい情報は `org.thymeleaf.cache.StandardCacheManager` のJavadoc APIを参照してください。

テンプレートキャッシュから手動でエントリーを削除することもできます:

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

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

17 Appendix A: Expression Basic Objects

(OGNLやSpringELによって実行される)変数式の中で、常に使用可能なオブジェクトや変数マップがあります。それを見てみましょう:

基本オブジェクト

- **#ctx** : コンテキストオブジェクト。環境(スタンドアローンかウェブか)によって `org.thymeleaf.context.IContext` や `org.thymeleaf.context.IWebContext` の実装になります。 *Spring連携モジュール* を使用している場合は、`org.thymeleaf.spring[3|4].context.SpringWebContext` のインスタンスになります。

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.IContext
 * ======================================================================
 */

${#ctx.locale}
${#ctx.variables}

/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.IWebContext
 * ======================================================================
 */

${#ctx.applicationAttributes}
${#ctx.httpServletRequest}
${#ctx.httpServletResponse}
${#ctx.httpSession}
${#ctx.requestAttributes}
${#ctx.requestParameters}
${#ctx.servletContext}
${#ctx.sessionAttributes}
```

- **#locale** : 現在のリクエストに関連付けられている `java.util.Locale` への直接アクセス。

```
${#locale}
```

- **#vars** : コンテキスト内の全ての変数を持った `org.thymeleaf.context.VariablesMap` のインスタンス(通常は `#ctx.variables` に含まれている変数にローカル変数を加えたものです)。

  限定子がついていない式はこのオブジェクトに対して評価されます。実際のところ `${something}` は `${#vars.something}` と完全に同等です(がより美しいです)。

  `#root` はこのオブジェクトの同意語です。

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.VariablesMap
 * ======================================================================
 */

${#vars.get('foo')}
${#vars.containsKey('foo')}
${#vars.size()}
...
```

request/session 属性などに対するウェブコンテキストネームスペース

ウェブ環境で**Thymeleaf**を使っている場合、リクエストパラメータ、セッション属性、アプリケーション属性にアクセスするのにショートカットを使用することができます。

> これらは「コンテキストオブジェクト」ではなく、コンテキストに対して変数として追加されたマップです。ですので `#` を使いません。そのため、ある意味で「名前空間」のように振る舞います。

- **param :** リクエストパラメータを取得するために使用します。 `${param.foo}` は `foo` リクエストパラメータの値を持つ `String[]` です。ですので、最初の値を取得するために普通は `${param.foo[0]}` を使用します。

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.WebRequestParamsVariablesMap
 * ======================================================================
 */

${param.foo}              // Retrieves a String[] with the values of request parameter 'foo'
${param.size()}
${param.isEmpty()}
${param.containsKey('foo')}
...
```

- **session :** セッション属性を取得するために使用します。

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.WebSessionVariablesMap
 * ======================================================================
 */

${session.foo}                // Retrieves the session atttribute 'foo'
${session.size()}
${session.isEmpty()}
${session.containsKey('foo')}
...
```

- **application :** アプリケーション/サーブレットコンテキストを取得するために使用します。

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.context.WebServletContextVariablesMap
 * ======================================================================
 */

${application.foo}              // Retrieves the ServletContext atttribute 'foo'
${application.size()}
${application.isEmpty()}
${application.containsKey('foo')}
...
```

リクエスト属性にアクセスする際には(リクエストパラメータとは対照的に)名前空間を指定する必要がない ことに注意してください。なぜなら、全てのリクエスト属性は自動的にコンテキストルートの変数としてコンテキストに追加されるからです:

```
${myRequestAttribute}
```

ウェブコンテキストオブジェクト

ウェブ環境の場合は、次のようなオブジェクトにも直接アクセスすることができます(これらはオブジェクトであって、マップや名前空間ではないことに注意して下さい):

- **#httpServletRequest** : 現在のリクエストに関連付けられた `javax.servlet.http.HttpServletRequest` オブジェクトへの直接アクセス

```
${#httpServletRequest.getAttribute('foo')}
${#httpServletRequest.getParameter('foo')}
${#httpServletRequest.getContextPath()}
${#httpServletRequest.getRequestName()}
...
```

- **#httpSession** : 現在のリクエストに関連付けられた `javax.servlet.http.HttpSession` オブジェクトへの直接アクセス。

```
${#httpSession.getAttribute('foo')}
${#httpSession.id}
${#httpSession.lastAccessedTime}
...
```

Springコンテキストオブジェクト

SpringからThymeleafを使用している場合は、これらのオブジェクトにもアクセスできます:

- **#themes** : Springの `spring:theme` JSPタグと同じ機能を提供します。

```
${#themes.code('foo')}
```

Springビーン

Thymeleafでは、SpringELによってSpringアプリケーションコンテキストに通常の方法で定義されて登録されたビーンに `@beanName` シンタックスを使用してアクセスすることができます。例:

```
<div th:text="${@authService.getUserName()}">...</div>
```

18 Appendix B: Expression Utility Objects

日付

- **#dates**: `java.util.Date` オブジェクトに対するユーティリティメソッド群:

```
/*
 * =====================================================================
 * See javadoc API for class org.thymeleaf.expression.Dates
 * =====================================================================
 */

/*
 * 標準ロケールフォーマットで日付をフォーマットします
 * 配列、リスト、セットにも対応しています
 */
${#dates.format(date)}
${#dates.arrayFormat(datesArray)}
${#dates.listFormat(datesList)}
${#dates.setFormat(datesSet)}

/*
 * ISO8601フォーマットで日付をフォーマットします
 * 配列、リスト、セットにも対応しています
 */
${#dates.formatISO(date)}
${#dates.arrayFormatISO(datesArray)}
${#dates.listFormatISO(datesList)}
${#dates.setFormatISO(datesSet)}

/*
 * 指定されたパターンで日付をフォーマットします
 * 配列、リスト、セットにも対応しています
 */
${#dates.format(date, 'dd/MMM/yyyy HH:mm')}
${#dates.arrayFormat(datesArray, 'dd/MMM/yyyy HH:mm')}
${#dates.listFormat(datesList, 'dd/MMM/yyyy HH:mm')}
${#dates.setFormat(datesSet, 'dd/MMM/yyyy HH:mm')}

/*
 * 日付のプロパティを取得します
 * 配列、リスト、セットにも対応しています
 */
${#dates.day(date)}                     // also arrayDay(...), listDay(...), etc.
${#dates.month(date)}                   // also arrayMonth(...), listMonth(...), etc.
${#dates.monthName(date)}               // also arrayMonthName(...), listMonthName(...), etc.
${#dates.monthNameShort(date)}          // also arrayMonthNameShort(...), listMonthNameShort(...), etc.
${#dates.year(date)}                    // also arrayYear(...), listYear(...), etc.
${#dates.dayOfWeek(date)}               // also arrayDayOfWeek(...), listDayOfWeek(...), etc.
${#dates.dayOfWeekName(date)}           // also arrayDayOfWeekName(...), listDayOfWeekName(...), etc.
${#dates.dayOfWeekNameShort(date)}      // also arrayDayOfWeekNameShort(...), listDayOfWeekNameShort(...), etc
${#dates.hour(date)}                    // also arrayHour(...), listHour(...), etc.
${#dates.minute(date)}                  // also arrayMinute(...), listMinute(...), etc.
${#dates.second(date)}                  // also arraySecond(...), listSecond(...), etc.
${#dates.millisecond(date)}             // also arrayMillisecond(...), listMillisecond(...), etc.

/*
 * コンポーネントを指定して日付オブジェクト(java.util.Date)を作成します
 */
${#dates.create(year,month,day)}
${#dates.create(year,month,day,hour,minute)}
```

```
${#dates.create(year,month,day,hour,minute,second)}
${#dates.create(year,month,day,hour,minute,second,millisecond)}

/*
 * 現在日時の日付オブジェクト(java.util.Date)を作成します
 */
${#dates.createNow()}

/*
 * 現在日の日付のオブジェクト(java.util.Date)を作成します(時間は00:00に設定されます)
 */
${#dates.createToday()}
```

カレンダー

- **#calendars**: `#dates` に似ていますが、 `java.util.Calendar` オブジェクト用です:

```
/*
 * =========================================================================
 * See javadoc API for class org.thymeleaf.expression.Calendars
 * =========================================================================
 */

/*
 * 標準ロケールフォーマットでカレンダーをフォーマットします
 * 配列、リスト、セットにも対応しています
 */
${#calendars.format(cal)}
${#calendars.arrayFormat(calArray)}
${#calendars.listFormat(calList)}
${#calendars.setFormat(calSet)}

/*
 * ISO8601フォーマットでカレンダーをフォーマットします
 * 配列、リスト、セットにも対応しています
 */
${#calendars.formatISO(cal)}
${#calendars.arrayFormatISO(calArray)}
${#calendars.listFormatISO(calList)}
${#calendars.setFormatISO(calSet)}

/*
 * 指定されたパターンでカレンダーをフォーマットします
 * 配列、リスト、セットにも対応しています
 */
${#calendars.format(cal, 'dd/MMM/yyyy HH:mm')}
${#calendars.arrayFormat(calArray, 'dd/MMM/yyyy HH:mm')}
${#calendars.listFormat(calList, 'dd/MMM/yyyy HH:mm')}
${#calendars.setFormat(calSet, 'dd/MMM/yyyy HH:mm')}

/*
 * カレンダーのプロパティを取得します
 * 配列、リスト、セットにも対応しています
 */
${#calendars.day(date)}                // also arrayDay(...), listDay(...), etc.
${#calendars.month(date)}              // also arrayMonth(...), listMonth(...), etc.
${#calendars.monthName(date)}          // also arrayMonthName(...), listMonthName(...), etc.
${#calendars.monthNameShort(date)}     // also arrayMonthNameShort(...), listMonthNameShort(...), etc.
${#calendars.year(date)}               // also arrayYear(...), listYear(...), etc.
${#calendars.dayOfWeek(date)}          // also arrayDayOfWeek(...), listDayOfWeek(...), etc.
${#calendars.dayOfWeekName(date)}      // also arrayDayOfWeekName(...), listDayOfWeekName(...), etc.
${#calendars.dayOfWeekNameShort(date)} // also arrayDayOfWeekNameShort(...), listDayOfWeekNameShort(...), etc
${#calendars.hour(date)}               // also arrayHour(...), listHour(...), etc.
```

```
${#calendars.hour(date)}                // also arrayHour(...), listHour(...), etc.
${#calendars.minute(date)}              // also arrayMinute(...), listMinute(...), etc.
${#calendars.second(date)}              // also arraySecond(...), listSecond(...), etc.
${#calendars.millisecond(date)}         // also arrayMillisecond(...), listMillisecond(...), etc.

/*
 * コンポーネントを指定してカレンダーオブジェクト(java.util.Calendar)を作成します
 */
${#calendars.create(year,month,day)}
${#calendars.create(year,month,day,hour,minute)}
${#calendars.create(year,month,day,hour,minute,second)}
${#calendars.create(year,month,day,hour,minute,second,millisecond)}

/*
 * 現在日時のカレンダーオブジェクト(java.util.Calendar)を作成します
 */
${#calendars.createNow()}

/*
 * 現在日のカレンダーのオブジェクト(java.util.Calendar)を作成します(時間は00:00に設定されます)
 */
${#calendars.createToday()}
```

数値

- **#numbers :** 数値オブジェクトに対するユーティリティメソッド群:

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.expression.Numbers
 * ======================================================================
 */

/*
 * ==========================
 * 整数値のフォーマット
 * ==========================
 */

/*
 * 整数の最小桁数を設定します。
 * 配列、リスト、セットにも対応しています
 */
${#numbers.formatInteger(num,3)}
${#numbers.arrayFormatInteger(numArray,3)}
${#numbers.listFormatInteger(numList,3)}
${#numbers.setFormatInteger(numSet,3)}


/*
 * 整数の最小桁数と千の位の区切り文字を設定します:
 * 'POINT', 'COMMA', 'WHITESPACE', 'NONE' または 'DEFAULT' (ロケールに依存)。
 * 配列、リスト、セットにも対応しています
 */
${#numbers.formatInteger(num,3,'POINT')}
${#numbers.arrayFormatInteger(numArray,3,'POINT')}
${#numbers.listFormatInteger(numList,3,'POINT')}
${#numbers.setFormatInteger(numSet,3,'POINT')}


/*
 * ==========================
 * 小数値のフォーマット
```

```
 * ========================
 */

/*
 * 整数の最小桁数と小数桁数を設定します。
 * 配列、リスト、セットにも対応しています
 */
${#numbers.formatDecimal(num,3,2)}
${#numbers.arrayFormatDecimal(numArray,3,2)}
${#numbers.listFormatDecimal(numList,3,2)}
${#numbers.setFormatDecimal(numSet,3,2)}

/*
 * 整数の最小桁数と小数桁数と小数点の文字を設定します。
 * 配列、リスト、セットにも対応しています
 */
${#numbers.formatDecimal(num,3,2,'COMMA')}
${#numbers.arrayFormatDecimal(numArray,3,2,'COMMA')}
${#numbers.listFormatDecimal(numList,3,2,'COMMA')}
${#numbers.setFormatDecimal(numSet,3,2,'COMMA')}

/*
 * 整数の最小桁数と小数桁数と小数点の文字と千の位の区切り文字を設定します。
 * 配列、リスト、セットにも対応しています
 */
${#numbers.formatDecimal(num,3,'POINT',2,'COMMA')}
${#numbers.arrayFormatDecimal(numArray,3,'POINT',2,'COMMA')}
${#numbers.listFormatDecimal(numList,3,'POINT',2,'COMMA')}
${#numbers.setFormatDecimal(numSet,3,'POINT',2,'COMMA')}



/*
 * ========================
 * ユーティリティメソッド
 * ========================
 */

/*
 * xからyまでの整数のシーケンス(配列)を作成します
 */
${#numbers.sequence(from,to)}
${#numbers.sequence(from,to,step)}
```

文字列

- **#strings**: `String` オブジェクトに対するユーティリティメソッド群:

```
/*
 * ======================================================================
 * See javadoc API for class org.thymeleaf.expression.Strings
 * ======================================================================
 */

/*
 * Null安全な toString()
 */
${#strings.toString(obj)}                           // array*, list* and set* にも対応しています

/*
 * 文字列が空(またはnull)かどうかをチェックします。チェック前に trim() 処理をします。
 * 配列、リスト、セットにも対応しています
 */
```

```
${#strings.isEmpty(name)}
${#strings.arrayIsEmpty(nameArr)}
${#strings.listIsEmpty(nameList)}
${#strings.setIsEmpty(nameSet)}

/*
 * 'isEmpty()' を実行して false の場合はその文字列を返し、true の場合は指定されたデフォルト文字列を返します
 * 配列、リスト、セットにも対応しています
 */
${#strings.defaultString(text,default)}
${#strings.arrayDefaultString(textArr,default)}
${#strings.listDefaultString(textList,default)}
${#strings.setDefaultString(textSet,default)}

/*
 * 文字列にフラグメントが含まれているかどうかをチェックします
 * 配列、リスト、セットにも対応しています
 */
${#strings.contains(name,'ez')}                     // also array*, list* and set*
${#strings.containsIgnoreCase(name,'ez')}           // also array*, list* and set*

/*
 * 文字列が指定されたフラグメントで始まっているかどうかまたは終わっているかどうかをチェックします
 * 配列、リスト、セットにも対応しています
 */
${#strings.startsWith(name,'Don')}                  // also array*, list* and set*
${#strings.endsWith(name,endingFragment)}           // also array*, list* and set*

/*
 * 部分文字列関係
 * 配列、リスト、セットにも対応しています
 */
${#strings.indexOf(name,frag)}                      // also array*, list* and set*
${#strings.substring(name,3,5)}                     // also array*, list* and set*
${#strings.substringAfter(name,prefix)}             // also array*, list* and set*
${#strings.substringBefore(name,suffix)}            // also array*, list* and set*
${#strings.replace(name,'las','ler')}               // also array*, list* and set*

/*
 * Append と prepend
 * 配列、リスト、セットにも対応しています
 */
${#strings.prepend(str,prefix)}                     // also array*, list* and set*
${#strings.append(str,suffix)}                      // also array*, list* and set*

/*
 * 大文字小文字変換
 * 配列、リスト、セットにも対応しています
 */
${#strings.toUpperCase(name)}                       // also array*, list* and set*
${#strings.toLowerCase(name)}                       // also array*, list* and set*

/*
 * Split と join
 */
${#strings.arrayJoin(namesArray,',')}
${#strings.listJoin(namesList,',')}
${#strings.setJoin(namesSet,',')}
${#strings.arraySplit(namesStr,',')}                // returns String[]
${#strings.listSplit(namesStr,',')}                 // returns List<String>
${#strings.setSplit(namesStr,',')}                  // returns Set<String>

/*
 * Trim
 * 配列、リスト、セットにも対応しています
 */
${#strings.trim(str)}                               // also array*, list* and set*
```

```
/*
 * 長さの計算
 * 配列、リスト、セットにも対応しています
 */
${#strings.length(str)}                              // also array*, list* and set*

/*
 * 与えられたテキストが最大サイズnになるよう省略処理をします。
 * もしテキストがそれよりも大きい場合は、切り取られて最後に "..." がつきます。
 * 配列、リスト、セットにも対応しています
 */
${#strings.abbreviate(str,10)}                       // also array*, list* and set*

/*
 * 最初の文字を大文字に変換(とその逆)
 */
${#strings.capitalize(str)}                          // also array*, list* and set*
${#strings.unCapitalize(str)}                        // also array*, list* and set*

/*
 * 単語の最初の文字を大文字に変換
 */
${#strings.capitalizeWords(str)}                     // also array*, list* and set*
${#strings.capitalizeWords(str,delimiters)}          // also array*, list* and set*

/*
 * 文字列のエスケープ
 */
${#strings.escapeXml(str)}                           // also array*, list* and set*
${#strings.escapeJava(str)}                          // also array*, list* and set*
${#strings.escapeJavaScript(str)}                    // also array*, list* and set*
${#strings.unescapeJava(str)}                        // also array*, list* and set*
${#strings.unescapeJavaScript(str)}                  // also array*, list* and set*

/*
 * Null安全な比較と連結
 */
${#strings.equals(first, second)}
${#strings.equalsIgnoreCase(first, second)}
${#strings.concat(values...)}
${#strings.concatReplaceNulls(nullValue, values...)}

/*
 * Random
 */
${#strings.randomAlphanumeric(count)}
```

オブジェクト

- **#objects**: 一般的なオブジェクトに対するユーティリティメソッド群

```
/*
 * =============================================================================
 * See javadoc API for class org.thymeleaf.expression.Objects
 * =============================================================================
 */

/*
 * null でなければ obj を、null の場合は指定されたデフォルト値を返します
 * 配列、リスト、セットにも対応しています
 */
${#objects.nullSafe(obj,default)}
${#objects.arrayNullSafe(objArray,default)}
${#objects.listNullSafe(objList,default)}
${#objects.setNullSafe(objSet,default)}
```

真偽値

- **#bools :** 真偽値評価に対するユーティリティメソッド群

```
/*
 * =============================================================================
 * See javadoc API for class org.thymeleaf.expression.Bools
 * =============================================================================
 */

/*
 * th:if タグと同じように条件を評価します(条件の評価の章を後で参照してください)。
 * 配列、リスト、セットにも対応しています
 */
${#bools.isTrue(obj)}
${#bools.arrayIsTrue(objArray)}
${#bools.listIsTrue(objList)}
${#bools.setIsTrue(objSet)}

/*
 * 否定の評価
 * 配列、リスト、セットにも対応しています
 */
${#bools.isFalse(cond)}
${#bools.arrayIsFalse(condArray)}
${#bools.listIsFalse(condList)}
${#bools.setIsFalse(condSet)}

/*
 * 評価してAND演算子を適用
 * 配列、リスト、セットをパラメータとして受け取ります
 */
${#bools.arrayAnd(condArray)}
${#bools.listAnd(condList)}
${#bools.setAnd(condSet)}

/*
 * 評価してOR演算子を適用
 * 配列、リスト、セットをパラメータとして受け取ります
 */
${#bools.arrayOr(condArray)}
${#bools.listOr(condList)}
${#bools.setOr(condSet)}
```

配列

- **#arrays** : 配列に対するユーティリティメソッド群

```
/*
 * =========================================================================
 * See javadoc API for class org.thymeleaf.expression.Arrays
 * =========================================================================
 */

/*
 * コンポーネントクラスを推測して配列に変換します。
 * 結果の配列が空もしくは、対象オブジェクトに複数のクラスが含まれる場合 Object[] を変えします。
 */
${#arrays.toArray(object)}

/*
 * コンポーネントクラスを指定して配列に変換
 */
${#arrays.toStringArray(object)}
${#arrays.toIntegerArray(object)}
${#arrays.toLongArray(object)}
${#arrays.toDoubleArray(object)}
${#arrays.toFloatArray(object)}
${#arrays.toBooleanArray(object)}

/*
 * 長さを計算
 */
${#arrays.length(array)}

/*
 * 配列が空かどうかをチェック
 */
${#arrays.isEmpty(array)}

/*
 * 1つまたは複数の要素が配列に含まれているかどうかをチェック
 */
${#arrays.contains(array, element)}
${#arrays.containsAll(array, elements)}
```

リスト

- **#lists** : リストに対するユーティリティメソッド群

```
/*
 * ============================================================================
 * See javadoc API for class org.thymeleaf.expression.Lists
 * ============================================================================
 */

/*
 * リストに変換
 */
${#lists.toList(object)}

/*
 * サイズを計算
 */
${#lists.size(list)}

/*
 * リストが空かどうかをチェック
 */
${#lists.isEmpty(list)}

/*
 * 1つまたは複数の要素がリストに含まれているかどうかをチェック
 */
${#lists.contains(list, element)}
${#lists.containsAll(list, elements)}

/*
 * 与えられたリストのコピーをソート。リストのメンバーが comparable を実装しているか
 * または comparator が指定されている必要があります。
 */
${#lists.sort(list)}
${#lists.sort(list, comparator)}
```

セット

- **#sets**: セットに対するユーティリティメソッド群

```
/*
 * ============================================================================
 * See javadoc API for class org.thymeleaf.expression.Sets
 * ============================================================================
 */

/*
 * セットに変換
 */
${#sets.toSet(object)}

/*
 * サイズを計算
 */
${#sets.size(set)}

/*
 * セットが空かどうかをチェック
 */
${#sets.isEmpty(set)}

/*
 * 1つまたは複数の要素がセットに含まれているかどうかをチェック
 */
${#sets.contains(set, element)}
${#sets.containsAll(set, elements)}
```

マップ

- **#maps**: マップに対するユーティリティメソッド群

```
/*
 * ============================================================================
 * See javadoc API for class org.thymeleaf.expression.Maps
 * ============================================================================
 */

/*
 * サイズを計算
 */
${#maps.size(map)}

/*
 * マップが空かどうかをチェック
 */
${#maps.isEmpty(map)}

/*
 * キーや値がマップに含まれているかどうかをチェック
 */
${#maps.containsKey(map, key)}
${#maps.containsAllKeys(map, keys)}
${#maps.containsValue(map, value)}
${#maps.containsAllValues(map, value)}
```

集約

- **#aggregates**: 配列やコレクションに対する集約を生成するユーティリティメソッド群

```
/*
 * ============================================================================
 * See javadoc API for class org.thymeleaf.expression.Aggregates
 * ============================================================================
 */

/*
 * 合計値を計算。配列またはコレクションが空の場合は null を返します
 */
${#aggregates.sum(array)}
${#aggregates.sum(collection)}

/*
 * 平均を計算。配列またはコレクションが空の場合は null を返します
 */
${#aggregates.avg(array)}
${#aggregates.avg(collection)}
```

メッセージ

- **#messages :** 変数式の中で外部化メッセージを取得するためのユーティリティメソッド群。 `#{...}` 構文を使用して取得するのと同じです。

```
/*
 * ============================================================================
 * See javadoc API for class org.thymeleaf.expression.Messages
 * ============================================================================
 */

/*
 * 外部化メッセージを取得します。単一のキー、単一のキーと引数、
 * キーの配列/リスト/セット(この場合は外部化メッセージの配列/リスト/セットを返します)を渡すことができます。
 * メッセージが見つからない場合は、デフォルトメッセージ( '??msgKey??' など)を返します。
 */
${#messages.msg('msgKey')}
${#messages.msg('msgKey', param1)}
${#messages.msg('msgKey', param1, param2)}
${#messages.msg('msgKey', param1, param2, param3)}
${#messages.msgWithParams('msgKey', new Object[] {param1, param2, param3, param4})}
${#messages.arrayMsg(messageKeyArray)}
${#messages.listMsg(messageKeyList)}
${#messages.setMsg(messageKeySet)}

/*
 * 外部化メッセージまたは null を取得します。指定されたキーに対するメッセージが見つからない場合に
 * デフォルトメッセージの代わりに null を返します。
 */
${#messages.msgOrNull('msgKey')}
${#messages.msgOrNull('msgKey', param1)}
${#messages.msgOrNull('msgKey', param1, param2)}
${#messages.msgOrNull('msgKey', param1, param2, param3)}
${#messages.msgOrNullWithParams('msgKey', new Object[] {param1, param2, param3, param4})}
${#messages.arrayMsgOrNull(messageKeyArray)}
${#messages.listMsgOrNull(messageKeyList)}
${#messages.setMsgOrNull(messageKeySet)}
```

ID

- **#ids :** (繰り返し処理の中などで)繰り返し登場する `id` 属性を扱うためのユーティリティメソッド群

```
/*
 * ============================================================================
 * See javadoc API for class org.thymeleaf.expression.Ids
 * ============================================================================
 */

/*
 * 通常は th:id 属性に使用されます。
 * id属性値にカウンターの値を加えるので、繰り返し処理の中でもユニークな値を持つことができます。
 */
${#ids.seq('someId')}

/*
 * 通常は <label> タグの中の th:for 属性に使用されます。
 * #ids.seq(...) 関数で生成されたidをラベルから参照することができます。
 *
 * <label> が #ids.seq(...) 関数を持った要素の前にあるか後ろにあるかによって、
 * "next" (ラベルが"seq"の前の場合) または "prev" (ラベルが"seq"の後の場合) 関数を呼び出します。
 */
${#ids.next('someId')}
${#ids.prev('someId')}
```

19 Appendix C: DOM Selector syntax

DOMセレクターはXPATHやCSSやjQueryの構文を参考にして、テンプレートフラグメントを特定するための簡単かつパワフルな方法を提供しています。

例えば、次のセレクターはマークアップの中で `content` クラスを持った `<div>` を全て取得します:

```
<div th:include="mytemplate :: [//div[@class='content']]">...</div>
```

XPathを参考にした基本構文には次のようなものがあります:

- `/x` 現在のノードの直接の子の中でxという名前を持つノード。
- `//x` 現在のノードの子孫の中でxという名前を持つノード。
- `x[@z="v"]` xという名前の要素で、z属性の値が"v"のもの。
- `x[@z1="v1" and @z2="v2"]` xという名前の要素で、z1,z2属性の値がそれぞれ"v1","v2"のもの。
- `x[i]` xという名前の要素の兄弟の中でi番目のもの。
- `x[@z="v"][i]` xという名前の要素で、z属性の値が"v"の兄弟の中でi番目のもの。

ですが、もっと簡潔な構文もあります:

- `x` は `//x` と全く同じ意味です(深さに関係なく `x` という名前または参照を持つ要素を探します)。
- 引数を持つ場合は要素名や参照を指定しなくても大丈夫です。ですので `[@class='oneclass']` は、class属性の値が"oneclass"の要素(タグ)を探す、という意味の有効なセレクターになります。

高度な属性選択機能:

- `=` (equal)の他にも比較演算子が使用できます: `!=` (not equal), `^=` (starts with) と `$=` (ends with)。例: `x[@class^='section']` は `x` という名前の要素で `class` 属性の値が `section` で始まっているものを指します。
- 属性の指定は `@` で始まっていても(XPath-style)、始まっていなくても(jQuery-style)大丈夫です。ですので、 `x[z='v']` は `x[@z='v']` と同じ意味になります。
- 複数属性を指定する場合は `and` でつないでも(XPath-style)、複数の修飾子をつないでも(jQuery-style)大丈夫です。ですので、 `x[@z1='v1' and @z2='v2']` と `x[@z1='v1'][@z2='v2']` は同じ意味になります(`x[z1='v1'][z2='v2']` もです)。

「jQueryのような」ダイレクトセレクター:

- `x.oneclass` は `x[class='oneclass']` と同等です。
- `.oneclass` は `[class='oneclass']` と同等です。
- `x#oneid` は `x[id='oneid']` と同等です。
- `#oneid` は `[id='oneid']` と同等です。
- `x%oneref` は、xという名前を持った-要素だけではなく-ノードの中で、指定された `DOMSelector.INodeReferenceChecker` 実装に従って *oneref* という参照に一致するもの指します。
- `%oneref` は、名前に関係なく-要素だけではなく-ノードの中で、指定された `DOMSelector.INodeReferenceChecker` 実装に従って *oneref* という参照に一致するもの指します。参照は要素名の代わりに使用されるので、実際は単に `oneref` と同等であることに注意してください。
- ダイレクトセレクターと属性セレクターは混ぜることができます: `a.external[@href^='https']`.

上記のDOMセレクター式は:

```
<div th:include="mytemplate :: [//div[@class='content']]">...</div>
```

このように書くことができます:

```
<div th:include="mytemplate :: [div.content]">...</div>
```

複数の値を持つclassのマッチング

DOMセレクターは **複数の値を持った** class属性に対応しているので、要素がいくつかのclass値を持っている場合でも、セレクターを適用することができます。

例えば、 `div[class='two']` は `<div class="one two three" />` にマッチします。

任意の括弧

フラグメントインクルード属性の構文は全てのフラグメント選択をDOM選択に変換するので、括弧 `[...]` はなくても大丈夫です(あってもいいですが)。

なので、次のように括弧を付けなくても、上記の括弧をつけたものと同等になります:

```
<div th:include="mytemplate :: div.content">...</div>
```

ですので、まとめると:

```
<div th:replace="mytemplate :: myfrag">...</div>
```

これは `th:fragment="myfrag"` フラグメントシグネチャを探します。しかし、(HTMLには存在しませんが)もし存在するならば `myfrag` という名前のタグも探します。次の違いに気をつけてください:

```
<div th:replace="mytemplate :: .myfrag">...</div>
```

この場合は実際 `class="myfrag"` の要素を探しますが、 `th:fragment` シグネチャについては気にしません。

---

1. application/xhtml+xmlコンテンツタイプで取り扱われるXML整形式のHTML5はXHTML5と呼ばれるので、Thymeleafは XHTML5をサポートしていると言ってもいいかもしれません。
2. このテンプレートは妥当なXHTMLですが、テンプレートモードとしては “VALIDXHTML” ではなく “XHTML” を選んでいます。ですので今のところ、バリデーションを気にしなくても問題ないのですが、そうはいってもIDEにたくさん指摘されるのも嫌ですよね。
3. 訳注: iterated expression の適当な訳が分かりませんでした...