



Folha de tomilho

Tutorial: Usando Thymeleaf

Versão do documento: 20230730 - 30 de julho de 2023

Versão do projeto: 3.1.2.RELEASE

Site do projeto: <https://www.thymeleaf.org>

1 Apresentando Thymeleaf

1.1 O que é Thymeleaf?

Thymeleaf é um moderno mecanismo de modelo Java do lado do servidor para ambientes web e autônomos, capaz de processar HTML, XML, JavaScript, CSS e até mesmo texto simples.

O principal objetivo do Thymeleaf é fornecer uma maneira elegante e de alta manutenção de criação de modelos. Para conseguir isso, ele se baseia no conceito de *Modelos Naturais* para injetar sua lógica em arquivos de modelo de uma forma que não afete o uso do modelo como protótipo de design. Isso melhora a comunicação do design e preenche a lacuna entre as equipes de design e desenvolvimento.

Thymeleaf também foi projetado desde o início com os padrões da Web em mente – especialmente **HTML5** – permitindo que você crie modelos de validação completa se for necessário.

1.2 Que tipo de modelos o Thymeleaf pode processar?

Pronto para uso, o Thymeleaf permite processar seis tipos de modelos, cada um dos quais é chamado de **Modo de modelo**:

- HTML
- XML
- TEXTO
- JAVASCRIPT
- CSS
- CRU

Existem dois modos de modelo *de marcação* HTML (e XML), três modos de modelo *textual* TEXT (, JAVASCRIPT e CSS) e um modo de modelo *autônomo* RAW ().

O HTMLmodo de modelo permitirá qualquer tipo de entrada HTML, incluindo HTML5, HTML 4 e XHTML. Nenhuma validação ou verificação de boa formação será realizada e o código/estrutura do modelo será respeitado ao máximo possível na saída.

O XMLmodo de modelo permitirá a entrada XML. Nesse caso, espera-se que o código esteja bem formado – sem tags não fechadas, sem atributos sem aspas, etc. – e o analisador lançará exceções se forem encontradas violações de boa formação. Observe que nenhuma *validação* (em relação a um DTD ou esquema XML) será executada.

O TEXTmodo de modelo permitirá o uso de uma sintaxe especial para modelos sem marcação. Exemplos de tais modelos podem ser e-mails de texto ou modelos de documentação. Observe que os modelos HTML ou XML também podem ser processados como TEXT, caso em que não serão analisados como marcação e cada tag, DOCTYPE, comentário, etc., será tratado como mero texto.

O JAVASCRIPTmodo modelo permitirá o processamento de arquivos JavaScript em um aplicativo Thymeleaf. Isso significa ser capaz de usar dados de modelo dentro de arquivos JavaScript da mesma forma que pode ser feito em arquivos HTML, mas com integrações específicas de JavaScript, como escape especializado ou *script natural*. O JAVASCRIPT modo modelo é considerado um modo *textual* e, portanto, usa a mesma sintaxe especial do TEXT modo modelo.

O CSSmodo template permitirá o processamento de arquivos CSS envolvidos em uma aplicação Thymeleaf. Semelhante ao JAVASCRIPT modo, o CSS modo modelo também é um modo *textual* e usa a sintaxe de processamento especial do TEXT modo modelo.

O RAWmodo de modelo simplesmente não processará nenhum modelo. Ele deve ser usado para inserir recursos intocados (arquivos, respostas de URL, etc.) nos modelos que estão sendo processados. Por exemplo, recursos externos e não controlados em formato HTML poderiam ser incluídos em modelos de aplicativos, sabendo com segurança que qualquer código Thymeleaf que esses recursos possam incluir não será executado.

1.3 Dialetos: O Dialeto Padrão

Thymeleaf é um mecanismo de modelo extremamente extensível (na verdade, poderia ser chamado de *estrutura de mecanismo de modelo*) que permite definir e personalizar a maneira como seus modelos serão processados com um nível de detalhe preciso.

Um objeto que aplica alguma lógica a um artefato de marcação (uma tag, algum texto, um comentário ou um mero espaço reservado se os modelos não forem marcação) é chamado de processador, e um conjunto desses processadores – além talvez de alguns artefatos extras – é o

que um **dialeto** normalmente é composto de. Pronto para uso, a biblioteca principal do Thymeleaf fornece um dialeto chamado **Standard Dialect**, que deve ser suficiente para a maioria dos usuários.

Observe que os dialetos podem, na verdade, não ter processadores e ser inteiramente compostos por outros tipos de artefatos, mas os processadores são definitivamente o caso de uso mais comum.

Este tutorial cobre o dialeto padrão . Cada recurso de atributo e sintaxe que você aprenderá nas páginas a seguir é definido por esse dialeto, mesmo que isso não seja explicitamente mencionado.

Claro, os usuários podem criar seus próprios dialetos (até mesmo estendendo o Padrão) se quiserem definir sua própria lógica de processamento enquanto aproveitam os recursos avançados da biblioteca. O Thymeleaf também pode ser configurado para usar vários dialetos ao mesmo tempo.

Os pacotes oficiais de integração thymeleaf-spring3 e thymeleaf-spring4 definem um dialeto chamado "SpringStandard Dialect", que é basicamente o mesmo que o Dialeto Padrão, mas com pequenas adaptações para fazer melhor uso de alguns recursos do Spring Framework (por exemplo , usando Spring Expression Language ou SpringEL em vez de OGNL). Portanto, se você é um usuário Spring MVC, não está perdendo tempo, pois quase tudo que você aprender aqui será útil em suas aplicações Spring.

A maioria dos processadores do Dialeto Padrão são *processadores de atributos* . Isso permite que os navegadores exibam corretamente os arquivos de modelo HTML, mesmo antes de serem processados, porque eles simplesmente ignorarão os atributos adicionais. Por exemplo, embora um JSP usando bibliotecas de tags possa incluir um fragmento de código que não pode ser exibido diretamente por um navegador, como:

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

...o Dialeto Padrão Thymeleaf nos permitiria alcançar a mesma funcionalidade com:

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

Isso não apenas será exibido corretamente pelos navegadores, mas também nos permite (opcionalmente) especificar um atributo de valor nele ("James Carrot", neste caso) que será exibido quando o protótipo for aberto estaticamente em um navegador, e que será substituído pelo valor resultante da avaliação \${user.name} durante o processamento do template.

Isso ajuda seu designer e desenvolvedor a trabalhar no mesmo arquivo de modelo e reduz o esforço necessário para transformar um protótipo estático em um arquivo de modelo funcional. A capacidade de fazer isso é um recurso chamado *Natural Templating* .

2 Mercearia Virtual The Good Thymes

O código-fonte dos exemplos mostrados neste e nos capítulos futuros deste guia pode ser encontrado no aplicativo de exemplo *Good Thymes Virtual Grocery (GTVG)*, que possui duas versões (equivalentes):

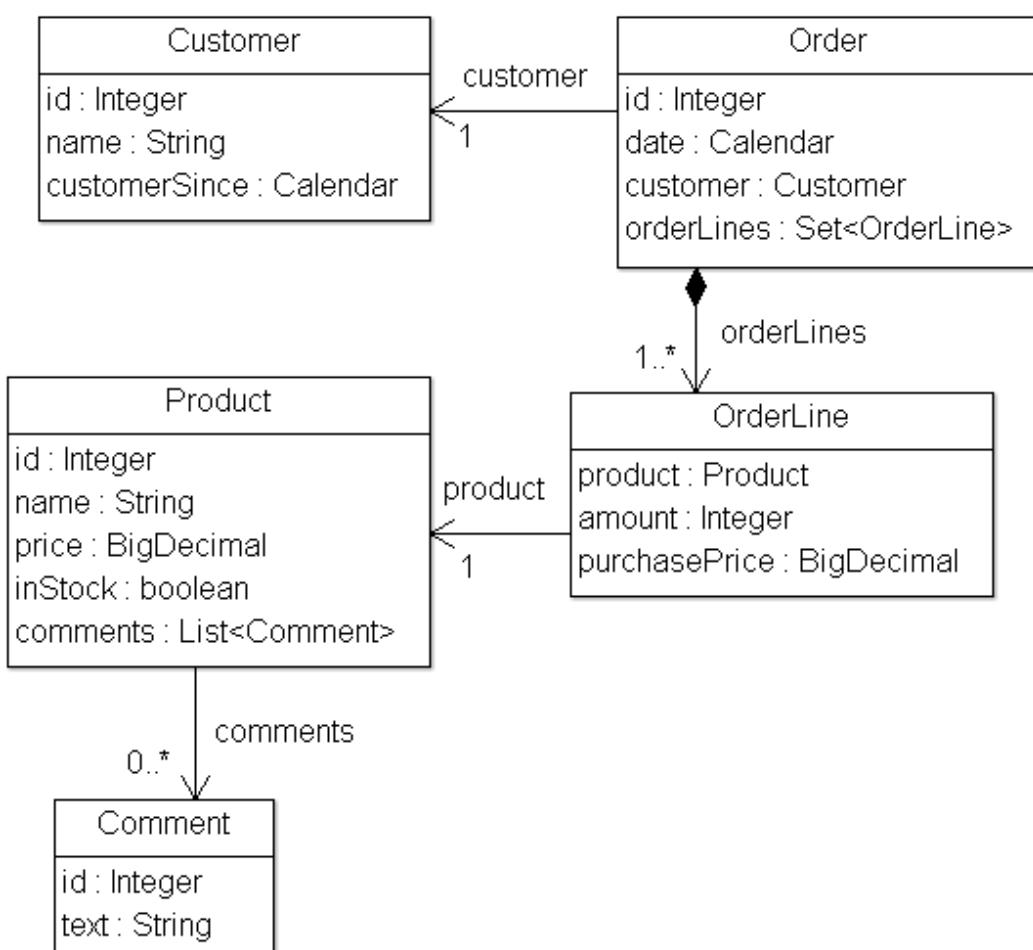
- javax.* baseado em: [gtvg-javax](#) .
- jakarta.* baseado em: [gtvg-jakarta](#) .

2.1 Um site para uma mercearia

Para explicar melhor os conceitos envolvidos no processamento de templates com Thymeleaf, este tutorial utilizará um aplicativo de demonstração que você pode baixar no site do projeto.

Este aplicativo é o site de uma mercearia virtual imaginária e nos fornecerá vários cenários para mostrar os diversos recursos do Thymeleaf.

Para começar, precisamos de um conjunto simples de entidades modelo para nossa aplicação: **Products** que são vendidas **Customers** criando arquivos **Orders**. Também estaremos gerenciando **Comments** sobre aqueles **Products**:



Modelo de aplicativo de exemplo

Nossa aplicação também terá uma camada de serviço bem simples, composta por **Service** objetos contendo métodos como:

```
public class ProductService {

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

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

}

Na camada web nossa aplicação terá um filtro que delegará a execução aos comandos habilitados para Thymeleaf dependendo da URL da solicitação:

```
/*
 * The application object needs to be declared first (implements IWebApplication)
 * In this case, the Jakarta-based version will be used.
 */
public void init(final FilterConfig filterConfig) throws ServletException {
    this.application =
        JakartaServletWebApplication.buildApplication(
            filterConfig.getServletContext());
    // We will see later how the TemplateEngine object is built and configured
    this.templateEngine = buildTemplateEngine(this.application);
}

/*
 * Each request will be processed by creating an exchange object (modeling
 * the request, its response and all the data needed for this process) and
 * then calling the corresponding controller.
 */
private boolean process(HttpServletRequest request, HttpServletResponse response)
    throws ServletException {

    try {

        final IWebExchange webExchange =
            this.application.buildExchange(request, response);
        final IWebRequest webRequest = webExchange.getRequest();

        // This prevents triggering engine executions for resource URLs
        if (request.getRequestURI().startsWith("/css") ||
            request.getRequestURI().startsWith("/images") ||
            request.getRequestURI().startsWith("/favicon")) {
            return false;
        }

        /*
         * 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.
         */
        final IGVController controller =
            ControllerMappings.resolveControllerForRequest(webRequest);
        if (controller == null) {
            return false;
        }

        /*
         * 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);

        /*
         * Obtain the response writer
         */
        final Writer writer = response.getWriter();

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

        return true;
    } catch (Exception e) {
        try {
            response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        }
    }
}
```

```

        } catch (final IOException ignored) {
            // Just ignore this
        }
        throw new ServletException(e);
    }
}

```

Esta é a nossa `IGTVGController` interface:

```

public interface IGTVGController {

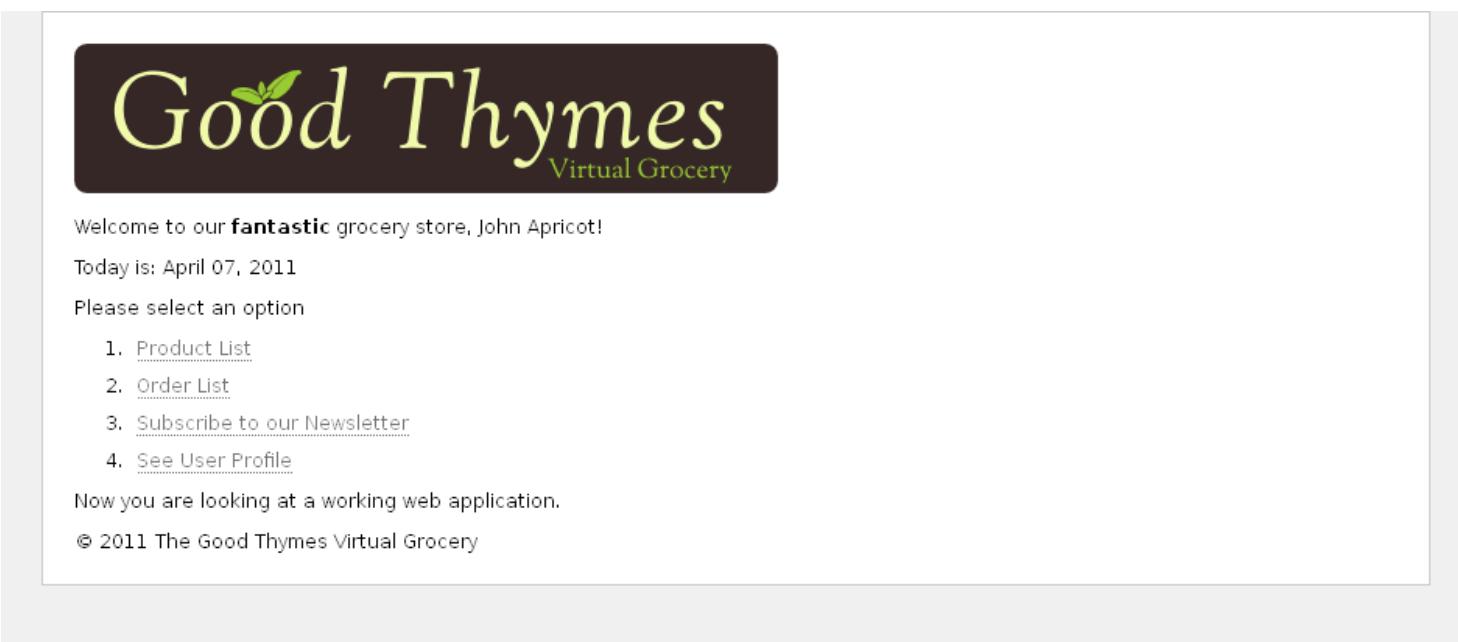
    public void process(
        final IWebExchange webExchange,
        final ITemplateEngine templateEngine,
        final Writer writer)
        throws Exception;

}

```

Tudo o que precisamos fazer agora é criar implementações da `IGTVGController` interface, recuperando dados dos serviços e processando templates utilizando o `ITemplateEngine` objeto.

No final, ficará assim:



Exemplo de página inicial do aplicativo

Mas primeiro vamos ver como esse mecanismo de modelo é inicializado.

2.2 Criando e configurando o Template Engine

O método `init(...)` em nosso filtro continha esta linha:

```
this.templateEngine = buildTemplateEngine(this.application);
```

Vamos ver agora como nosso `org.thymeleaf.TemplateEngine` objeto é inicializado:

```

private static ITemplateEngine buildTemplateEngine(final IWebApplication application) {

    // Templates will be resolved as application (ServletContext) resources
    final WebApplicationTemplateResolver templateResolver =
        new WebApplicationTemplateResolver(application);

    // HTML is the default mode, but we will set it anyway for better understanding of code
    templateResolver.setTemplateMode(TemplateMode.HTML);
    // This will convert "home" to "/WEB-INF/templates/home.html"
    templateResolver.setPrefix("/WEB-INF/templates/");
}

```

```

templateResolver.setSuffix(".html");
// Set template cache TTL to 1 hour. If not set, entries would live in cache until expelled by LRU
templateResolver.setCacheTTLMs(Long.valueOf(3600000L));

// Cache is set to true by default. Set to false if you want templates to
// be automatically updated when modified.
templateResolver.setCacheable(true);

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

return templateEngine;
}

```

Existem muitas maneiras de configurar um `TemplateEngine` objeto, mas por enquanto essas poucas linhas de código nos ensinarão o suficiente sobre as etapas necessárias.

O resolvedor de modelos

Vamos começar com o Resolvedor de Modelos:

```

final WebApplicationTemplateResolver templateResolver =
    new WebApplicationTemplateResolver(application);

```

Template Resolvers são objetos que implementam uma interface da API Thymeleaf chamada `org.thymeleaf.templateresolver.ITemplateResolver`:

```

public interface ITemplateResolver {

    ...

    /*
     * Templates are resolved by their name (or content) and also (optionally) their
     * owner template in case we are trying to resolve a fragment for another template.
     * Will return null if template cannot be handled by this template resolver.
     */
    public TemplateResolution resolveTemplate(
        final IEngineConfiguration configuration,
        final String ownerTemplate, final String template,
        final Map<String, Object> templateResolutionAttributes);
}

```

Esses objetos são responsáveis por determinar como nossos templates serão acessados e, nesta aplicação GTVG, a utilização de meios `org.thymeleaf.templateresolver.WebApplicationTemplateResolver` para recuperar nossos arquivos de templates como recursos do objeto `IWebApplication`: uma abstração Thymeleaf que, em aplicações baseadas em Servlet, basicamente envolve `[javax|jakarta].servlet.ServletContext` o objeto da API do Servlet e resolve recursos da raiz do aplicativo da web.

Mas isso não é tudo que podemos dizer sobre o resovedor de modelos, porque podemos definir alguns parâmetros de configuração nele. Primeiro, o modo de modelo:

```

templateResolver.setTemplateMode(TemplateMode.HTML);

```

HTML é o modo de modelo padrão para `WebApplicationTemplateResolver`, mas é uma boa prática estabelecê-lo de qualquer maneira, para que nosso código documente claramente o que está acontecendo.

```

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

```

O *prefixo* e o *sufixo* modificam os nomes dos templates que passaremos ao mecanismo para obter os nomes reais dos recursos a serem utilizados.

Usando esta configuração, o nome do modelo “*produto/lista*” corresponderia a:

```

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

```

Opcionalmente, o período de tempo que um modelo analisado pode permanecer no cache é configurado no Template Resolver por meio da propriedade `cacheTTLMs`:

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

Um modelo ainda pode ser expulso do cache antes que o TTL seja atingido se o tamanho máximo do cache for atingido e for a entrada mais antiga atualmente armazenada em cache.

O comportamento e os tamanhos do cache podem ser definidos pelo usuário implementando a `ICacheManager` interface ou modificando o `StandardCacheManager` objeto para gerenciar o cache padrão.

Há muito mais para aprender sobre resolvedores de modelos, mas por enquanto vamos dar uma olhada na criação de nosso objeto Template Engine.

O mecanismo de modelo

Os objetos do Template Engine são implementações da `org.thymeleaf.ITemplateEngine` interface. Uma dessas implementações é oferecida pelo núcleo do Thymeleaf: `org.thymeleaf.TemplateEngine` e criamos uma instância dela aqui:

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

Bastante simples, não é? Tudo o que precisamos é criar uma instância e definir o Template Resolver para ela.

Um resolvedor de modelo é o único parâmetro *necessário* `TemplateEngine`, embora existam muitos outros que serão abordados posteriormente (resolvedores de mensagens, tamanhos de cache, etc.). Por enquanto, isso é tudo que precisamos.

Nosso Template Engine agora está pronto e podemos começar a criar nossas páginas usando Thymeleaf.

3 Usando Textos

3.1 Boas-vindas multilíngues

Nossa primeira tarefa será criar uma página inicial para nosso site de mercearia.

A primeira versão desta página será extremamente simples: apenas um título e uma mensagem de boas-vindas. Este é o nosso `/WEB-INF/templates/home.html` arquivo:

```
<!DOCTYPE html>

<html 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>
```

A primeira coisa que você notará é que este arquivo é HTML5 e pode ser exibido corretamente por qualquer navegador porque não inclui nenhuma tag que não seja HTML (os navegadores ignoram todos os atributos que não entendem, como `th:text`).

Mas você também pode notar que este modelo não é realmente um documento HTML5 *válido*, porque esses atributos não padrão que usamos no `th:*` formulário não são permitidos pela especificação HTML5. Na verdade, estamos até adicionando um `xmlns:th` atributo à nossa `<html>` tag, algo absolutamente diferente do HTML5:

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

...que não tem nenhuma influência no processamento de templates, mas funciona como um *encantamento* que evita que nosso IDE reclame da falta de uma definição de namespace para todos esses `th:*` atributos.

E daí se quiséssemos tornar este modelo **válido para HTML5**? Fácil: mude para a sintaxe de atributos de dados do Thymeleaf, usando o `data-` prefixo para nomes de atributos e - separadores de hífen () em vez de ponto e vírgula (:):

```
<!DOCTYPE html>

<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" data-th-href="@{/css/gtvg.css}" />
    </head>

    <body>

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

    </body>

</html>
```

Atributos prefixados personalizados `data-` são permitidos pela especificação HTML5, portanto, com esse código acima, nosso modelo seria um *documento HTML5 válido*.

Ambas as notações são completamente equivalentes e intercambiáveis, mas por uma questão de simplicidade e compactação dos exemplos de código, este tutorial usará a *notação de namespace* (`th:*`). Além disso, a `th:*` notação é mais geral e permitida em todos os modos de modelo do Thymeleaf (XML , TEXT ...), enquanto a `data-` notação só é permitida no HTML modo.

Usando `th:text` e externalizando texto

Externalizar texto é extrair fragmentos de código de modelo de arquivos de modelo para que possam ser mantidos em arquivos separados (normalmente `.properties` arquivos) e possam ser facilmente substituídos por textos equivalentes escritos em outras línguas (um processo chamado internacionalização ou simplesmente *i18n*). Fragmentos externalizados de texto são geralmente chamados de “mensagens”.

As mensagens sempre possuem uma chave que as identifica, e o Thymeleaf permite especificar que um texto deve corresponder a uma mensagem específica com a `#{...}` sintaxe:

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

O que podemos ver aqui são, na verdade, duas características diferentes do dialeto padrão Thymeleaf:

- O `th:text` atributo, que avalia sua expressão de valor e define o resultado como o corpo da tag host, substituindo efetivamente “Bem-vindo ao nosso supermercado!” texto que vemos no código.
- A `#{home.welcome}` expressão, especificada na *Sintaxe de Expressão Padrão*, instruindo que o texto a ser usado pelo `th:text` atributo deve ser a mensagem com a `home.welcome` chave correspondente ao local com o qual estamos processando o modelo.

Agora, onde está esse texto externalizado?

A localização do texto externalizado no Thymeleaf é totalmente configurável e dependerá da `org.thymeleaf.messageresolver.IMessageResolver` implementação específica utilizada. Normalmente `.properties` será utilizada uma implementação baseada em arquivos, mas poderíamos criar nossas próprias implementações se quiséssemos, por exemplo, obter mensagens de um banco de dados.

No entanto, não especificamos um resolvedor de mensagens para nosso mecanismo de modelo durante a inicialização, e isso significa que nosso aplicativo está usando o *Standard Message Resolver*, implementado por `org.thymeleaf.messageresolver.StandardMessageResolver`.

O resolvedor de mensagens padrão espera encontrar mensagens `/WEB-INF/templates/home.html` em arquivos de propriedades na mesma pasta e com o mesmo nome do modelo, como:

- `/WEB-INF/templates/home_en.properties` para textos em inglês.
- `/WEB-INF/templates/home_es.properties` para textos em espanhol.
- `/WEB-INF/templates/home_pt_BR.properties` para textos em língua portuguesa (Brasil).
- `/WEB-INF/templates/home.properties` para textos padrão (se a localidade não corresponder).

Vamos dar uma olhada em nosso `home_es.properties` arquivo:

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

Isso é tudo que precisamos para fazer com que o Thymeleaf processe nosso modelo. Vamos criar nosso controlador Home então.

Contextos

Para processar nosso template, criaremos uma `HomeController` classe implementando a `IGTVGController` interface que vimos antes:

```
public class HomeController implements ITVGController {  
  
    public void process(  
        final IWebExchange webExchange,  
        final ITemplateEngine templateEngine,  
        final Writer writer)  
        throws Exception {  
  
        WebContext ctx = new WebContext(webExchange, webExchange.getLocale());  
  
        templateEngine.process("home", ctx, writer);  
    }  
}
```

```
}
```

A primeira coisa que vemos é a criação de um *contexto*. Um contexto Thymeleaf é um objeto que implementa a `org.thymeleaf.context.IContext` interface. Os contextos devem conter todos os dados necessários para a execução do mecanismo de template em um mapa de variáveis, e também referenciar o código do idioma que deve ser usado para mensagens externalizadas.

```
public interface IContext {  
  
    public Locale getLocale();  
    public boolean containsVariable(final String name);  
    public Set<String> getVariableNames();  
    public Object getVariable(final String name);  
  
}
```

Existe uma extensão especializada desta interface, `org.thymeleaf.context.IWebContext` destinada a ser usada em aplicações web.

```
public interface IWebContext extends IContext {  
  
    public IWebExchange getExchange();  
  
}
```

A biblioteca principal do Thymeleaf oferece uma implementação de cada uma destas interfaces:

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

E como você pode ver no código do controlador, `WebContext` é o que usamos. Na verdade, temos que fazê-lo, porque o uso de a `WebApplicationTemplateResolver` requer que usemos uma implementação de contexto `IWebContext`.

```
WebContext ctx = new WebContext(webExchange, webExchange.getLocale());
```

O `WebContext` construtor requer informações contidas no `IWebExchange` objeto de abstração que foi criado no filtro que representa esse intercâmbio baseado na web (ou seja, solicitação + resposta). A localidade padrão do sistema será usada se nenhuma for especificada (embora você nunca deva deixar isso acontecer em aplicações reais).

Existem algumas expressões especializadas que poderemos usar para obter os parâmetros de solicitação e os atributos de solicitação, sessão e aplicação em `WebContext` nossos modelos. Por exemplo:

- `${x}` retornará uma variável `x` armazenada no contexto do Thymeleaf ou como um *atributo de troca* (um “*atributo de solicitação*” no jargão do Servlet).
- `${param.x}` retornará um *parâmetro de solicitação* chamado `x` (que pode ter vários valores).
- `${session.x}` retornará um *atributo de sessão* chamado `x`.
- `${application.x}` retornará um *atributo de aplicativo* chamado `x` (um “*atributo de contexto de servlet*” no jargão do Servlet).

Executando o mecanismo de modelo

Com nosso objeto de contexto pronto, agora podemos dizer ao mecanismo de modelo para processar o modelo (pelo seu nome) usando o contexto e passando-o para um gravador de resposta para que a resposta possa ser escrita nele:

```
templateEngine.process("home", ctx, writer);
```

Vamos ver os resultados disso usando a localidade espanhola:

```
<!DOCTYPE html>  
  
<html>  
    <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 Mais sobre textos e variáveis

Texto sem escape

A versão mais simples da nossa página inicial parece estar pronta agora, mas há algo em que não pensamos... e se tivéssemos uma mensagem como esta?

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

Se executarmos este modelo como antes, obteremos:

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

O que não é exatamente o que esperávamos, pois nossa **b** tag foi escapada e portanto será exibida no navegador.

Este é o comportamento padrão do **th:text** atributo. Se quisermos que o Thymeleaf respeite nossas tags HTML e não escape delas, teremos que usar um atributo diferente: **th:utext** (para “texto sem escape”):

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

Isso produzirá nossa mensagem exatamente como queríamos:

```
<p>Welcome to our <b>fantastic</b> grocery store!</p>
```

Usando e exibindo variáveis

Agora vamos adicionar mais conteúdo à nossa página inicial. Por exemplo, podemos querer exibir a data abaixo da nossa mensagem de boas-vindas, assim:

```
Welcome to our fantastic grocery store!
```

```
Today is: 12 july 2010
```

Primeiro de tudo, teremos que modificar nosso controlador para adicionarmos essa data como uma variável de contexto:

```
public void process(  
    final IWebExchange webExchange,  
    final ITemplateEngine templateEngine,  
    final Writer writer)  
throws Exception {  
  
    SimpleDateFormat dateFormat = new SimpleDateFormat("dd MMMM yyyy");  
    Calendar cal = Calendar.getInstance();  
  
    WebContext ctx = new WebContext(webExchange, webExchange.getLocale());  
    ctx.setVariable("today", dateFormat.format(cal.getTime()));  
  
    templateEngine.process("home", ctx, writer);  
}
```

Adicionamos uma `String` variável chamada `today` ao nosso contexto e agora podemos exibi-la em nosso modelo:

```
<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>
```

Como você pode ver, ainda estamos usando o `th:text` atributo para o trabalho (e isso está correto, porque queremos substituir o corpo da tag), mas a sintaxe é um pouco diferente desta vez e em vez de um `# {...}` valor de expressão, estamos usando um `${...}` um. Esta é uma **expressão variável** e contém uma expressão em uma linguagem chamada *OGNL (Object-Graph Navigation Language)* que será executada no mapa de variáveis de contexto de que falamos antes.

A `${today}` expressão significa simplesmente “obter a variável chamada hoje”, mas essas expressões podem ser mais complexas (como `${user.name}` “obter a variável chamada usuário e chamar seu `getName()` método”).

Existem muitas possibilidades em valores de atributos: mensagens, expressões de variáveis... e muito mais. O próximo capítulo nos mostrará quais são todas essas possibilidades.

4 Sintaxe de Expressão Padrão

Faremos uma pequena pausa no desenvolvimento de nossa loja virtual de mercearia para aprender sobre uma das partes mais importantes do Dialetto Padrão Thymeleaf: a sintaxe da Expressão Padrão Thymeleaf.

Já vimos dois tipos de valores de atributos válidos expressos nesta sintaxe: mensagens e expressões variáveis:

```
<p th:utext="#{home.welcome}">Welcome to our grocery store!</p>  
<p>Today is: <span th:text="${today}">13 february 2011</span></p>
```

Mas existem mais tipos de expressões e mais detalhes interessantes para aprender sobre as que já conhecemos. Primeiro, vamos ver um rápido resumo dos recursos da Expressão Padrão:

- Expressões simples:
 - Expressões variáveis: \${...}
 - Expressões de variáveis de seleção: *{...}
 - Expressões de mensagem: #{...}
 - Expressões de URL do link: @{...}
 - Expressões de fragmento: ~{...}
- Literais
 - Literais de texto: 'one text' , 'Another one!' ,...
 - Literais de número: 0 , 34 , 3.0 , 12.3 ,...
 - Literais booleanos: true , false
 - Literal nulo: null
 - Tokens literais: one , sometext , main ,...
- Operações de texto:
 - Concatenação de strings: +
 - Substituições literais: |The name is \${name}|
- Operações aritméticas:
 - Operadores binários: + , - , * , / , %
 - Sinal de menos (operador unário): -
- Operações booleanas:
 - Operadores binários: and , or
 - Negação booleana (operador unário): ! , not
- Comparações e igualdade:
 - Comparadores: > , < , >= , <= (gt , lt , ge , le)
 - Operadores de igualdade: == , != (eq , ne)
- Operadores condicionais:
 - Se então: (if) ? (then)
 - Se-então-senão: (if) ? (then) : (else)
 - Padrão: (value) ?: (defaultvalue)
- Tokens especiais:
 - Nenhuma operação: _

Todos esses recursos podem ser combinados e aninhados:

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

4.1 Mensagens

Como já sabemos, `#{...}` as expressões de mensagem nos permitem vincular isto:

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

...para isso:

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

Mas há um aspecto que ainda não pensamos: o que acontece se o texto da mensagem não for completamente estático? E se, por exemplo, a nossa aplicação soubesse quem é o usuário que visita o site a qualquer momento e quiséssemos cumprimentá-lo pelo nome?

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

Isso significa que precisaríamos adicionar um parâmetro à nossa mensagem. Bem assim:

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

Os parâmetros são especificados de acordo com a `java.text.MessageFormat`sintaxe padrão, o que significa que você pode formatar números e datas conforme especificado na documentação da API para classes no `java.text.*` pacote.

Para especificar um valor para nosso parâmetro, e dado um atributo de sessão HTTP chamado `user`, poderíamos ter:

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

Observe que o uso `th:utext` aqui significa que a mensagem formatada não terá escape. Este exemplo pressupõe que `user.name` já tenha escapado.

Vários parâmetros podem ser especificados, separados por vírgulas.

A própria chave da mensagem pode vir de uma variável:

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

4.2 Variáveis

Já mencionamos que `${...}` as expressões são na verdade expressões OGNL (Object-Graph Navigation Language) executadas no mapa de variáveis contidas no contexto.

Para obter informações detalhadas sobre a sintaxe e os recursos do OGNL, você deve ler o [Guia de linguagem OGNL](#).

Em aplicativos habilitados para Spring MVC, o OGNL será substituído por `SpringEL`, mas sua sintaxe é muito semelhante à do OGNL (na verdade, exatamente a mesma para os casos mais comuns).

Pela sintaxe do OGNL, sabemos que a expressão em:

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

...é de fato equivalente a isto:

```
ctx.getVariable("today");
```

Mas OGNL nos permite criar expressões bem mais poderosas, e é assim:

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

...obtém o nome do usuário executando:

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

Mas a navegação pelo método getter é apenas um dos recursos do OGNL. Vejamos mais alguns:

```
/*  
 * Access to properties using the point (.). Equivalent to calling property getters.  
 */  
${person.father.name}  
  
/*  
 * Access to properties can also be made by using brackets ([] ) and writing  
 * the name of the property as a variable or between single quotes.  
 */  
${person['father']['name']}  
  
/*  
 * If the object is a map, both dot and bracket syntax will be equivalent to  
 * executing a call on its get(...) method.  
 */  
${countriesByCode.ES}  
 ${personsByName['Stephen Zucchini'].age}  
  
/*  
 * Indexed access to arrays or collections is also performed with brackets,  
 * writing the index without quotes.  
 */  
 ${personsArray[0].name}  
  
/*  
 * Methods can be called, even with arguments.  
 */  
 ${person.createCompleteName()}  
 ${person.createCompleteNameWithSeparator('-')}
```

Expressão Objetos Básicos

Ao avaliar expressões OGNL nas variáveis de contexto, alguns objetos são disponibilizados para expressões para maior flexibilidade. Esses objetos serão referenciados (de acordo com o padrão OGNL) começando com o # símbolo:

- `#ctx`: o objeto de contexto.
- `#vars`: as variáveis de contexto.
- `#locale`: a localidade do contexto.

Então podemos fazer isso:

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

Você pode ler a referência completa desses objetos no Apêndice A.

Objetos utilitários de expressão

Além destes objetos básicos, Thymeleaf nos oferecerá um conjunto de objetos utilitários que nos ajudarão a realizar tarefas comuns em nossas expressões.

- `#execInfo`: informações sobre o modelo que está sendo processado.

- `#messages` : métodos para obter mensagens externalizadas dentro de expressões de variáveis, da mesma forma que seriam obtidas usando a sintaxe `#{...}`.
- `#uris` : métodos para escapar de partes de URLs/URIs
- `#conversions` : métodos para executar o *serviço de conversão* configurado (se houver).
- `#dates` : métodos para `java.util.Date` objetos: formatação, extração de componentes, etc.
- `#calendars` : análogo a `#dates`, mas para `java.util.Calendar` objetos.
- `#temporals` : para lidar com datas e horas usando a `java.time` API no JDK8+.
- `#numbers` : métodos para formatar objetos numéricos.
- `#strings` : métodos para `String` objetos: contém, inicia com, precedendo/anexando, etc.
- `#objects` : métodos para objetos em geral.
- `#bools` : métodos para avaliação booleana.
- `#arrays` : métodos para matrizes.
- `#lists` : métodos para listas.
- `#sets` : métodos para conjuntos.
- `#maps` : métodos para mapas.
- `#aggregates` : métodos para criar agregações em arrays ou coleções.
- `#ids` : métodos para lidar com atributos id que podem ser repetidos (por exemplo, como resultado de uma iteração).

Você pode verificar quais funções são oferecidas por cada um desses objetos utilitários no [Apêndice B](#).

Reformatando datas em nossa página inicial

Agora que conhecemos esses objetos utilitários, poderíamos usá-los para alterar a forma como mostramos a data em nossa página inicial. Em vez de fazer isso em nosso `HomeController`:

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

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

templateEngine.process("home", ctx, writer);
```

... podemos fazer exatamente isso:

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

templateEngine.process("home", ctx, writer);
```

...e então execute a formatação da data na própria camada de visualização:

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

4.3 Expressões em seleções (sintaxe de asterisco)

Não apenas expressões variáveis podem ser escritas como `${...}`, mas também como `*{...}`.

Porém, há uma diferença importante: a sintaxe do asterisco avalia expressões em *objetos selecionados*, e não em todo o contexto. Ou seja, desde que não haja nenhum objeto selecionado, as sintaxes do dólar e do asterisco fazem exatamente o mesmo.

E o que é um objeto selecionado? O resultado de uma expressão usando o `th:object` atributo. Vamos usar um em nossa `userprofile.html` página de perfil de usuário ():

```
<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>
```

O que é exatamente equivalente a:

```
<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>
```

É claro que a sintaxe do dólar e do asterisco pode ser misturada:

```
<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>
```

Quando uma seleção de objeto estiver em vigor, o objeto selecionado também estará disponível para expressões de cifrão como a `#object` variável de expressão:

```
<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>
```

Como dito, se nenhuma seleção de objeto tiver sido realizada, as sintaxes de dólar e asterisco serão equivalentes.

```
<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 URLs de links

Devido à sua importância, os URLs são cidadãos de primeira classe em modelos de aplicativos da web, e o *Dialeto Padrão Thymeleaf* possui uma sintaxe especial para eles, a `@` sintaxe: `@{...}`

Existem diferentes tipos de URLs:

- URLs absolutos: `http://www.thymeleaf.org`
- URLs relativos, que podem ser:
 - Relativo à página: `user/login.html`
 - Relativo ao contexto: `/itemdetails?id=3` (o nome do contexto no servidor será adicionado automaticamente)
 - Relativo ao servidor: `~/billing/processInvoice` (permite chamar URLs em outro contexto (=aplicação) no mesmo servidor).
 - URLs relativos ao protocolo: `//code.jquery.com/jquery-2.0.3.min.js`

O real processamento dessas expressões e sua conversão nas URLs que serão geradas é feito por implementações da `org.thymeleaf.linkbuilder.ILinkBuilder` interface que são cadastradas no `ITemplateEngine` objeto utilizado.

Por padrão, é registrada uma única implementação desta interface da classe `org.thymeleaf.linkbuilder.StandardLinkBuilder`, o que é suficiente tanto para cenários offline (não web) quanto também web baseados na API do Servlet. Outros cenários (como integração com estruturas web não ServletAPI) podem precisar de implementações específicas da interface do construtor de links.

Vamos usar esta nova sintaxe. Conheça o `th:href` atributo:

```
<!-- 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>
```

Algumas coisas a serem observadas aqui:

- `th:href` é um atributo modificador: uma vez processado, ele calculará a URL do link a ser usada e definirá esse valor como o `href` atributo da `<a>` tag.
- Temos permissão para usar expressões para parâmetros de URL (como você pode ver em `orderId=${o.id}`). As operações necessárias de codificação de parâmetros de URL também serão executadas automaticamente.
- Se forem necessários vários parâmetros, estes serão separados por vírgulas: `@{/order/process(execId=${execId},execType='FAST')}`
- Modelos variáveis também são permitidos em caminhos de URL: `@{/order/{orderId}/details(orderId=${orderId})}`
- URLs relativos começando com `/` (por exemplo: `/order/details`) serão automaticamente prefixados pelo nome do contexto do aplicativo.
- Se os cookies não estiverem habilitados ou isso ainda não for conhecido, um `";jsessionid=..."` sufixo poderá ser adicionado às URLs relativas para que a sessão seja preservada. Isso é chamado de *reescrita de URL* e o Thymeleaf permite que você conecte seus próprios filtros de reescrita usando o `response.encodeURL(...)` mecanismo da API do Servlet para cada URL.
- O `th:href` atributo nos permite (opcionalmente) ter um `href` atributo estático funcional em nosso modelo, de modo que nossos links de modelo permaneçam navegáveis por um navegador quando abertos diretamente para fins de prototipagem.

Assim como aconteceu com a sintaxe da mensagem (`# {...}`), as bases de URL também podem ser o resultado da avaliação de outra expressão:

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

Um menu para nossa página inicial

Agora que sabemos como criar URLs de links, que tal adicionar um pequeno menu em nossa página inicial para algumas das outras páginas do site?

```
<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>
```

URLs relativos raiz do servidor

Uma sintaxe adicional pode ser usada para criar URLs relativos à raiz do servidor (em vez de relativos à raiz do contexto) para vincular a diferentes contextos no mesmo servidor. Esses URLs serão especificados como `@{~/path/to/something}`

4.5 Fragmentos

Expressões de fragmento são uma maneira fácil de representar fragmentos de marcação e movê-los pelos modelos. Isso nos permite replicá-los, passá-los para outros modelos como argumentos e assim por diante.

O uso mais comum é para inserção de fragmentos usando `th:insert` ou `th:replace` (mais sobre isso em uma seção posterior):

```
<div th:insert="~{commons :: main}">...</div>
```

Mas elas podem ser usadas em qualquer lugar, assim como qualquer outra variável:

```
<div th:with="frag=~{footer :: #main/text()}">
  <p th:insert="${frag}">
</div>
```

Posteriormente neste tutorial há uma seção inteira dedicada ao Layout do Modelo, incluindo explicações mais detalhadas sobre expressões de fragmentos.

Literais de texto

Literais de texto são apenas cadeias de caracteres especificadas entre aspas simples. Eles podem incluir qualquer caractere, mas você deve escapar de aspas simples dentro deles usando \'.

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

Literais numéricos

Literais numéricos são apenas isso: números.

```
<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>
```

Literais booleanos

Os literais booleanos são `true` e `false`. Por exemplo:

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

Neste exemplo, o `== false` está escrito fora dos colchetes e, portanto, é o Thymeleaf quem cuida disso. Se estivesse escrito entre colchetes, seria de responsabilidade dos motores OGNL/SpringEL:

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

O literal nulo

O `null` literal também pode ser usado:

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

Tokens literais

Literais numéricos, booleanos e nulos são na verdade um caso particular de *tokens literais*.

Esses tokens permitem um pouco de simplificação nas Expressões Padrão. Eles funcionam exatamente da mesma forma que literais de texto ('...'), mas só permitem letras (A-Z e a-z), números (0-9), colchetes ([e]), pontos (.), hífens (-) e sublinhados (_). Portanto, sem espaços em branco, sem vírgulas, etc.

A parte legal? Os tokens não precisam de aspas. Então podemos fazer isso:

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

em vez de:

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

4.7 Anexando textos

Os textos, não importa se são literais ou o resultado da avaliação de expressões de variáveis ou mensagens, podem ser facilmente anexados usando o + operador:

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

4.8 Substituições literais

As substituições literais permitem uma formatação fácil de strings contendo valores de variáveis sem a necessidade de acrescentar literais com '...' + '...'.

Essas substituições devem ser circundadas por barras verticais (|), como:

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

O que equivale a:

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

As substituições literais podem ser combinadas com outros tipos de expressões:

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

Somente expressões de variáveis/mensagens (\${...} , *{...} , # {...}) são permitidas dentro de | ... | substituições literais. Nenhum outro literal ('...'), tokens booleanos/numéricos, expressões condicionais etc.

4.9 Operações aritméticas

Algumas operações aritméticas também estão disponíveis: +, -, *, / %

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

Observe que esses operadores também podem ser aplicados dentro das próprias expressões de variáveis OGNL (e nesse caso serão executados pelo OGNL em vez do mecanismo Thymeleaf Standard Expression):

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

Observe que existem aliases textuais para alguns destes operadores: div (/), mod (%).

4.10 Comparadores e Igualdade

Os valores nas expressões podem ser comparados com os símbolos >, e , e os operadores e podem ser usados para verificar a < igualdade (ou a falta dela). Observe que o XML estabelece que os símbolos e não devem ser usados em valores de atributos e, portanto, devem ser substituídos por e .>= <= == != < > < >

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

Uma alternativa mais simples pode ser usar aliases textuais que existem para alguns destes operadores: gt (>), lt (<), ge (>=), le (<=), not (!). Também eq (==), neq / ne (!=).

4.11 Expressões condicionais

As expressões condicionais destinam-se a avaliar apenas uma de duas expressões, dependendo do resultado da avaliação de uma condição (que é em si outra expressão).

Vamos dar uma olhada em um exemplo de fragmento (apresentando outro *modificador de atributo*, `th:class`):

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

Todas as três partes de uma expressão condicional (`condition`, `then` e `else`) são elas próprias expressões, o que significa que podem ser variáveis (`${...}`, `*{...}`), mensagens (`# {...}`), URLs (`@ {...}`) ou literais (`'...'`).

Expressões condicionais também podem ser aninhadas usando parênteses:

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

As expressões `else` também podem ser omitidas; nesse caso, um valor nulo será retornado se a condição `for` falsa:

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

4.12 Expressões padrão (operador Elvis)

Uma *expressão padrão* é um tipo especial de valor condicional sem a parte `then`. É equivalente ao *operador Elvis* presente em algumas linguagens como Groovy, permite especificar duas expressões: a primeira é usada se não for avaliada como nula, mas se for, então a segunda é usada.

Vamos ver isso em ação em nossa página de perfil de usuário:

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

Como você pode ver, o operador é `? :`, e o usamos aqui para especificar um valor padrão para um nome (um valor literal, neste caso) somente se o resultado da avaliação `*{age}` for nulo. Isto é, portanto, equivalente a:

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

Tal como acontece com os valores condicionais, eles podem conter expressões aninhadas entre parênteses:

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

4.13 O token sem operação

O token No-Operation é representado por um símbolo de sublinhado (`_`).

A idéia por trás deste token é especificar que o resultado desejado para uma expressão é não *fazer nada*, ou seja, fazer exatamente como se o atributo processável (por exemplo `th:text`,) não estivesse lá.

Entre outras possibilidades, isso permite que os desenvolvedores usem texto de prototipagem como valores padrão. Por exemplo, em vez de:

```
<span th:text="${user.name} ?: 'no user authenticated'">...</span>
```

...podemos usar diretamente '*nenhum usuário autenticado*' como texto de prototipagem, o que resulta em um código mais conciso e versátil do ponto de vista do design:

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

4.14 Conversão/Formatação de Dados

Thymeleaf define uma sintaxe de *colchetes duplos* para expressões de variável (\${...}) e seleção (*{...}) que nos permite aplicar a conversão de dados por meio de um *serviço de conversão* configurado .

Basicamente é assim:

```
<td th:text="${{user.lastAccessDate}}">...</td>
```

Notou a chave dupla aí?: \${{{...}}} . Isso instrui o Thymeleaf a passar o resultado da `user.lastAccessDate` expressão para o *serviço de conversão* e solicita que ele execute uma **operação de formatação** (uma conversão para `String`) antes de gravar o resultado.

Supondo que `user.lastAccessDate` seja do tipo `java.util.Calendar`, se um *serviço de conversão* (implementação de `IStandardConversionService`) tiver sido registrado e contiver uma conversão válida para `Calendar -> String`, ele será aplicado.

A implementação padrão `IStandardConversionService` (da `StandardConversionService` classe) simplesmente é executada `.toString()` em qualquer objeto convertido para `String`. Para obter mais informações sobre como registrar uma implementação de *serviço de conversão* customizada, dê uma olhada na seção [Mais sobre Configuração..](#)

Os pacotes de integração oficiais `thymeleaf-spring3` e `thymeleaf-spring4` integram de forma transparente o mecanismo de serviço de conversão do Thymeleaf com a própria infraestrutura de *serviço de conversão* do Spring, para que os serviços de conversão e formatadores declarados na configuração do Spring sejam disponibilizados automaticamente para expressões \${{{...}}} e * {{...}} expressões.

4.15 Pré-processamento

Além de todos esses recursos para processamento de expressões, o Thymeleaf possui o recurso de *pré-processamento* de expressões.

O pré-processamento é uma execução das expressões feitas antes da normal que permite a modificação da expressão que eventualmente será executada.

As expressões pré-processadas são exatamente como as normais, mas aparecem rodeadas por um símbolo de sublinhado duplo (como `__${expression}__`).

Vamos imaginar que temos uma `Messages_fr.properties` entrada i18n contendo uma expressão OGNL chamando um método estático específico da linguagem, como:

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

...e um `Messages_es.properties` equivalente:

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

Podemos criar um fragmento de marcação que avalia uma expressão ou outra dependendo da localidade. Para isso, primeiro selecionaremos a expressão (por pré-processamento) e depois deixaremos o Thymeleaf executá-la:

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

Observe que a etapa de pré-processamento para uma localidade francesa criará o seguinte equivalente:

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

A String de pré-processamento __ pode ser escapada em atributos usando _.

5 Configurando Valores de Atributos

Este capítulo explicará como podemos definir (ou modificar) valores de atributos em nossa marcação.

5.1 Definindo o valor de qualquer atributo

Digamos que nosso site publica um boletim informativo e queremos que nossos usuários possam assiná-lo, então criamos um `/WEB-INF/templates/subscribe.html` modelo com um formulário:

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

Tal como acontece com o Thymeleaf, este modelo começa mais como um protótipo estático do que como um modelo para um aplicativo da web. Primeiro, o `action` atributo em nosso formulário é vinculado estaticamente ao próprio arquivo de modelo, de modo que não há lugar para reescrita útil de URL. Segundo, o `value` atributo no botão enviar faz com que ele exiba um texto em inglês, mas gostaríamos que fosse internacionalizado.

Insira então o `th:attr` atributo e sua capacidade de alterar o valor dos atributos das tags em que está definido:

```
<form action="subscribe.html" th:attr="action=@{/subscribe}">
  <fieldset>
    <input type="text" name="email" />
    <input type="submit" value="Subscribe!" th:attr="value=#[subscribe.submit]"/>
  </fieldset>
</form>
```

O conceito é bastante simples: `th:attr` simplesmente pega uma expressão que atribui um valor a um atributo. Tendo criado os arquivos de controlador e mensagens correspondentes, o resultado do processamento deste arquivo será:

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

Além dos novos valores de atributos, você também pode ver que o nome do contexto do aplicativo foi automaticamente prefixado à URL base em `/gtvg/subscribe`, conforme explicado no capítulo anterior.

Mas e se quiséssemos definir mais de um atributo por vez? As regras XML não permitem que você defina um atributo duas vezes em uma tag, portanto, `th:attr` será necessária uma lista de atribuições separadas por vírgula, como:

```

```

Dados os arquivos de mensagens necessários, isso resultará:

```

```

5.2 Definindo valor para atributos específicos

Até agora, você deve estar pensando que algo como:

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

...é uma marcação bastante feia. Especificar uma atribuição dentro do valor de um atributo pode ser muito prático, mas não é a maneira mais elegante de criar modelos se você tiver que fazer isso o tempo todo.

Thymeleaf concorda com você e é por isso que `th:attr` é pouco usado em templates. Normalmente, você usará outros `th:*` atributos cuja tarefa é definir atributos de tags específicos (e não qualquer atributo como `th:attr`).

Por exemplo, para definir o `value` atributo, use `th:value`:

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

Isso parece muito melhor! Vamos tentar fazer o mesmo com o `action` atributo da `form` tag:

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

E você lembra daqueles `th:href` que colocamos no nosso `home.html` antes? Eles são exatamente o mesmo tipo de atributos:

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

Existem muitos atributos como estes, cada um deles direcionado a um atributo HTML5 específico:

| | | |
|--------------------|---------------------|-------------------|
| 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:clsid |
| 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:fragment |
| th:frame | th:frameborder | th:headers |
| th:height | th:high | th:href |
| th:hreflang | th:hspace | th:http-equiv |
| th:icon | th:id | th:inline |
| 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:onabort | th:onafterprint |
| th:onbeforeprint | th:onbeforeunload | th:onblur |
| th:oncanplay | th:oncanplaythrough | th:onchange |
| th:onclick | th:oncontextmenu | th:ondblclick |
| th:ondrag | th:ondragend | th:ondragenter |
| th:ondragleave | th:ondragover | th:ondragstart |
| th:ondrop | th:ondurationchange | th:onemptied |

| | | |
|-----------------------|---------------------|-----------------|
| th:onended | th:onerror | th:onfocus |
| th:onformchange | th:onforminput | th:onhashchange |
| th:oninput | th:oninvalid | th:onkeydown |
| th:onkeypress | th:onkeyup | th:onload |
| th:onloadeddata | th:onloadedmetadata | th:onloadstart |
| th:onmessage | th:onmousedown | th:onmousemove |
| th:onmouseout | th:onmouseover | th:onmouseup |
| th:onmousewheel | th:onoffline | th:ononline |
| th:onpause | th:onplay | th:onplaying |
| th:onpopstate | th:onprogress | th:onratechange |
| th:onreadystatechange | th:onredo | th:onreset |
| th:onresize | th:onscroll | th:onseeked |
| th:onseeking | th:onselect | th:onshow |
| th:onstalled | th:onstorage | th:onsubmit |
| th:onsuspend | th:ontimeupdate | th:onundo |
| th:onunload | th:onvolumechange | th:onwaiting |
| 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 Definir mais de um valor por vez

Existem dois atributos bastante especiais chamados `th:alt-title` e `th:lang-xmllang` que podem ser usados para definir dois atributos com o mesmo valor ao mesmo tempo. Especificamente:

- `th:alt-title` irá definir `alt` e `title`.
- `th:lang-xmllang` irá definir `lang` e `xml:lang`.

Para nossa página inicial do GTVG, isso nos permitirá substituir isto:

```

```

...ou isto, que é equivalente:

```

```

...com isso:

```

```

5.4 Anexando e acrescentando

Thymeleaf também oferece os atributos `th:attrappend` e `th:attrprepend`, que acrescentam (sufixo) ou precedem (prefixo) o resultado de sua avaliação aos valores de atributos existentes.

Por exemplo, você pode querer armazenar o nome de uma classe CSS a ser adicionada (não definida, apenas adicionada) a um dos seus botões em uma variável de contexto, porque a classe CSS específica a ser usada dependeria de algo que o usuário fez antes:

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

Se você processar este modelo com a `cssStyle` variável definida como "warning", obterá:

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

Existem também dois *atributos de acréscimo* específicos no dialeto padrão: os atributos `th:classappend` e `th:styleappend`, que são usados para adicionar uma classe CSS ou um fragmento de *estilo* a um elemento sem substituir os existentes:

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

(Não se preocupe com esse `th:each` atributo. É um *atributo iterativo* e falaremos sobre isso mais tarde.)

5.5 Atributos booleanos de valor fixo

HTML possui o conceito de *atributos booleanos*, atributos que não possuem valor e a presença de um significa que o valor é "verdadeiro". Em XHTML, esses atributos assumem apenas 1 valor, que é ele mesmo.

Por exemplo `checked` ;

```
<input type="checkbox" name="option2" checked /> <!-- HTML -->
<input type="checkbox" name="option1" checked="checked" /> <!-- XHTML -->
```

O dialeto padrão inclui atributos que permitem definir esses atributos avaliando uma condição, de modo que, se avaliado como verdadeiro, o atributo será definido com seu valor fixo e, se avaliado como falso, o atributo não será definido:

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

Os seguintes atributos booleanos de valor fixo existem no dialeto padrão:

| | | |
|-------------------|--------------|-------------|
| 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:novalue |
| th:nowrap | th:open | th:pubdate |
| th:readonly | th:required | th:reversed |
| th:scoped | th:seamless | th:selected |

5.6 Configurando o valor de qualquer atributo (processador de atributos padrão)

Thymeleaf oferece um *processador de atributos padrão* que nos permite definir o valor de *qualquer* atributo, mesmo que nenhum `th:*` processador específico tenha sido definido para ele no Dialetos Padrão.

Então, algo como:

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

```
<span whatever="John Apricot">...</span>
```

5.7 Suporte para atributos e nomes de elementos compatíveis com HTML5

Também é possível usar uma sintaxe completamente diferente para aplicar processadores aos seus modelos de uma maneira mais compatível com HTML5.

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

A `data-{prefix}-{name}` sintaxe é a maneira padrão de escrever atributos personalizados em HTML5, sem exigir que os desenvolvedores usem nomes com namespaces como `th:*`. Thymeleaf disponibiliza esta sintaxe automaticamente para todos os seus dialetos (não apenas os padrão).

Há também uma sintaxe para especificar tags personalizadas: `{prefix}-{name}`, que segue a *especificação de elementos personalizados do W3C* (uma parte da *especificação maior de componentes da Web do W3C*). Isto pode ser usado, por exemplo, para o `th:block` elemento (ou também `th-block`), que será explicado em uma seção posterior.

Importante: esta sintaxe é um acréscimo à sintaxe com namespace `th:*`, não a substitui. Não há nenhuma intenção de descontinuar a sintaxe com namespace no futuro.

6 Iteração

Até agora criámos uma página inicial, uma página de perfil de utilizador e também uma página para permitir aos utilizadores subscrever a nossa newsletter... mas e os nossos produtos? Para isso, precisaremos de uma maneira de iterar os itens de uma coleção para construir nossa página de produto.

6.1 Noções básicas de iteração

Para exibir os produtos em nossa `/WEB-INF/templates/product/list.html` página utilizaremos uma tabela. Cada um dos nossos produtos será exibido em uma linha (um `<tr>` elemento), e assim para o nosso modelo precisaremos criar uma *linha de modelo* – uma que exemplifique como queremos que cada produto seja exibido – e então instruir o Thymeleaf a repeti-lo, uma vez para cada produto.

O Dialet Padrão nos oferece um atributo exatamente para isso: `th:each`.

Usando `th:each`

Para nossa página de lista de produtos, precisaremos de um método controlador que recupere a lista de produtos da camada de serviço e a adicione ao contexto do modelo:

```
public void process(
    final IWebExchange webExchange,
    final ITemplateEngine templateEngine,
    final Writer writer)
    throws Exception {

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

    final WebContext ctx = new WebContext(webExchange, webExchange.getLocale());
    ctx.setVariable("prods", allProducts);

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

E então usaremos `th:each` em nosso modelo para iterar a lista de produtos:

```
<!DOCTYPE html>

<html 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>
    </body>
</html>
```

```

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

</body>

</html>

```

Esse `prod : ${prods}` valor de atributo que você vê acima significa “para cada elemento no resultado da avaliação `${prods}` , repita este fragmento do template, usando o elemento atual em uma variável chamada `prod`”. Vamos dar um nome a cada uma das coisas que vemos:

- Chamaremos `${prods}` a *expressão iterada* ou *variável iterada*.
- Chamaremos `prod` a *variável de iteração* ou simplesmente *variável de iter*.

Observe que a `prod` variável iter tem como escopo o `<tr>` elemento, o que significa que está disponível para tags internas como `<td>`.

Valores iteráveis

A `java.util.List` classe não é o único valor que pode ser usado para iteração no Thymeleaf. Existe um conjunto bastante completo de objetos que são considerados *iteráveis* por um `th:each` atributo:

- Qualquer objeto implementando `java.util.Iterable`
- Qualquer objeto implementando `java.util Enumeration` .
- Qualquer objeto implementando `java.util.Iterator` , cujos valores serão usados à medida que forem retornados pelo iterador, sem a necessidade de armazenar todos os valores em cache na memória.
- Qualquer objeto implementando `java.util.Map` . Ao iterar mapas, as variáveis iter serão da classe `java.util.Map.Entry` .
- Qualquer objeto implementando `java.util.stream.Stream` .
- Qualquer matriz.
- Qualquer outro objeto será tratado como se fosse uma lista de valor único contendo o próprio objeto.

6.2 Mantendo o status da iteração

Ao usar `th:each` , o Thymeleaf oferece um mecanismo útil para acompanhar o status da sua iteração: a *variável status* .

As variáveis de status são definidas em um `th:each` atributo e contêm os seguintes dados:

- O *índice de iteração* atual , começando com 0. Esta é a `index` propriedade.
- O *índice de iteração* atual , começando com 1. Esta é a `count` propriedade.
- A quantidade total de elementos na variável iterada. Esta é a `size` propriedade.
- A *variável iter* para cada iteração. Esta é a `current` propriedade.
- Se a iteração atual é par ou ímpar. Estas são as `even/odd` propriedades booleanas.
- Se a iteração atual é a primeira. Esta é a `first` propriedade booleana.
- Se a iteração atual é a última. Esta é a `last` propriedade booleana.

Vamos ver como poderíamos usá-lo com o exemplo anterior:

```

<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>

```

A variável `status` (`iterStat` neste exemplo) é definida no `th:each` atributo escrevendo seu nome após a própria variável `iter`, separada por vírgula. Assim como a variável `iter`, a variável `status` também tem como escopo o fragmento de código definido pela tag que contém o `th:each` atributo.

Vamos dar uma olhada no resultado do processamento do nosso modelo:

```

<!DOCTYPE html>

<html>

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

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

</html>

```

Observe que nossa variável de status de iteração funcionou perfeitamente, estabelecendo a `odd` classe CSS apenas para linhas ímpares.

Se você não definir explicitamente uma variável de status, o Thymeleaf sempre criará uma para você adicionando um sufixo `Stat` ao nome da variável de iteração:

```

<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>

```

Às vezes, podemos querer otimizar a recuperação de coleções de dados (por exemplo, de um banco de dados) para que essas coleções só sejam recuperadas se realmente forem usadas.

Na verdade, isso é algo que pode ser aplicado a *qualquer* dado, mas dado o tamanho que as coleções na memória podem ter, a recuperação de coleções que devem ser iteradas é o caso mais comum neste cenário.

Para suportar isso, o Thymeleaf oferece um mecanismo para *carregar variáveis de contexto preguiçosamente*. Variáveis de contexto que implementam a `ILazyContextVariable` interface – provavelmente estendendo sua `LazyContextVariable` implementação padrão – serão resolvidas no momento de serem executadas. Por exemplo:

```
context.setVariable(  
    "users",  
    new LazyContextVariable<List<User>>() {  
        @Override  
        protected List<User> loadValue() {  
            return databaseRepository.findAllUsers();  
        }  
    });
```

Esta variável pode ser usada sem conhecimento de sua *preguiça*, em códigos como:

```
<ul>  
    <li th:each="u : ${users}" th:text="${u.name}">user name</li>  
</ul>
```

Mas, ao mesmo tempo, nunca será inicializado (seu `loadValue()` método nunca será chamado) se `condition` for avaliado `false` em código como:

```
<ul th:if="${condition}">  
    <li th:each="u : ${users}" th:text="${u.name}">user name</li>  
</ul>
```

7 Avaliação Condicional

7.1 Condicionais simples: "se" e "a menos que"

Às vezes você precisará que um fragmento do seu modelo apareça apenas no resultado se uma determinada condição for atendida.

Por exemplo, imagine que queremos mostrar em nossa tabela de produtos uma coluna com a quantidade de comentários que existem para cada produto e, se houver algum comentário, um link para a página de detalhes dos comentários desse produto.

Para fazer isso, usariamos o `th:if` atributo:

```
<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>
```

Muitas coisas para ver aqui, então vamos nos concentrar na linha importante:

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

Isso criará um link para a página de comentários (com URL `/product/comments`) com um `prodId` parâmetro definido para `id` o produto, mas somente se o produto tiver algum comentário.

Vamos dar uma olhada na marcação resultante:

```
<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>

```

Perfeito! Isso é exatamente o que queríamos.

Observe que o `th:if` atributo não avaliará apenas condições *booleanas*. Suas capacidades vão um pouco além disso e avaliará a expressão especificada `true` seguindo estas regras:

- Se o valor não for nulo:
 - Se value for booleano e for `true`.
 - Se o valor for um número e for diferente de zero
 - Se o valor for um caractere e for diferente de zero
 - Se o valor for uma String e não for "false", "off" ou "no"
 - Se o valor não for um booleano, um número, um caractere ou uma String.
- (Se o valor for nulo, `th:if` será avaliado como falso).

Além disso, `th:if` possui um atributo inverso, `th:unless` que poderíamos ter usado no exemplo anterior em vez de usar a `not` dentro da expressão OGNL:

```

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

```

7.2 Declarações de troca

Também existe uma maneira de exibir o conteúdo condicionalmente usando o equivalente a uma estrutura *de switch* em Java: o conjunto de atributos `th:switch` / `th:case`.

```

<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>

```

Observe que assim que um `th:case` atributo é avaliado como `true`, todos os outros `th:case` atributos no mesmo contexto de switch são avaliados como `false`.

A opção padrão é especificada como `th:case="*"`:

```

<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 Layout do modelo

8.1 Incluindo fragmentos de modelo

Definindo e referenciando fragmentos

Em nossos modelos, muitas vezes desejaremos incluir partes de outros modelos, partes como rodapés, cabeçalhos, menus...

Para fazer isso, o Thymeleaf precisa que definamos essas partes, “fragmentos”, para inclusão, o que pode ser feito usando o `th:fragment` atributo.

Digamos que queremos adicionar um rodapé de direitos autorais padrão a todas as nossas páginas de supermercado, então criamos um `/WEB-INF/templates/footer.html` arquivo contendo este código:

```
<!DOCTYPE html>

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

    <body>

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

    </body>

</html>
```

O código acima define um fragmento chamado `copy` que podemos incluir facilmente em nossa página inicial usando um dos atributos `th:insert` ou `:th:replace`

```
<body>

    ...

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

</body>
```

Observe que `th:insert` espera uma *expressão de fragmento* (`~{...}`), que é *uma expressão que resulta em um fragmento*.

Sintaxe de especificação de fragmento

A sintaxe das *expressões de fragmentos* é bastante direta. Existem três formatos diferentes:

- "`~{templatename::selector}`" Inclui o fragmento resultante da aplicação do Seletor de marcação especificado no modelo denominado `templatename`. Observe que `selector` pode ser um mero nome de fragmento, então você pode especificar algo tão simples como `~{templatename::fragmentname}` acima `~{footer :: copy}`.

A sintaxe do seletor de marcação é definida pela biblioteca de análise AttoParser subjacente e é semelhante às expressões XPath ou seletores CSS. Consulte [o Apêndice C](#) para obter mais informações.

- "`~{templatename}`" Inclui o modelo completo chamado `templatename`.

Observe que o nome do modelo que você usa nas tags `th:insert` / `th:replace` terá que ser resolvido pelo Template Resolver atualmente usado pelo Template Engine.

- `~{::selector}" ou "~{this::selector}"` Insere um fragmento do mesmo modelo, correspondente a `selector`. Se não for encontrado no modelo onde a expressão aparece, a pilha de chamadas de modelo (inserções) é percorrida em direção ao modelo originalmente processado (*raiz*), até `selector` corresponder em algum nível.

Ambos `templatename` e `selector` nos exemplos acima podem ser expressões completas (até mesmo condicionais!), Como:

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

Os fragmentos podem incluir quaisquer `th:*` atributos. Esses atributos serão avaliados assim que o fragmento for incluído no modelo de destino (aquele com o atributo `th:insert` / `th:replace`) e poderão fazer referência a quaisquer variáveis de contexto definidas neste modelo de destino.

Uma grande vantagem dessa abordagem para fragmentos é que você pode escrever seus fragmentos em páginas que podem ser perfeitamente exibidas por um navegador, com uma estrutura de marcação completa e até *válida*, enquanto ainda mantém a capacidade de fazer com que o Thymeleaf os inclua em outros modelos.

Referenciando fragmentos sem `th:fragment`

Graças ao poder dos seletores de marcação, podemos incluir fragmentos que não utilizam nenhum `th:fragment` atributo. Pode até ser um código de marcação vindo de um aplicativo diferente, sem nenhum conhecimento do Thymeleaf:

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

Podemos usar o fragmento acima simplesmente referenciando-o pelo seu `id` atributo, de forma semelhante a um seletor CSS:

```
<body>
    ...
    <div th:insert="~{#copy-section}"></div>
</body>
```

Diferença entre `th:insert` e `th:replace`

E qual é a diferença entre `th:insert` e `th:replace`?

- `th:insert` irá simplesmente inserir o fragmento especificado como o corpo de sua tag host.
- `th:replace` na verdade, *substitui* sua tag de host pelo fragmento especificado.

Portanto, um fragmento HTML como este:

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

...incluído duas vezes nas `<div>` tags host, assim:

```
<body>
    ...
    <div th:insert="~{#copy}"></div>
    <div th:replace="~{#copy}"></div>
```

```
</body>
```

...vai resultar em:

```
<body>
  ...
<div>
  <footer>
    © 2011 The Good Thymes Virtual Grocery
  </footer>
</div>

<footer>
  © 2011 The Good Thymes Virtual Grocery
</footer>

</body>
```

8.2 Assinaturas de fragmentos parametrizáveis

Para criar um mecanismo mais *funcional* para fragmentos de modelo, os fragmentos definidos com `th:fragment` podem especificar um conjunto de parâmetros:

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

Isso requer o uso de uma dessas duas sintaxes para chamar o fragmento from `th:insert` ou `th:replace`:

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

Observe que a ordem não é importante na última opção:

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

Fragmentar variáveis locais sem argumentos de fragmento

Mesmo que os fragmentos sejam definidos sem argumentos como este:

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

Poderíamos usar a segunda sintaxe especificada acima para chamá-los (e apenas a segunda):

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

Isso seria equivalente a uma combinação de `th:replace` e `th:with`:

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

Observe que esta especificação de variáveis locais para um fragmento – não importa se ele possui uma assinatura de argumento ou não – não faz com que o contexto seja esvaziado antes de sua execução. Os fragmentos ainda poderão acessar todas as variáveis de contexto usadas no modelo de chamada, como fazem atualmente.

th:assert para asserções no modelo

O `th:assert` atributo pode especificar uma lista de expressões separadas por vírgula que devem ser avaliadas e produzir verdadeiras para cada avaliação, gerando uma exceção caso contrário.

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

Isto é útil para validar parâmetros em uma assinatura de fragmento:

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

8.3 Layouts flexíveis: além da mera inserção de fragmentos

Graças às *expressões de fragmento*, podemos especificar parâmetros para fragmentos que não são textos, números, objetos bean... mas sim fragmentos de marcação.

Isso nos permite criar nossos fragmentos de forma que possam ser *enriquecidos* com marcação proveniente dos modelos de chamada, resultando em um **mecanismo de layout de modelo** muito flexível.

Observe o uso das variáveis `title` e `links` no fragmento abaixo:

```
<head th:fragment="common_header(title,links)">

    <title th:replace="${title}">The awesome application</title>

    <!-- Common styles and scripts -->
    <link rel="stylesheet" type="text/css" media="all" th:href="@{/css/awesomeapp.css}">
    <link rel="shortcut icon" th:href="@{/images/favicon.ico}">
    <script type="text/javascript" th:src="@{/sh/scripts/codebase.js}"></script>

    <!--/* Per-page placeholder for additional links */-->
    <th:block th:replace="${links}" />

</head>
```

Agora podemos chamar este fragmento como:

```
...
<head th:replace="~{ base :: common_header(~{::title},~{::link}) }">

    <title>Awesome - Main</title>

    <link rel="stylesheet" th:href="@{/css/bootstrap.min.css}">
    <link rel="stylesheet" th:href="@{/themes/smoothness/jquery-ui.css}">

</head>
...
```

...e o resultado usará as tags reais `<title>` e `<link>` do nosso modelo de chamada como os valores das variáveis `title` e `links`, resultando na personalização do nosso fragmento durante a inserção:

```
...
<head>

    <title>Awesome - Main</title>

    <!-- Common styles and scripts -->
    <link rel="stylesheet" type="text/css" media="all" href="/awe/css/awesomeapp.css">
    <link rel="shortcut icon" href="/awe/images/favicon.ico">
    <script type="text/javascript" src="/awe/sh/scripts/codebase.js"></script>

    <link rel="stylesheet" href="/awe/css/bootstrap.min.css">
    <link rel="stylesheet" href="/awe/themes/smoothness/jquery-ui.css">
```

```
</head>
```

```
...
```

Usando o fragmento vazio

Uma expressão de fragmento especial, o *fragmento vazio* (`~{}`), pode ser usada para especificar *nenhuma marcação*. Usando o exemplo anterior:

```
<head th:replace="~{ base :: common_header(~{::title},~{}) }">

<title>Awesome - Main</title>

</head>
...
```

Observe como o segundo parâmetro do fragmento (`links`) está definido para o *fragmento vazio* e, portanto, nada é escrito para o `<th:block th:replace="${links}" />` bloco:

```
...
<head>

<title>Awesome - Main</title>

<!-- Common styles and scripts --&gt;
&lt;link rel="stylesheet" type="text/css" media="all" href="/awe/css/awesomeapp.css"&gt;
&lt;link rel="shortcut icon" href="/awe/images/favicon.ico"&gt;
&lt;script type="text/javascript" src="/awe/sh/scripts/codebase.js"&gt;&lt;/script&gt;

&lt;/head&gt;
...</pre>
```

Usando o token de não operação

O no-op também pode ser usado como parâmetro para um fragmento se quisermos apenas deixar nosso fragmento usar sua marcação atual como valor padrão. Novamente, usando o `common_header` exemplo:

```
...
<head th:replace="~{base :: common_header(_,~{::link})}">

<title>Awesome - Main</title>

<link rel="stylesheet" th:href="@{/css/bootstrap.min.css}">
<link rel="stylesheet" th:href="@{/themes/smoothness/jquery-ui.css}">

</head>
...
```

Veja como o `title` argumento (primeiro argumento do `common_header` fragmento) é definido como *no-op* (`_`), o que faz com que esta parte do fragmento não seja executada (`title = no-operation`):

```
<title th:replace="${title}">The awesome application</title>
```

Então o resultado é:

```
...
<head>

<title>The awesome application</title>

<!-- Common styles and scripts --&gt;
&lt;link rel="stylesheet" type="text/css" media="all" href="/awe/css/awesomeapp.css"&gt;
&lt;link rel="shortcut icon" href="/awe/images/favicon.ico"&gt;
&lt;script type="text/javascript" src="/awe/sh/scripts/codebase.js"&gt;&lt;/script&gt;

&lt;link rel="stylesheet" href="/awe/css/bootstrap.min.css"&gt;</pre>
```

```
<link rel="stylesheet" href="/awe/themes/smoothness/jquery-ui.css">
```

```
</head>
```

```
...
```

Inserção condicional avançada de fragmentos

A disponibilidade do *fragmento vazio* e do *token sem operação* nos permite realizar a inserção condicional de fragmentos de uma forma muito fácil e elegante.

Por exemplo, poderíamos fazer isso para inserir nosso `common :: adminhead` fragmento *apenas* se o usuário for um administrador, e inserir nada (fragmento vazio) se não for:

```
...
<div th:insert="${user.isAdmin()} ? ~{common :: adminhead} : ~{}>...</div>
...
```

Além disso, podemos usar o *token de não operação* para inserir um fragmento somente se a condição especificada for atendida, mas deixar a marcação sem modificações se a condição não for atendida:

```
...
<div th:insert="${user.isAdmin()} ? ~{common :: adminhead} : _">
    Welcome [[${user.name}]], click <a th:href="@{/support}">here</a> for help-desk support.
</div>
...
```

Além disso, se tivermos configurado nossos resolvedores de modelo para *verificar a existência* dos recursos do modelo – por meio de seu `checkExistence` sinalizador – podemos usar a existência do próprio fragmento como condição em uma operação *padrão*:

```
...
<!-- The body of the <div> will be used if the "common :: salutation" fragment -->
<!-- does not exist (or is empty). -->
<div th:insert="~{common :: salutation} ?: _">
    Welcome [[${user.name}]], click <a th:href="@{/support}">here</a> for help-desk support.
</div>
...
```

8.4 Removendo fragmentos de modelo

Voltando ao aplicativo de exemplo, vamos revisitar a última versão do nosso modelo de lista de produtos:

```
<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>
```

Este código funciona perfeitamente como modelo, mas como página estática (quando aberta diretamente por um navegador sem que o Thymeleaf o processe) não seria um bom protótipo.

Por que? Porque, embora perfeitamente exibível pelos navegadores, essa tabela possui apenas uma linha, e essa linha possui dados simulados. Como protótipo, simplesmente não pareceria realista o suficiente... deveríamos ter mais de um produto, *precisamos de mais linhas*.

Então, vamos adicionar alguns:

```
<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>
```

Ok, agora temos três, definitivamente melhores para um protótipo. Mas... o que acontecerá quando processarmos com Thymeleaf?:

```
<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>
```

```

</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>

```

As duas últimas linhas são linhas simuladas! Bem, é claro que são: a iteração foi aplicada apenas à primeira linha, então não há razão para que o Thymeleaf tenha removido as outras duas.

Precisamos de uma maneira de remover essas duas linhas durante o processamento do modelo. Vamos usar o `th:remove` atributo na segunda e terceira `<tr>` tags:

```


| NAME          | PRICE | IN STOCK | COMMENTS                                                                                                                                                                                            |
|---------------|-------|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Onions        | 2.41  | yes      | <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 |
| Blue Lettuce  | 9.55  | no       | <span>0</span> comment/s                                                                                                                                                                            |
| Mild Cinnamon | 1.99  | yes      | <span>3</span> comment/s <a href="comments.html">view</a>                                                                                                                                           |


```

Depois de processado, tudo ficará como deveria:

```


| NAME               | PRICE | IN STOCK | COMMENTS                                                                    |
|--------------------|-------|----------|-----------------------------------------------------------------------------|
| Fresh Sweet Basil  | 4.99  | yes      | <span>0</span> comment/s <a href="/gtvg/product/comments?prodId=1">view</a> |
| Italian Tomato     | 1.25  | no       | <span>2</span> comment/s <a href="/gtvg/product/comments?prodId=2">view</a> |
| Yellow Bell Pepper | 2.50  | yes      | <span>0</span> comment/s                                                    |
| Old Cheddar        | 18.75 | yes      | <span>1</span> comment/s <a href="/gtvg/product/comments?prodId=4">view</a> |


```

E o que `all` significa esse valor no atributo? `th:remove` pode se comportar de cinco maneiras diferentes, dependendo do seu valor:

- `all`: Remova a tag que contém e todos os seus filhos.
- `body`: não remova a tag que contém, mas remova todos os seus filhos.
- `tag`: remova a tag que contém, mas não remova seus filhos.
- `all-but-first`: Remova todos os filhos da tag que contém, exceto o primeiro.
- `none`: Fazer nada. Este valor é útil para avaliação dinâmica.

Para que esse `all-but-first` valor pode ser útil? Isso nos permitirá economizar alguns `th:remove="all"` durante a prototipagem:

```


| NAME   | PRICE | IN STOCK | COMMENTS                                                                                                                                                                                        |
|--------|-------|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Onions | 2.41  | yes      | <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> |


```

```

</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>

```

O `th:remove` atributo pode assumir qualquer *Expressão Padrão Thymeleaf*, desde que retorne um dos valores String permitidos (`all` , `tag` , `body` , `all-but-first` ou `none`).

Isso significa que as remoções podem ser condicionais, como:

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

Observe também que `th:remove` considera `null` um sinônimo para `none` , portanto o seguinte funciona da mesma forma que o exemplo acima:

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

Neste caso, se `${condition}` for falso, `null` será retornado e, portanto, nenhuma remoção será realizada.

8.5 Herança de Layout

Para poder ter um único arquivo como layout, podem ser utilizados fragmentos. Um exemplo de layout simples com `title` e `content` usando `th:fragment` and `th:replace` :

```

<!DOCTYPE html>
<html th:fragment="layout (title, content)" xmlns:th="http://www.thymeleaf.org">
<head>
    <title th:replace="${title}">Layout Title</title>
</head>
<body>
    <h1>Layout H1</h1>
    <div th:replace="${content}">
        <p>Layout content</p>
    </div>
    <footer>
        Layout footer
    </footer>
</body>
</html>

```

Este exemplo declara um fragmento chamado **layout** tendo *título* e *conteúdo* como parâmetros. Ambos serão substituídos na página que a herda pelas expressões de fragmento fornecidas no exemplo abaixo.

```

<!DOCTYPE html>
<html th:replace="~{layoutFile :: layout(~{::title}, ~{::section})}">
<head>
    <title>Page Title</title>
</head>
<body>
<section>
    <p>Page content</p>
    <div>Included on page</div>

```

```
</section>
</body>
</html>
```

Neste arquivo, a `html` tag será substituída por *layout*, mas no layout `title` e `content` terá sido substituída pelos blocos `title` e `section` respectivamente.

Se desejar, o layout pode ser composto por vários fragmentos como *cabeçalho* e *rodapé*.

9 variáveis locais

Thymeleaf chama de *variáveis locais* as variáveis que são definidas para um fragmento específico de um modelo e estão disponíveis apenas para avaliação dentro desse fragmento.

Um exemplo que já vimos é a `prod` variável iter em nossa página de lista de produtos:

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

Essa `prod` variável estará disponível apenas dentro dos limites da `<tr>` tag. Especificamente:

- Ele estará disponível para quaisquer outros `th:*` atributos executados naquela tag com menor *precedência* (o `th:each` que significa que eles serão executados após `th:each`).
- Estará disponível para qualquer elemento filho da `<tr>` tag, como qualquer `<td>` elemento.

Thymeleaf oferece uma maneira de declarar variáveis locais sem iteração, usando o `th:with` atributo, e sua sintaxe é semelhante à das atribuições de valores de atributos:

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

Quando `th:with` processada, essa `firstPer` variável é criada como uma variável local e adicionada ao mapa de variáveis provenientes do contexto, para que fique disponível para avaliação junto com quaisquer outras variáveis declaradas no contexto, mas apenas dentro dos limites da tag que a contém `<div>`.

Você pode definir diversas variáveis ao mesmo tempo usando a sintaxe usual de atribuição múltipla:

```
<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>
```

O `th:with` atributo permite reutilizar variáveis definidas no mesmo atributo:

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

Vamos usar isso na página inicial do nosso supermercado! Lembra do código que escrevemos para gerar uma data formatada?

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

Bem, e se quiséssemos que isso "dd MMMM yyyy" realmente dependesse da localidade? Por exemplo, podemos querer adicionar a seguinte mensagem ao nosso `home_en.properties`:

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

...e um equivalente ao nosso `home_es.properties`:

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

Agora, vamos usar `th:with` para obter o formato de data localizado em uma variável e, em seguida, usá-lo em nossa `th:text` expressão:

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

Isso foi limpo e fácil. Na verdade, dado o fato de `th:with` ter um valor maior `precedence` que `th:text`, poderíamos ter resolvido tudo isso na `span` tag:

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

Você pode estar pensando: Precedência? Ainda não falamos sobre isso! Bem, não se preocupe porque é exatamente disso que trata o próximo capítulo.

10 Precedência de atributos

O que acontece quando você escreve mais de um `th:*` atributo na mesma tag? Por exemplo:

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

Esperaríamos que esse `th:each` atributo fosse executado antes `th:text` de obtermos os resultados desejados, mas dado o fato de que os padrões HTML/XML não dão nenhum tipo de significado à ordem em que os atributos em uma tag são escritos, uma *precedência* O mecanismo teve que ser estabelecido nos próprios atributos para ter certeza de que funcionaria conforme o esperado.

Assim, todos os atributos do Thymeleaf definem uma precedência numérica, que estabelece a ordem em que são executados na tag. Esta ordem é:

| Ordem | Recurso | Atributos |
|-------|-------------------------------------|--|
| 1 | Inclusão de fragmentos | <code>th:insert</code> <code>th:replace</code> |
| 2 | Iteração de fragmento | <code>th:each</code> |
| 3 | Avaliação condicional | <code>th:if</code> <code>th:unless</code> <code>th:switch</code> <code>th:case</code> |
| 4 | Definição de variável local | <code>th:object</code> <code>th:with</code> |
| 5 | Modificação geral de atributos | <code>th:attr</code> <code>th:attrprepend</code> <code>th:attrappend</code> |
| 6 | Modificação de atributo específico | <code>th:value</code> <code>th:href</code> <code>th:src</code> ... |
| 7 | Texto (modificação do corpo da tag) | <code>th:text</code> <code>th:utext</code> |
| 8 | Especificação do fragmento | <code>th:fragment</code> |
| 9 | Remoção de fragmentos | <code>th:remove</code> |

Este mecanismo de precedência significa que o fragmento de iteração acima dará exatamente os mesmos resultados se a posição do atributo `for` invertida (embora seja um pouco menos legível):

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

11 comentários e bloqueios

11.1. Comentários HTML/XML padrão

Comentários HTML/XML padrão `<!-- ... -->` podem ser usados em qualquer lugar nos modelos Thymeleaf. Qualquer coisa dentro desses comentários não será processada pelo Thymeleaf e será copiada literalmente para o resultado:

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

11.2. Blocos de comentários no nível do analisador Thymeleaf

Os blocos de comentários no nível do analisador são códigos que serão simplesmente removidos do modelo quando o Thymeleaf o analisar. Eles se parecem com isto:

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

O Thymeleaf removerá tudo entre `<!--/*` e `*/-->`, portanto, esses blocos de comentários também podem ser usados para exibir código quando um modelo estiver aberto estaticamente, sabendo que ele será removido quando o Thymeleaf o processar:

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

Isso pode ser muito útil para criar protótipos de tabelas com muitos `<tr>`'s, por exemplo:

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

11.3. Blocos de comentários somente para protótipo do Thymeleaf

O Thymeleaf permite a definição de blocos de comentários especiais marcados como comentários quando o modelo é aberto estaticamente (ou seja, como um protótipo), mas considerado marcação normal pelo Thymeleaf ao executar o modelo.

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

O sistema de análise do Thymeleaf simplesmente removerá os marcadores `<!--/*/` e `/*-->`, mas não seu conteúdo, que será deixado sem comentários. Portanto, ao executar o modelo, o Thymeleaf verá o seguinte:

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

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

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

Tal como acontece com os blocos de comentários no nível do analisador, esse recurso é independente do dialeto.

11.4. `th:block` Etiqueta sintética

O único processador de elementos do Thymeleaf (não um atributo) incluído nos Dialetos Padrão é `th:block`.

`th:block` é um mero contêiner de atributos que permite aos desenvolvedores de modelos especificar os atributos que desejarem. O Thymeleaf executará esses atributos e simplesmente fará o bloco, mas não seu conteúdo, desaparecer.

Portanto, pode ser útil, por exemplo, ao criar tabelas iteradas que exijam mais de uma `<tr>` para cada elemento:

```
<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>
```

E especialmente útil quando usado em combinação com blocos de comentários apenas de protótipo:

```
<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>
```

Observe como esta solução permite que os modelos sejam HTML válidos (sem necessidade de adicionar `<div>` blocos proibidos dentro de `<table>`) e ainda funciona bem quando abertos estaticamente em navegadores como protótipos!

12.1 Inlining de expressão

Embora o Dialet Padrão nos permita fazer quase tudo usando atributos de tags, há situações em que poderíamos preferir escrever expressões diretamente em nossos textos HTML. Por exemplo, poderíamos preferir escrever isto:

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

...em vez disso:

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

Expressões entre `[[...]]` ou `[(...)]` são consideradas **expressões embutidas** no Thymeleaf, e dentro delas podemos usar qualquer tipo de expressão que também seria válida em um atributo `th:text` ou `th:utext`.

Observe que, while `[[...]]` corresponde a `th:text` (ou seja, o resultado será *escapado de HTML*), `[(...)]` corresponde `th:utext` e não executará nenhum escape de HTML. Assim, com uma variável como `msg = 'This is great!'`, dado este fragmento:

```
<p>The message is "[(${msg})]"</p>
```

O resultado terá essas `` tags sem escape, então:

```
<p>The message is "This is <b>great!</b>"</p>
```

Considerando que se escapou como:

```
<p>The message is "[[ ${msg} ]]"</p>
```

O resultado será escapado em HTML:

```
<p>The message is "This is &lt;b&gt;great!&lt;/b&gt;"</p>
```

Observe que **o inlining de texto está ativo por padrão** no corpo de cada tag em nossa marcação – e não nas tags em si –, então não há nada que precisemos fazer para habilitá-lo.

Modelos embutidos versus modelos naturais

Se você vem de outros mecanismos de modelo nos quais essa forma de saída de texto é a norma, você pode estar se perguntando: *Por que não estamos fazendo isso desde o início? É menos código do que todos esses th:text atributos!*

Bem, tenha cuidado, porque embora você possa achar o inlining bastante interessante, você deve sempre lembrar que as expressões inline serão exibidas literalmente em seus arquivos HTML quando você os abrir estaticamente, então você provavelmente não será capaz de usá-los como protótipos de design não mais!

A diferença entre como um navegador exibiria estaticamente nosso fragmento de código sem usar inlining...

```
Hello, Sebastian!
```

...e usá-lo...

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

...é bastante claro em termos de utilidade do design.

Desativando inlining

Porém, esse mecanismo pode ser desativado, porque pode haver ocasiões em que desejamos gerar as sequências `[[...]]` ou `[(...)]` sem que seu conteúdo seja processado como uma expressão. Para isso, usaremos `th:inline="none"`:

```
<p th:inline="none">A double array looks like this: [[1, 2, 3], [4, 5]]!</p>
```

Isso resultará em:

```
<p>A double array looks like this: [[1, 2, 3], [4, 5]]!</p>
```

12.2 Inlining de texto

O *inlining de texto* é muito semelhante ao recurso *de inlining de expressão* que acabamos de ver, mas na verdade adiciona mais poder. Deve ser habilitado explicitamente com `th:inline="text"`.

O inlining de texto não apenas nos permite usar as mesmas *expressões inline* que acabamos de ver, mas na verdade processa *os corpos das tags* como se fossem modelos processados no `TEXT` modo de modelo, o que nos permite executar lógica de modelo baseada em texto (não apenas expressões de saída).

Veremos mais sobre isso no próximo capítulo sobre os *modos de modelo textual*.

12.3 Incorporação de JavaScript

O inlining de JavaScript permite uma melhor integração de `<script>` blocos JavaScript em modelos sendo processados no `HTML` modo de modelo.

Tal como acontece com *text inlining*, isso é na verdade equivalente a processar o conteúdo dos scripts como se fossem modelos no `JAVASCRIPT` modo de modelo e, portanto, todo o poder dos *modos de modelo textual* (veja o próximo capítulo) estará disponível. No entanto, nesta seção vamos nos concentrar em como podemos usá-lo para adicionar a saída de nossas expressões Thymeleaf em nossos blocos JavaScript.

Este modo deve ser habilitado explicitamente usando `th:inline="javascript"`:

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

Isso resultará em:

```
<script th:inline="javascript">
    ...
    var username = "Sebastian \"Fruity\" Applejuice";
    ...
</script>
```

Duas coisas importantes a serem observadas no código acima:

Primeiro, esse inlining de JavaScript não apenas produzirá o texto necessário, mas também o colocará entre aspas e escapará de JavaScript de seu conteúdo, para que os resultados da expressão sejam exibidos como um **literal de JavaScript bem formado**.

Segundo, isso está acontecendo porque estamos emitindo a `${session.user.name}` expressão como **escaped**, ou seja, usando uma expressão entre colchetes duplos: `[[${session.user.name}]]`. Se, em vez disso, usássemos *sem escape* como:

```
<script th:inline="javascript">
    ...
    var username = [(${session.user.name})];
```

```
</script>
```

O resultado seria parecido com:

```
<script th:inline="javascript">
...
var username = Sebastian "Fruity" Applejuice;
...
</script>
```

...que é um código JavaScript malformado. Mas gerar algo sem escape pode ser o que precisamos se estivermos construindo partes do nosso script por meio do acréscimo de expressões embutidas, por isso é bom ter essa ferramenta em mãos.

Modelos naturais JavaScript

A *inteligência* mencionada do mecanismo inlining de JavaScript vai muito além de apenas aplicar escape específico de JavaScript e gerar resultados de expressão como literais válidos.

Por exemplo, podemos agrupar nossas expressões embutidas (escapadas) em comentários JavaScript como:

```
<script th:inline="javascript">
...
var username = /*[${session.user.name}]*/ "Gertrud Kiwifruit";
...
</script>
```

E o Thymeleaf irá ignorar tudo o que escrevemos *após o comentário e antes do ponto e vírgula* (neste caso '`'Gertrud Kiwifruit'`), então o resultado da execução disso será exatamente igual a quando não estávamos usando os comentários de agrupamento:

```
<script th:inline="javascript">
...
var username = "Sebastian \"Fruity\" Applejuice";
...
</script>
```

Mas dê uma outra olhada cuidadosa no código do modelo original:

```
<script th:inline="javascript">
...
var username = /*[${session.user.name}]*/ "Gertrud Kiwifruit";
...
</script>
```

Observe como este é um código **JavaScript válido**. E será executado perfeitamente quando você abrir seu arquivo de modelo de maneira estática (sem executá-lo em um servidor).

Então o que temos aqui é uma maneira de fazer **templates naturais em JavaScript**!

Avaliação embutida avançada e serialização de JavaScript

Uma coisa importante a ser observada em relação ao inlining do JavaScript é que essa avaliação de expressão é inteligente e não se limita a Strings. Thymeleaf escreverá corretamente na sintaxe JavaScript os seguintes tipos de objetos:

- Cordas
- Números
- Booleanos
- Matrizes
- Coleções
- Mapas
- Beans (objetos com métodos `getter` e `setter`)

Por exemplo, se tivéssemos o seguinte código:

```
<script th:inline="javascript">
...
var user = /*[[${session.user}]]*/ null;
...
</script>
```

Essa \${session.user} expressão será avaliada como um User objeto e o Thymeleaf irá convertê-la corretamente para a sintaxe Javascript:

```
<script th:inline="javascript">
...
var user = {"age":null,"firstName":"John","lastName":"Apricot",
            "name":"John Apricot","nationality":"Antarctica"};
...
</script>
```

A forma como essa serialização do JavaScript é feita é por meio de uma implementação da `org.thymeleaf.standard.serializer.IStandardJavaScriptSerializer` interface, que pode ser configurada na instância que `StandardDialect` está sendo utilizada no template engine.

A implementação padrão deste mecanismo de serialização JS irá procurar a [biblioteca Jackson](#) no classpath e, se presente, irá utilizá-la. Caso contrário, aplicará um mecanismo de serialização integrado que cobre as necessidades da maioria dos cenários e produz resultados semelhantes (mas é menos flexível).

12.4 CSS embutido

Thymeleaf também permite o uso de inlining em `<style>` tags CSS, como:

```
<style th:inline="css">
...
</style>
```

Por exemplo, digamos que temos duas variáveis definidas com dois `String` valores diferentes:

```
classname = 'main elems'
align = 'center'
```

Poderíamos usá-los assim:

```
<style th:inline="css">
    .[[${classname}]] {
        text-align: [[${align}]];
    }
</style>
```

E o resultado seria:

```
<style th:inline="css">
    .main\ elems {
        text-align: center;
    }
</style>
```

Observe como o inlining CSS também traz alguma *inteligência*, assim como o JavaScript. Especificamente, as expressões geradas por meio de expressões *com escape*, como `[[${classname}]]` serão escapadas como **identificadores CSS**. É por isso que o nosso `classname = 'main elems'` se transformou `main\ elems` no fragmento de código acima.

Recursos avançados: modelos naturais CSS, etc.

De forma equivalente ao que foi explicado anteriormente para JavaScript, CSS inlining também permite que nossas `<style>` tags funcionem tanto estaticamente quanto dinamicamente, ou seja, como **modelos naturais de CSS** por meio de agrupamento de expressões inline em comentários. Ver:

```
<style th:inline="css">
  .main\ elems {
    text-align: /*[[ ${align} ]]*/ left;
  }
</style>
```

13 modos de modelo textual

13.1 Sintaxe textual

Três dos *modos de modelo* do Thymeleaf são considerados **textuais**: **TEXT** e **JAVASCRIPT**. **CSS** Isso os diferencia dos modos de modelo de marcação: **HTML** e **XML**.

A principal diferença entre os modos de modelo *textual* e os de marcação é que em um modelo textual não há tags nas quais inserir lógica na forma de atributos, portanto, temos que contar com outros mecanismos.

O primeiro e mais básico desses mecanismos é o **inlining**, que já detalhamos no capítulo anterior. A sintaxe inlining é a maneira mais simples de gerar resultados de expressões no modo de modelo textual, portanto, este é um modelo perfeitamente válido para um email de texto.

```
Dear [(${name})],  
  
Please find attached the results of the report you requested  
with name "[(${report.name})]".  
  
Sincerely,  
The Reporter.
```

Mesmo sem tags, o exemplo acima é um template Thymeleaf completo e válido que pode ser executado no **TEXT** modo template.

Mas para incluir uma lógica mais complexa do que meras *expressões de saída*, precisamos de uma nova sintaxe não baseada em tags:

```
[# th:each="item : ${items}"]  
- [(${item})]  
[/]
```

Qual é na verdade a versão *condensada* do mais detalhado:

```
[#th:block th:each="item : ${items}"]  
- [#th:block th:utext="${item}" /]  
[/th:block]
```

Observe como esta nova sintaxe é baseada em elementos (ou seja, tags processáveis) que são declarados como `[#element ...]` em vez de `<element ...>`. Os elementos são abertos como `[#element ...]` e fechados como `[/element]`, e tags independentes podem ser declaradas minimizando o elemento aberto de uma / forma quase equivalente às tags XML: `[#element ... /]`.

O Dialeto Padrão contém apenas um processador para um desses elementos: o já conhecido `th:block`, embora pudéssemos estender isso em nossos dialetos e criar novos elementos da maneira usual. Além disso, o `th:block` elemento (`[#th:block ...] ... [/th:block]`) pode ser abreviado como uma string vazia (`[# ...] ... [/]`), portanto o bloco acima é na verdade equivalente a:

```
[# th:each="item : ${items}"]  
- [# th:utext="${item}" /]  
[/]
```

E dado que `[# th:utext="${item}" /]` é equivalente a uma *expressão embutida sem escape*, poderíamos simplesmente usá-la para ter menos código. Assim terminamos com o primeiro fragmento de código que vimos acima:

```
[# th:each="item : ${items}"]  
- [(${item})]  
[/]
```

Observe que a *sintaxe textual* requer equilíbrio total dos elementos (sem tags não fechadas) e atributos entre aspas – é mais no estilo XML do que no estilo HTML.

Vamos dar uma olhada em um exemplo mais completo de **TEXT** modelo, um modelo de email de *texto simples*:

```
Dear [(${customer.name})],
```

```
This is the list of our products:
```

```
[# th:each="prod : ${products}"]
- [${prod.name}]. Price: [${prod.price}] EUR/kg
[]
```

```
Thanks,
The Thymeleaf Shop
```

Após a execução, o resultado disso poderia ser algo como:

```
Dear Mary Ann Blueberry,
```

```
This is the list of our products:
```

```
- Apricots. Price: 1.12 EUR/kg
- Bananas. Price: 1.78 EUR/kg
- Apples. Price: 0.85 EUR/kg
- Watermelon. Price: 1.91 EUR/kg
```

```
Thanks,
The Thymeleaf Shop
```

E outro exemplo em **JAVASCRIPT modo template**, um `greeter.js` arquivo, processamos como um template textual e cujo resultado chamamos de nossas páginas HTML. *Observe que este não é um <script> bloco em um modelo HTML, mas um .js arquivo sendo processado como modelo por si só:*

```
var greeter = function() {
    var username = [[${session.user.name}]];
    [# th:each="salut : ${salutations}"]
        alert([[${salut}]] + " " + username);
    []
};

};
```

Após a execução, o resultado disso poderia ser algo como:

```
var greeter = function() {
    var username = "Bertrand \\"Crunchy\\" Pear";
    alert("Hello" + " " + username);
    alert("Ol\u00fch" + " " + username);
    alert("Hola" + " " + username);
};

};
```

Atributos de elemento com escape

Para evitar interações com partes do modelo que podem ser processadas em outros modos (por exemplo, `text` -mode inlining dentro de um HTML modelo), o Thymeleaf 3.0 permite que os atributos em elementos em sua *sintaxe textual* sejam escapados. Então:

- Os atributos no `TEXT` modo de modelo não terão *escape HTML*.
- Os atributos no `JAVASCRIPT` modo de modelo não terão *escape de JavaScript*.
- Os atributos no `CSS` modo de modelo não terão *escape CSS*.

Portanto, isso seria perfeitamente aceitável em um `TEXT` modelo -mode (observe o `>`):

```
[# th:if="${120<user.age}"]
    Congratulations!
[]
```

É claro que isso `<` não faria sentido em um modelo de *texto real*, mas é uma boa ideia se estivermos processando um modelo HTML com um `th:inline="text"` bloco contendo o código acima e quisermos ter certeza de que nosso navegador não considerará isso `<user.age` o nome de uma tag open ao abrir estaticamente o arquivo como um protótipo.

13.2 Extensibilidade

Uma das vantagens desta sintaxe é que ela é tão extensível quanto a de *marcação*. Os desenvolvedores ainda podem definir seus próprios dialetos com elementos e atributos personalizados, aplicar um prefixo a eles (opcionalmente) e então usá-los em modos de modelo textual:

```
[#myorg:doSomething myorg:importantattr="211"]some text[/myorg:doSomething]
```

13.3 Blocos de comentários somente de protótipo textual: adicionando código

Os modos de modelo `JAVASCRIPT` e `CSS` (não disponíveis para `TEXT`) permitem incluir código entre uma sintaxe de comentário especial `/*[...]*/` para que o Thymeleaf remova automaticamente o comentário desse código ao processar o modelo:

```
var x = 23;  
  
/*[+  
  
var msg = "This is a working application";  
  
+]*/  
  
var f = function() {  
    ...
```

Será executado como:

```
var x = 23;  
  
var msg = "This is a working application";  
  
var f = function() {  
    ...
```

Você pode incluir expressões dentro desses comentários e elas serão avaliadas:

```
var x = 23;  
  
/*[+  
  
var msg = "Hello, " + [[${session.user.name}]];  
  
+]*/  
  
var f = function() {  
    ...
```

13.4 Blocos de comentários em nível de analisador textual: removendo código

De maneira semelhante à dos blocos de comentários apenas de protótipo, todos os três modos de modelo textual (`TEXT`, `JAVASCRIPT` e `CSS`) tornam possível instruir o Thymeleaf a remover o código entre marcas especiais `/*[- */` e `/* - */`, assim:

```
var x = 23;  
  
/*[- */  
  
var msg = "This is shown only when executed statically!";  
  
/* - */  
  
var f = function() {  
    ...
```

Ou isto, no **TEXT** modo:

```
...
/*[- Note the user is obtained from the session, which must exist -]*/
Welcome [(${session.user.name})]!
...
```

13.5 Modelos naturais de JavaScript e CSS

Como visto no capítulo anterior, JavaScript e CSS inlining oferecem a possibilidade de incluir expressões inline dentro de comentários JavaScript/CSS, como:

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

...que é JavaScript válido e, uma vez executado, pode ser semelhante a:

```
...
var username = "John Apricot";
...
```

Este mesmo *truque* de colocar expressões embutidas dentro de comentários pode, de fato, ser usado para toda a sintaxe do modo textual:

```
/*[# th:if="${user.admin}"]*
 alert('Welcome admin');
/*[]*/
```

Esse alerta no código acima será mostrado quando o template for aberto estaticamente – pois é JavaScript 100% válido – e também quando o template for executado se o usuário for administrador. É equivalente a:

```
[# th:if="${user.admin}"]
 alert('Welcome admin');
[]
```

...que é na verdade o código para o qual a versão inicial é convertida durante a análise do modelo.

Observe, entretanto, que agrupar elementos em comentários não limpa as linhas em que eles residem (à direita até que a ; seja encontrado), como fazem as expressões de saída embutidas. Esse comportamento é reservado apenas para expressões de saída embutidas.

Assim, o Thymeleaf 3.0 permite o desenvolvimento de **scripts JavaScript complexos e folhas de estilo CSS na forma de modelos naturais**, válidos tanto como *protótipo* quanto como *modelo de trabalho*.

14 Mais algumas páginas para nossa mercearia

Agora que sabemos muito sobre como usar o Thymeleaf, podemos adicionar algumas novas páginas ao nosso site para gerenciamento de pedidos.

Observe que nos concentraremos no código HTML, mas você pode dar uma olhada no código-fonte incluído se quiser ver os controladores correspondentes.

14.1 Lista de Pedidos

Vamos começar criando uma página de lista de pedidos `/WEB-INF/templates/order/list.html`:

```
<!DOCTYPE html>

<html 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>
```

Não há nada aqui que deva nos surpreender, exceto este pouco da magia OGNL:

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

O que isso faz é, para cada linha de pedido (`OrderLine` objeto) do pedido, multiplicar suas propriedades `purchasePrice` and `amount` (chamando os métodos `getPurchasePrice()` and correspondentes `getAmount()`) e retornar o resultado em uma lista de números, posteriormente agregada pela `#aggregates.sum(...)` função para obter o total do pedido preço.

Você tem que amar o poder do OGNL.

14.2 Detalhes do Pedido

Agora, a página de detalhes do pedido, na qual faremos uso intenso da sintaxe do asterisco:

```
<!DOCTYPE html>

<html 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>
```

Não há muita novidade aqui, exceto esta seleção de objetos aninhados:

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

    ...

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

```
...  
</body>
```

...o que torna isso `*{name}` equivalente a:

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

15 Mais sobre configuração

15.1 Resolvedores de Modelo

Para nosso Good Thymes Virtual Grocery, escolhemos uma `ITemplateResolver` implementação chamada `WebApplicationTemplateResolver` que nos permitiu obter templates como recursos dos recursos da aplicação (o `Contexto Servlet` em um webapp baseado em Servlet).

Além de nos dar a capacidade de criar nosso próprio resovedor de modelo implementando `ITemplateResolver`, o Thymeleaf, inclui quatro implementações prontas para uso:

- `org.thymeleaf.templateresolver.ClassLoaderTemplateResolver`, que resolve modelos como recursos do carregador de classe, como:

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

- `org.thymeleaf.templateresolver.FileTemplateResolver`, que resolve modelos como arquivos do sistema de arquivos, como:

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

- `org.thymeleaf.templateresolver.UrlTemplateResolver`, que resolve modelos como URLs (mesmo os não locais), como:

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

- `org.thymeleaf.templateresolver.StringTemplateResolver`, que resolve os modelos diretamente conforme `String` especificado como `template` (ou *nome do modelo*, que neste caso é obviamente muito mais do que um mero nome):

```
return new StringReader(templateName);
```

Todas as implementações pré-agrupadas permitem `ITemplateResolver` o mesmo conjunto de parâmetros de configuração, que incluem:

- Prefixo e sufixo (como já visto):

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

- Aliases de modelos que permitem o uso de nomes de modelos que não correspondem diretamente aos nomes de arquivos. Se existirem sufixo/prefixo e alias, o alias será aplicado antes do prefixo/sufixo:

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

- Codificação a ser aplicada na leitura de templates:

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

- Modo de modelo a ser usado:

```
// Default is HTML
templateResolver.setTemplateMode("XML");
```

- Modo padrão para cache de modelos e padrões para definir se modelos específicos podem ser armazenados em cache ou não:

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

- TTL em milissegundos para entradas de cache de modelo analisadas originadas neste resovedor de modelo. Se não for definido, a única maneira de remover uma entrada do cache será exceder o tamanho máximo do cache (a entrada mais antiga será removida).

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

Os pacotes de integração Thymeleaf + Spring oferecem uma `SpringResourceTemplateResolver` implementação que utiliza toda a infraestrutura Spring para acesso e leitura de recursos em aplicações, e que é a implementação recomendada em aplicações habilitadas para Spring.

Encadeando Resolvedores de Modelo

Além disso, um Template Engine pode especificar vários resolvedores de templates, caso em que uma ordem pode ser estabelecida entre eles para resolução de templates para que, caso o primeiro não consiga resolver o template, o segundo seja solicitado, e assim por diante:

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

WebApplicationTemplateResolver webApplicationTemplateResolver =
    new WebApplicationTemplateResolver(application);
webApplicationTemplateResolver.setOrder(Integer.valueOf(2));

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

Quando vários resolvedores de modelo são aplicados, é recomendado especificar padrões para cada resolvedor de modelo para que o Thymeleaf possa descartar rapidamente os resolvedores de modelo que não se destinam a resolver o modelo, melhorando o desempenho. Fazer isso não é um requisito, mas uma recomendação:

```
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");

WebApplicationTemplateResolver webApplicationTemplateResolver =
    new WebApplicationTemplateResolver(application);
webApplicationTemplateResolver.setOrder(Integer.valueOf(2));
```

Se esses *padrões resolvíveis* não forem especificados, estaremos contando com os recursos específicos de cada uma das `ITemplateResolver` implementações que estamos usando. Observe que nem todas as implementações podem ser capazes de determinar a existência de um modelo antes de resolver e, portanto, podem sempre considerar um modelo como *resolvível* e quebrar a cadeia de resolução (não permitindo que outros resolvedores verifiquem o mesmo modelo), mas então ser incapazes de ler o recurso real.

Todas as `ITemplateResolver` implementações incluídas no núcleo do Thymeleaf incluem um mecanismo que nos permitirá fazer com que os resolvedores *realmente verifiquem* se um recurso existe antes de considerá-lo *resolvível*. É a `checkExistence` bandeira, que funciona assim:

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

Este `checkExistence` sinalizador força o resolvedor a realizar uma *verificação real* da existência de recursos durante a fase de resolução (e permite que o resolvedor seguinte na cadeia seja chamado se a verificação de existência retornar falso). Embora isso possa parecer bom em todos os casos, na maioria dos casos isso significará um acesso duplo ao próprio recurso (uma vez para verificar a existência, outra vez para lê-lo) e pode ser um problema de desempenho em alguns cenários, por exemplo, baseado em URL remoto recursos de modelo – um possível problema de desempenho que pode, de qualquer forma, ser amplamente mitigado pelo uso do cache de modelo (nesse caso, os modelos só serão *resolvidos* na primeira vez em que forem acessados).

15.2 Resolvedores de Mensagens

Não especificamos explicitamente uma implementação do Message Resolver para nosso aplicativo Grocery e, como foi explicado anteriormente, isso significava que a implementação usada era um `org.thymeleaf.messageresolver.StandardMessageResolver` objeto.

`StandardMessageResolver` é a implementação padrão da `IMessageResolver` interface, mas poderíamos criar a nossa própria se quiséssemos, adaptada às necessidades específicas da nossa aplicação.

Os pacotes de integração Thymeleaf + Spring oferecem por padrão uma `IMessageResolver` implementação que utiliza a forma padrão Spring de recuperar mensagens externalizadas, usando `MessageSource` beans declarados no Spring Application Context.

Resolvedor de mensagens padrão

Então, como você `StandardMessageResolver` procura as mensagens solicitadas em um modelo específico?

Se o nome do modelo for `home` e estiver localizado em `/WEB-INF/templates/home.html`, e a localidade solicitada for, `gl_ES` então este resolvedor procurará mensagens nos seguintes arquivos, nesta ordem:

- `/WEB-INF/templates/home_gl_ES.properties`
- `/WEB-INF/templates/home_gl.properties`
- `/WEB-INF/templates/home.properties`

Consulte a documentação JavaDoc da `StandardMessageResolver` classe para obter mais detalhes sobre como funciona o mecanismo completo de resolução de mensagens.

Configurando resolvedores de mensagens

E se quiséssemos adicionar um resolvedor de mensagens (ou mais) ao Template Engine? Fácil:

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

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

E por que quereríamos ter mais de um resolvedor de mensagens? Pela mesma razão que os resolvedores de modelo: os resolvedores de mensagens são ordenados e se o primeiro não conseguir resolver uma mensagem específica, o segundo será solicitado, depois o terceiro, etc.

15.3 Serviços de Conversão

O serviço de conversão que nos permite realizar operações de conversão e formatação de dados por meio da sintaxe *de colchetes duplos* `${{...}}` () é na verdade um recurso do Dialet Padrão, não do próprio Thymeleaf Template Engine.

Dessa forma, a maneira de configurá-lo é definindo nossa implementação personalizada da `IStandardConversionService` interface diretamente na instância `StandardDialect` que está sendo configurada no mecanismo de modelo. Como:

```
IStandardConversionService customConversionService = ...

StandardDialect dialect = new StandardDialect();
dialect.setConversionService(customConversionService);

templateEngine.setDialect(dialect);
```

Observe que os pacotes `thymeleaf-spring3` e `thymeleaf-spring4` contêm o `SpringStandardDialect`, e esse dialeto já vem pré-configurado com uma implementação `IStandardConversionService` que integra a infraestrutura do próprio *Serviço de Conversão do Spring ao Thymeleaf*.

15.4 Registro

Thymeleaf presta muita atenção ao logging e sempre tenta oferecer o máximo de informações úteis através de sua interface de logging.

A biblioteca de log usada é `slf4j`, a que na verdade atua como uma ponte para qualquer implementação de log que queiramos usar em nosso aplicativo (por exemplo, `log4j`).

As classes Thymeleaf registrarão informações de nível e , dependendo do nível de detalhe que desejamos, e além do registro geral, usará três registradores especiais associados à classe TemplateEngine que podemos configurar separadamente para diferentes propósitos

TRACE : DEBUG INFO

- `org.thymeleaf.TemplateEngine.CONFIG` gerará configuração detalhada da biblioteca durante a inicialização.
- `org.thymeleaf.TemplateEngine.TIMER` produzirá informações sobre o tempo necessário para processar cada modelo (útil para benchmarking!)
- `org.thymeleaf.TemplateEngine.cache` é o prefixo de um conjunto de registradores que geram informações específicas sobre os caches. Embora os nomes dos cache loggers sejam configuráveis pelo usuário e, portanto, possam mudar, por padrão eles são:
 - `org.thymeleaf.TemplateEngine.cache TEMPLATE_CACHE`
 - `org.thymeleaf.TemplateEngine.cache EXPRESSION_CACHE`

Um exemplo de configuração para a infraestrutura de registro do Thymeleaf, usando `log4j`, poderia ser:

```
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 Cache de modelos

Thymeleaf funciona graças a um conjunto de analisadores – para marcação e texto – que analisam modelos em sequências de eventos (tag aberta, texto, tag fechada, comentário, etc.) e uma série de processadores – um para cada tipo de comportamento que precisa ser aplicado – que modifica a sequência de eventos analisada do modelo para criar os resultados que esperamos, combinando o modelo original com nossos dados.

Também inclui – por padrão – um cache que armazena modelos analisados; a sequência de eventos resultantes da leitura e análise de arquivos de modelo antes de processá-los. Isso é especialmente útil ao trabalhar em um aplicativo Web e baseia-se nos seguintes conceitos:

- A entrada/saída é quase sempre a parte mais lenta de qualquer aplicativo. O processamento na memória é extremamente rápido em comparação.
- Clonar uma sequência de eventos existente na memória é sempre muito mais rápido do que ler um arquivo de modelo, analisá-lo e criar uma nova sequência de eventos para ele.
- Os aplicativos da Web geralmente possuem apenas algumas dezenas de modelos.
- Os arquivos de modelo são de tamanho pequeno a médio e não são modificados enquanto o aplicativo está em execução.

Tudo isso leva à ideia de que armazenar em cache os templates mais usados em uma aplicação web é viável sem desperdiçar grandes quantidades de memória, e também que economizará muito tempo que seria gasto em operações de entrada/saída em um pequeno conjunto de arquivos isso, na verdade, nunca muda.

E como podemos assumir o controle desse cache? Primeiro, já aprendemos que podemos habilitá-lo ou desabilitá-lo no Template Resolver, mesmo agindo apenas em templates específicos:

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

Além disso, poderíamos modificar sua configuração estabelecendo nosso próprio objeto *Cache Manager*, que poderia ser uma instância da *StandardCacheManager* implementação padrão:

```
// Default is 200
StandardCacheManager cacheManager = new StandardCacheManager();
cacheManager.setTemplateCacheMaxSize(100);
...
templateEngine.setCacheManager(cacheManager);
```

Consulte a API javadoc para `org.thymeleaf.cache.StandardCacheManager` obter mais informações sobre como configurar os caches.

As entradas podem ser removidas manualmente do cache de modelos:

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

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

17 Lógica de modelo dissociada

17.1 Lógica dissociada: O conceito

Até agora trabalhamos para nossa Mercearia com templates feitos da *maneira usual*, com a lógica sendo inserida em nossos templates em forma de atributos.

Mas o Thymeleaf também nos permite *dissociar completamente* a marcação do modelo de sua lógica, permitindo a criação de **modelos de marcação completamente sem lógica** nos modos de modelo `HTML` e `.XML`.

A idéia principal é que a lógica do modelo seja definida em um *arquivo lógico* separado (mais exatamente um *recurso lógico*, pois não precisa ser um *arquivo*). Por padrão, esse recurso lógico será um arquivo adicional localizado no mesmo local (por exemplo, pasta) que o arquivo de modelo, com o mesmo nome, mas com `.th.xml` extensão:

```
/templates  
+->/home.html  
+->/home.th.xml
```

Portanto, o `home.html` arquivo pode ser completamente sem lógica. Pode ser assim:

```
<!DOCTYPE html>  
<html>  
  <body>  
    <table id="usersTable">  
      <tr>  
        <td class="username">Jeremy Grapefruit</td>  
        <td class="usertype">Normal User</td>  
      </tr>  
      <tr>  
        <td class="username">Alice Watermelon</td>  
        <td class="usertype">Administrator</td>  
      </tr>  
    </table>  
  </body>  
</html>
```

A absolutamente nenhum código Thymeleaf lá. Este é um arquivo de modelo que um designer sem conhecimento do Thymeleaf ou de modelos poderia ter criado, editado e/ou compreendido. Ou um fragmento de HTML fornecido por algum sistema externo sem nenhum gancho do Thymeleaf.

Vamos agora transformar esse `home.html` modelo em um modelo Thymeleaf criando nosso `home.th.xml` arquivo adicional como este:

```
<?xml version="1.0"?>  
<thlogic>  
  <attr sel="#usersTable" th:remove="all-but-first">  
    <attr sel="/tr[0]" th:each="user : ${users}">  
      <attr sel="td.username" th:text="${user.name}" />  
      <attr sel="td.usertype" th:text="#{|user.type.${user.type}|}" />  
    </attr>  
  </attr>  
</thlogic>
```

Aqui podemos ver muitas `<attr>` tags dentro de um `thlogic` bloco. Essas `<attr>` tags realizam *injeção de atributos* em nós do template original selecionados por meio de seus `sel` atributos, que contêm *seletores de marcação* Thymeleaf (na verdade, *seletores de marcação AttoParser*).

Observe também que `<attr>` as tags podem ser aninhadas para que seus seletores sejam *anexados*. O `sel="/tr[0]"` acima, por exemplo, será processado como `sel="#usersTable/tr[0]"`. E o seletor do nome de usuário `<td>` será processado como `sel="#usersTable/tr[0]//td.username"`.

Assim, uma vez mesclados, os dois arquivos vistos acima serão iguais a:

```
<!DOCTYPE html>  
<html>
```

```

<body>
  <table id="usersTable" th:remove="all-but-first">
    <tr th:each="user : ${users}">
      <td class="username" th:text="${user.name}">Jeremy Grapefruit</td>
      <td class="usertype" th:text="#{${user.type}}">Normal User</td>
    </tr>
    <tr>
      <td class="username">Alice Watermelon</td>
      <td class="usertype">Administrator</td>
    </tr>
  </table>
</body>
</html>

```

Isso parece mais familiar e, na verdade, é menos *detalhado* do que criar dois arquivos separados. Mas a vantagem dos *modelos desacoplados* é que podemos dar aos nossos modelos total independência do Thymeleaf e, portanto, melhor capacidade de manutenção do ponto de vista do design.

É claro que alguns *contratos* entre designers ou desenvolvedores ainda serão necessários – por exemplo, o fato de que os usuários <table> precisarão de um `id="usersTable"` –, mas em muitos cenários um modelo HTML puro será um artefato de comunicação muito melhor entre as equipes de design e desenvolvimento.

17.2 Configurando modelos desacoplados

Habilitando modelos desacoplados

A lógica dissociada não será esperada para todos os modelos por padrão. Em vez disso, os resolvidores de modelos configurados (implementações de `ITemplateResolver`) precisarão marcar especificamente os modelos que resolvem como *usando lógica dissociada*.

Exceto `StringTemplateResolver` (que não permite lógica desacoplada), todas as outras implementações prontas para uso `ITemplateResolver` fornecerão um sinalizador chamado `useDecoupledLogic` que marcará todos os modelos resolvidos por esse resolvidor como potencialmente tendo toda ou parte de sua lógica vivendo em um recurso separado :

```

final WebApplicationTemplateResolver templateResolver =
  new WebApplicationTemplateResolver(application);
...
templateResolver.setUseDecoupledLogic(true);

```

Misturando lógica acoplada e desacoplada

A lógica de modelo desacoplada, quando habilitada, não é um requisito. Quando habilitado, significa que o mecanismo irá *procurar* um recurso contendo lógica desacoplada, analisando-o e mesclando-o com o modelo original, se existir. Nenhum erro será gerado se o recurso lógico desacoplado não existir.

Além disso, no mesmo modelo podemos misturar lógica *acoplada* e *desacoplada*, por exemplo, adicionando alguns atributos do Thymeleaf no arquivo de modelo original, mas deixando outros para o arquivo lógico desacoplado separado. O caso mais comum para isso é usar o novo `th:ref` atributo (na v3.0).

17.3 O atributo `th:ref`

`th:ref` é apenas um atributo de marcador. Ele não faz nada do ponto de vista de processamento e simplesmente desaparece quando o template é processado, mas sua utilidade reside no fato de atuar como uma *referência de marcação*, ou seja, pode ser resolvido por nome a partir de um *seletor de marcação* assim como um *nome de tag* ou um *fragmento* (`th:fragment`).

Então, se tivermos um seletor como:

```
<attr sel="whatever" .../>
```

Isso corresponderá a:

- Quaisquer `<whatever>` tags.

- Quaisquer tags com um `th:fragment="whatever"` atributo.
- Quaisquer tags com um `th:ref="whatever"` atributo.

Qual é a vantagem de `th:ref`, por exemplo, usar um atributo HTML puro `id`? Apenas o fato de que talvez não queiramos adicionar tantos `id` atributos `class` às nossas tags para atuarem como *âncoras lógicas*, o que pode acabar poluindo nossa saída.

E no mesmo sentido, qual é a desvantagem de `th:ref`? Bem, obviamente estariamos adicionando um pouco da lógica do Thymeleaf (“lógica”) aos nossos modelos.

Observe que esta aplicabilidade do `th:ref` atributo **não se aplica apenas a arquivos de modelo lógico desacoplados**: ele funciona da mesma forma em outros tipos de cenários, como em expressões de fragmento (`~{...}`).

17.4 Impacto no desempenho de modelos dissociados

O impacto é extremamente pequeno. Quando um modelo resolvido é marcado para usar lógica desacoplada e não é armazenado em cache, o recurso lógico do modelo será resolvido primeiro, analisado e processado em uma sequência de instruções na memória: basicamente uma lista de atributos a serem injetados em cada seletor de marcação.

Mas esta é a única *etapa adicional* necessária porque, depois disso, o modelo real será analisado e, enquanto ele é analisado, esses atributos serão injetados *instantaneamente* pelo próprio analisador, graças aos recursos avançados para seleção de nós no AttoParser. Portanto, os nós analisados sairão do analisador como se tivessem seus atributos injetados escritos no arquivo de modelo original.

A maior vantagem disso? Quando um modelo é configurado para ser armazenado em cache, ele já será armazenado em cache contendo os atributos injetados. Portanto, a sobrecarga do uso de *modelos desacoplados* para modelos armazenáveis em cache, uma vez armazenados em cache, será absolutamente zero.

17.5 Resolução de lógica desacoplada

A forma como o Thymeleaf resolve os recursos lógicos desacoplados correspondentes a cada template é configurável pelo usuário. É determinado por um ponto de extensão, o `org.thymeleaf.templateparser.markup.decoupled.IDecoupledTemplateLogicResolver`, para o qual é fornecida uma *implementação padrão* `StandardDecoupledTemplateLogicResolver`:

O que essa implementação padrão faz?

- Primeiro, aplica a `prefix` e a `suffix` ao *nome base* do recurso do modelo (obtido por meio de seu `ITemplateResource#getBaseName()` método). Tanto o prefixo quanto o sufixo podem ser configurados e, por padrão, o prefixo estará vazio e o sufixo será `.th.xml`.
- Segundo, ele pede ao recurso do modelo para resolver um *recurso relativo* com o nome computado por meio de seu `ITemplateResource#relative(String relativeLocation)` método.

A implementação específica a `IDecoupledTemplateLogicResolver` ser usada pode ser configurada facilmente `TemplateEngine`:

```
final StandardDecoupledTemplateLogicResolver decoupledresolver =
    new StandardDecoupledTemplateLogicResolver();
decoupledresolver.setPrefix("../viewlogic/");
...
templateEngine.setDecoupledTemplateLogicResolver(decoupledresolver);
```

18 Apêndice A: Objetos Básicos de Expressão

Alguns objetos e mapas de variáveis estão sempre disponíveis para serem invocados. Vamos vê-los:

Objetos básicos

- **#ctx**: o objeto de contexto. Uma implementação `org.thymeleaf.context.IContext` ou `org.thymeleaf.context.IWebContext` dependendo do nosso ambiente (autônomo ou web).

Observe `#vars` e `#root` são sinônimos para o mesmo objeto, mas o uso `#ctx` é recomendado.

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

${#ctx.locale}
${#ctx.variableNames}

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

${#ctx.request}
${#ctx.response}
${#ctx.session}
${#ctx.servletContext}
```

- **#locale**: acesso direto ao `java.util.Locale` associado à solicitação atual.

```
${#locale}
```

Namespaces de contexto da Web para atributos de solicitação/sessão, etc.

Ao utilizar o Thymeleaf em um ambiente web, podemos utilizar uma série de atalhos para acessar parâmetros de solicitação, atributos de sessão e atributos de aplicação:

Observe que estes não são *objetos de contexto*, mas mapas adicionados ao contexto como variáveis, portanto, os acessamos sem `#`. De alguma forma, eles atuam como *namespaces*.

- **param**: para recuperar parâmetros de solicitação. `${param.foo}` é um `String[]` com os valores do `foo` parâmetro de solicitação, portanto `${param.foo[0]}` normalmente será usado para obter o primeiro valor.

```
/*
 * ======
 * 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')}
...
```

- **sessão**: para recuperar atributos da sessão.

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

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

- **application** : para recuperar atributos de contexto de aplicativo/servlet.

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

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

Observe que não há **necessidade de especificar um namespace para acessar os atributos de solicitação** (em oposição aos *parâmetros de solicitação*) porque todos os atributos de solicitação são automaticamente adicionados ao contexto como variáveis na raiz do contexto:

```
${myRequestAttribute}
```

19 Apêndice B: Objetos Utilitários de Expressão

Informações de execução

- **#execInfo**: objeto de expressão que fornece informações úteis sobre o modelo que está sendo processado dentro das expressões padrão do Thymeleaf.

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

/*
 * Return the name and mode of the 'leaf' template. This means the template
 * from where the events being processed were parsed. So if this piece of
 * code is not in the root template "A" but on a fragment being inserted
 * into "A" from another template called "B", this will return "B" as a
 * name, and B's mode as template mode.
 */
${#execInfo.templateName}
${#execInfo.templateMode}

/*
 * Return the name and mode of the 'root' template. This means the template
 * that the template engine was originally asked to process. So if this
 * piece of code is not in the root template "A" but on a fragment being
 * inserted into "A" from another template called "B", this will still
 * return "A" and A's template mode.
 */
${#execInfo.processedTemplateName}
${#execInfo.processedTemplateMode}

/*
 * Return the stacks (actually, List<String> or List<TemplateMode>) of
 * templates being processed. The first element will be the
 * 'processedTemplate' (the root one), the last one will be the 'leaf'
 * template, and in the middle all the fragments inserted in nested
 * manner to reach the leaf from the root will appear.
 */
${#execInfo.templateNames}
${#execInfo.templateModes}

/*
 * Return the stack of templates being processed similarly (and in the
 * same order) to 'templateNames' and 'templateModes', but returning
 * a List<TemplateData> with the full template metadata.
 */
${#execInfo.templateStack}
```

Mensagens

- **#messages**: métodos utilitários para obtenção de mensagens externalizadas dentro de expressões de variáveis, da mesma forma que seriam obtidas pela `#{...}` sintaxe.

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

/*
 * Obtain externalized messages. Can receive a single key, a key plus arguments,
 * or an array/list/set of keys (in which case it will return an array/list/set of
 * externalized messages).
 * If a message is not found, a default message (like '??msgKey??') is returned.
 */
```

```

*/
${#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)}

/*
 * Obtain externalized messages or null. Null is returned instead of a default
 * message if a message for the specified key is not found.
 */
${#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)}

```

URIs/URLs

- **#uris** : objeto utilitário para realizar operações de URI/URL (especialmente escape/unescaping) dentro de expressões padrão do Thymeleaf.

```

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

/*
 * Escape/Unescape as a URI/URL path
 */
${#uris.escapePath(uri)}
${#uris.escapePath(uri, encoding)}
${#uris.unescapePath(uri)}
${#uris.unescapePath(uri, encoding)}

/*
 * Escape/Unescape as a URI/URL path segment (between '/' symbols)
 */
${#uris.escapePathSegment(uri)}
${#uris.escapePathSegment(uri, encoding)}
${#uris.unescapePathSegment(uri)}
${#uris.unescapePathSegment(uri, encoding)}

/*
 * Escape/Unescape as a Fragment Identifier (#frag)
 */
${#uris.escapeFragmentId(uri)}
${#uris.escapeFragmentId(uri, encoding)}
${#uris.unescapeFragmentId(uri)}
${#uris.unescapeFragmentId(uri, encoding)}

/*
 * Escape/Unescape as a Query Parameter (?var=value)
 */
${#uris.escapeQueryParam(uri)}
${#uris.escapeQueryParam(uri, encoding)}
${#uris.unescapeQueryParam(uri)}
${#uris.unescapeQueryParam(uri, encoding)}

```

Conversões

- **#conversions** : objeto utilitário que permite a execução do *Serviço de Conversão* em qualquer ponto de um template:

```

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

/*
 * Execute the desired conversion of the 'object' value into the
 * specified class.
 */
${#conversions.convert(object, 'java.util.TimeZone')}
${#conversions.convert(object, targetClass)}

```

datas

- **#dates** : métodos utilitários para `java.util.Date` objetos:

```

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

/*
 * Format date with the standard locale format
 * Also works with arrays, lists or sets
 */
${#dates.format(date)}
${#dates.arrayFormat(datesArray)}
${#dates.listFormat(datesList)}
${#dates.setFormat(datesSet)}

/*
 * Format date with the ISO8601 format
 * Also works with arrays, lists or sets
 */
${#dates.formatISO(date)}
${#dates.arrayFormatISO(datesArray)}
${#dates.listFormatISO(datesList)}
${#dates.setFormatISO(datesSet)}

/*
 * Format date with the specified pattern
 * Also works with arrays, lists or sets
 */
${#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')}

/*
 * Obtain date properties
 * Also works with arrays, lists or sets
 */
${#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.

/*
 * Create date (java.util.Date) objects from its components
 */
${#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)}

/*
 * Create a date (java.util.Date) object for the current date and time
 */
${#dates.createNow()}

${#dates.createNowForTimeZone()}

/*
 * Create a date (java.util.Date) object for the current date (time set to 00:00)
 */
${#dates.createToday()}

${#dates.createTodayForTimeZone()}

```

Calendários

- **#calendars**: análogo a **#dates**, mas para `java.util.Calendar` objetos:

```

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

/*
 * Format calendar with the standard locale format
 * Also works with arrays, lists or sets
 */
${#calendars.format(cal)}
${#calendars.arrayFormat(calArray)}
${#calendars.listFormat(calList)}
${#calendars.setFormat(calSet)}

/*
 * Format calendar with the ISO8601 format
 * Also works with arrays, lists or sets
 */
${#calendars.formatISO(cal)}
${#calendars.arrayFormatISO(calArray)}
${#calendars.listFormatISO(calList)}
${#calendars.setFormatISO(calSet)}

/*
 * Format calendar with the specified pattern
 * Also works with arrays, lists or sets
 */
${#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')}

/*
 * Obtain calendar properties
 * Also works with arrays, lists or sets
 */
${#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.minute(date)}        // also arrayMinute(...), listMinute(...), etc.
${#calendars.second(date)}        // also arraySecond(...), listSecond(...), etc.
${#calendars.millisecond(date)}   // also arrayMillisecond(...), listMillisecond(...), etc.

/*
 * Create calendar (java.util.Calendar) objects from its components
 */

```

```
*/
${#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)}

${#calendars.createForTimeZone(year,month,day,timeZone)}
${#calendars.createForTimeZone(year,month,day,hour,minute,timeZone)}
${#calendars.createForTimeZone(year,month,day,hour,minute,second,timeZone)}
${#calendars.createForTimeZone(year,month,day,hour,minute,second,millisecond,timeZone)}

/*
 * Create a calendar (java.util.Calendar) object for the current date and time
 */
${#calendars.createNow()}

${#calendars.createNowForTimeZone()}

/*
 * Create a calendar (java.util.Calendar) object for the current date (time set to 00:00)
 */
${#calendars.createToday()}

${#calendars.createTodayForTimeZone()}
```

Temporais (java.time)

- **#temporals**: lida com objetos de data/hora da `java.time` API JDK8+:

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

/*
 *
 * Format date with the standard locale format
 * Also works with arrays, lists or sets
 */
${#temporals.format(temporal)}
${#temporals.arrayFormat(temporalsArray)}
${#temporals.listFormat(temporalsList)}
${#temporals.setFormat(temporalsSet)}

/*
 * Format date with the standard format for the provided locale
 * Also works with arrays, lists or sets
 */
${#temporals.format(temporal, locale)}
${#temporals.arrayFormat(temporalsArray, locale)}
${#temporals.listFormat(temporalsList, locale)}
${#temporals.setFormat(temporalsSet, locale)}

/*
 * Format date with the specified pattern
 * SHORT, MEDIUM, LONG and FULL can also be specified to used the default java.time.format.FormatStyle patterns
 * Also works with arrays, lists or sets
 */
${#temporals.format(temporal, 'dd/MMM/yyyy HH:mm')}
${#temporals.format(temporal, 'dd/MMM/yyyy HH:mm', 'Europe/Paris')}
${#temporals.arrayFormat(temporalsArray, 'dd/MMM/yyyy HH:mm')}
${#temporals.listFormat(temporalsList, 'dd/MMM/yyyy HH:mm')}
${#temporals.setFormat(temporalsSet, 'dd/MMM/yyyy HH:mm')}

/*
 * Format date with the specified pattern and locale
 * Also works with arrays, lists or sets
 */
${#temporals.format(temporal, 'dd/MMM/yyyy HH:mm', locale)}
${#temporals.arrayFormat(temporalsArray, 'dd/MMM/yyyy HH:mm', locale)}
${#temporals.listFormat(temporalsList, 'dd/MMM/yyyy HH:mm', locale)}
${#temporals.setFormat(temporalsSet, 'dd/MMM/yyyy HH:mm', locale)}
```

```

/*
 * Format date with ISO-8601 format
 * Also works with arrays, lists or sets
 */
${#temporals.formatISO(temporal)}
${#temporals.arrayFormatISO(temporalsArray)}
${#temporals.listFormatISO(temporalsList)}
${#temporals.setFormatISO(temporalsSet)}

/*
 * Obtain date properties
 * Also works with arrays, lists or sets
 */
${#temporals.day(temporal)}                                // also arrayDay(...), listDay(...), etc.
${#temporals.month(temporal)}                             // also arrayMonth(...), listMonth(...), etc.
${#temporals.monthName(temporal)}                         // also arrayMonthName(...), listMonthName(...), etc.
${#temporals.monthNameShort(temporal)}                    // also arrayMonthNameShort(...), listMonthNameShort(...), etc.
${#temporals.year(temporal)}                               // also arrayYear(...), listYear(...), etc.
${#temporals.dayOfWeek(temporal)}                         // also arrayDayOfWeek(...), listDayOfWeek(...), etc.
${#temporals.dayOfWeekName(temporal)}                     // also arrayDayOfWeekName(...), listDayOfWeekName(...), etc.
${#temporals.dayOfWeekNameShort(temporal)}                // also arrayDayOfWeekNameShort(...), listDayOfWeekNameShort(...), etc.
${#temporals.hour(temporal)}                               // also arrayHour(...), listHour(...), etc.
${#temporals.minute(temporal)}                            // also arrayMinute(...), listMinute(...), etc.
${#temporals.second(temporal)}                            // also arraySecond(...), listSecond(...), etc.
${#temporals.nanosecond(temporal)}                       // also arrayNanosecond(...), listNanosecond(...), etc.

/*
 * Create temporal (java.time.Temporal) objects from its components
 */
${#temporals.create(year,month,day)}                      // return a instance of java.time.LocalDate
${#temporals.create(year,month,day,hour,minute)}           // return a instance of java.time.LocalDateTime
${#temporals.create(year,month,day,hour,minute,second)}    // return a instance of java.time.LocalDateTime
${#temporals.create(year,month,day,hour,minute,second,nanosecond)} // return a instance of java.time.LocalDateTime

/*
 * Create a temporal (java.time.Temporal) object for the current date and time
 */
${#temporals.createNow()}                                 // return a instance of java.time.LocalDateTime
${#temporals.createNowForTimeZone(zoneId)}                // return a instance of java.time.ZonedDateTime
${#temporals.createToday()}                               // return a instance of java.time.LocalDate
${#temporals.createTodayForTimeZone(zoneId)}              // return a instance of java.time.LocalDate

/*
 * Create a temporal (java.time.Temporal) object for the provided date
 */
${#temporals.createDate(isoDate)}                        // return a instance of java.time.LocalDate
${#temporals.createDateTime(isoDate)}                     // return a instance of java.time.LocalDateTime
${#temporals.createDate(isoDate, pattern)}                // return a instance of java.time.LocalDate
${#temporals.createDateTime(isoDate, pattern)}             // return a instance of java.time.LocalDateTime

```

Números

- **#numbers**: métodos utilitários para objetos numéricos:

```

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

/*
 * =====
 * Formatting integer numbers
 * =====
 */

/*
 * Set minimum integer digits.
 * Also works with arrays, lists or sets
 */
${#numbers.formatInteger(num,3)}
${#numbers.arrayFormatInteger(numArray,3)}

```

```

${#numbers.listFormatInteger(numList,3)}
${#numbers.setFormatInteger(numSet,3)}

/*
 * Set minimum integer digits and thousands separator:
 * 'POINT', 'COMMA', 'WHITESPACE', 'NONE' or 'DEFAULT' (by locale).
 * Also works with arrays, lists or sets
 */
${#numbers.formatInteger(num,3,'POINT')}
${#numbers.arrayFormatInteger(numArray,3,'POINT')}
${#numbers.listFormatInteger(numList,3,'POINT')}
${#numbers.setFormatInteger(numSet,3,'POINT')}

/*
 * =====
 * Formatting decimal numbers
 * =====
 */

/*
 * Set minimum integer digits and (exact) decimal digits.
 * Also works with arrays, lists or sets
 */
${#numbers.formatDecimal(num,3,2)}
${#numbers.arrayFormatDecimal(numArray,3,2)}
${#numbers.listFormatDecimal(numList,3,2)}
${#numbers.setFormatDecimal(numSet,3,2)}

/*
 * Set minimum integer digits and (exact) decimal digits, and also decimal separator.
 * Also works with arrays, lists or sets
 */
${#numbers.formatDecimal(num,3,2,'COMMA')}
${#numbers.arrayFormatDecimal(numArray,3,2,'COMMA')}
${#numbers.listFormatDecimal(numList,3,2,'COMMA')}
${#numbers.setFormatDecimal(numSet,3,2,'COMMA')}

/*
 * Set minimum integer digits and (exact) decimal digits, and also thousands and
 * decimal separator.
 * Also works with arrays, lists or sets
 */
${#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')}

/*
 * =====
 * Formatting currencies
 * =====
 */

${#numbers.formatCurrency(num)}
${#numbers.arrayFormatCurrency(numArray)}
${#numbers.listFormatCurrency(numList)}
${#numbers.setFormatCurrency(numSet)}

/*
 * =====
 * Formatting percentages
 * =====
 */

${#numbers.formatPercent(num)}
${#numbers.arrayFormatPercent(numArray)}
${#numbers.listFormatPercent(numList)}
${#numbers.setFormatPercent(numSet)}

/*
 * Set minimum integer digits and (exact) decimal digits.
 */

```



```

${#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 and prepend
 * Also works with arrays, lists or sets
 */
${#strings.prepend(str,prefix)}                  // also array*, list* and set*
${#strings.append(str,suffix)}                   // also array*, list* and set*

/*
 * Change case
 * Also works with arrays, lists or sets
 */
${#strings.toUpperCase(name)}                   // also array*, list* and set*
${#strings.toLowerCase(name)}                   // also array*, list* and set*

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

/*
 * Trim
 * Also works with arrays, lists or sets
 */
${#strings.trim(str)}                          // also array*, list* and set*

/*
 * Compute length
 * Also works with arrays, lists or sets
 */
${#strings.length(str)}                       // also array*, list* and set*

/*
 * Abbreviate text making it have a maximum size of n. If text is bigger, it
 * will be clipped and finished in "..."
 * Also works with arrays, lists or sets
 */
${#strings.abbreviate(str,10)}                 // also array*, list* and set*

/*
 * Convert the first character to upper-case (and vice-versa)
 */
${#strings.capitalize(str)}                   // also array*, list* and set*
${#strings.unCapitalize(str)}                // also array*, list* and set*

/*
 * Convert the first character of every word to upper-case
 */
${#strings.capitalizeWords(str)}              // also array*, list* and set*
${#strings.capitalizeWords(str,delimiters)}    // also array*, list* and set*

/*
 * Escape the string
 */
${#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-safe comparison and concatenation
 */
${#strings.equals(first, second)}            ${#strings.equalsIgnoreCase(first, second)}
${#strings.concat(values...)}                ${#strings.concatReplaceNulls(nullValue, values...)}

```

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

Objetos

- **#objects**: métodos utilitários para objetos em geral

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

/*
 * Return obj if it is not null, and default otherwise
 * Also works with arrays, lists or sets
 */
${#objects.nullSafe(obj,default)}
${#objects.arrayNullSafe(objArray,default)}
${#objects.listNullSafe(objList,default)}
${#objects.setNullSafe(objSet,default)}
```

Booleanos

- **#bools**: métodos utilitários para avaliação booleana

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

/*
 * Evaluate a condition in the same way that it would be evaluated in a th:if tag
 * (see conditional evaluation chapter afterwards).
 * Also works with arrays, lists or sets
 */
${#bools.isTrue(obj)}
${#bools.arrayIsTrue(objArray)}
${#bools.listIsTrue(objList)}
${#bools.setIsTrue(objSet)}

/*
 * Evaluate with negation
 * Also works with arrays, lists or sets
 */
${#bools.isFalse(cond)}
${#bools.arrayIsFalse(condArray)}
${#bools.listIsFalse(condList)}
${#bools.setIsFalse(condSet)}

/*
 * Evaluate and apply AND operator
 * Receive an array, a list or a set as parameter
 */
${#bools.arrayAnd(condArray)}
${#bools.listAnd(condList)}
${#bools.setAnd(condSet)}

/*
 * Evaluate and apply OR operator
 * Receive an array, a list or a set as parameter
 */
${#bools.arrayOr(condArray)}
${#bools.listOr(condList)}
${#bools.setOr(condSet)}
```

Matrizes

- **#arrays**: métodos utilitários para arrays

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

/*
 * Converts to array, trying to infer array component class.
 * Note that if resulting array is empty, or if the elements
 * of the target object are not all of the same class,
 * this method will return Object[].
 */
${#arrays.toArray(object)}

/*
 * Convert to arrays of the specified component class.
 */
${#arrays.toStringArray(object)}
${#arrays.toIntegerArray(object)}
${#arrays.toLongArray(object)}
${#arrays.toDoubleArray(object)}
${#arrays.toFloatArray(object)}
${#arrays.toBooleanArray(object)}

/*
 * Compute length
 */
${#arrays.length(array)}

/*
 * Check whether array is empty
 */
${#arrays.isEmpty(array)}

/*
 * Check if element or elements are contained in array
 */
${#arrays.contains(array, element)}
${#arrays.containsAll(array, elements)}
```

Listas

- **#lists**: métodos utilitários para listas

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

/*
 * Converts to list
 */
${#lists.toList(object)}

/*
 * Compute size
 */
${#lists.size(list)}

/*
 * Check whether list is empty
 */
${#lists.isEmpty(list)}
```

```

/* Check if element or elements are contained in list
 */
${#lists.contains(list, element)}
${#lists.containsAll(list, elements)}

/*
 * Sort a copy of the given list. The members of the list must implement
 * comparable or you must define a comparator.
 */
${#lists.sort(list)}
${#lists.sort(list, comparator)}

```

Conjuntos

- **#sets**: métodos utilitários para conjuntos

```

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

/*
 * Converts to set
 */
${#sets.toSet(object)}

/*
 * Compute size
 */
${#sets.size(set)}

/*
 * Check whether set is empty
 */
${#sets.isEmpty(set)}

/*
 * Check if element or elements are contained in set
 */
${#sets.contains(set, element)}
${#sets.containsAll(set, elements)}

```

Mapas

- **#maps**: métodos utilitários para mapas

```

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

/*
 * Compute size
 */
${#maps.size(map)}

/*
 * Check whether map is empty
 */
${#maps.isEmpty(map)}

/*
 * Check if key/s or value/s are contained in maps
 */
${#maps.containsKey(map, key)}
${#maps.containsAllKeys(map, keys)}

```

```
 ${#maps.containsValue(map, value)}
 ${#maps.containsAllValues(map, value)}
```

Agregados

- **#agregados**: métodos utilitários para criar agregações em arrays ou coleções

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

/*
 * Compute sum. Returns null if array or collection is empty
 */
${#aggregates.sum(array)}
${#aggregates.sum(collection)}

/*
 * Compute average. Returns null if array or collection is empty
 */
${#aggregates.avg(array)}
${#aggregates.avg(collection)}
```

IDs

- **#ids**: métodos utilitários para lidar com `id` atributos que podem ser repetidos (por exemplo, como resultado de uma iteração).

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

/*
 * Normally used in th:id attributes, for appending a counter to the id attribute value
 * so that it remains unique even when involved in an iteration process.
 */
${#ids.seq('someId')}

/*
 * Normally used in th:for attributes in <label> tags, so that these labels can refer to Ids
 * generated by means of the #ids.seq(...) function.
 *
 * Depending on whether the <label> goes before or after the element with the #ids.seq(...)
 * function, the "next" (label goes before "seq") or the "prev" function (label goes after
 * "seq") function should be called.
 */
${#ids.next('someId')}
${#ids.prev('someId')}
```

20 Apêndice C: Sintaxe do Seletor de Marcação

Os seletores de marcação do Thymeleaf são emprestados diretamente da biblioteca de análise do Thymeleaf: [AttoParser](#).

A sintaxe desses seletores tem grandes semelhanças com a dos seletores em XPath, CSS e jQuery, o que os torna fáceis de usar para a maioria dos usuários. Você pode dar uma olhada na referência de sintaxe completa na [documentação do AttoParser](#).

Por exemplo, o seletor a seguir selecionará every `<div>` com class `content`, em todas as posições dentro da marcação (observe que isso não é tão conciso quanto poderia ser, continue lendo para saber o porquê):

```
<div th:insert="~{mytemplate :: //div[@class='content']}>...</div>
```

A sintaxe básica inclui:

- `/x` significa filhos diretos do nó atual com nome `x`.
- `//x` significa filhos do nó atual com nome `x`, em qualquer profundidade.
- `x[@z="v"]` significa elementos com nome `x` e um atributo chamado `z` com valor “`v`”.
- `x[@z1="v1" and @z2="v2"]` significa elementos com nome `x` e atributos `z1` e `z2` com valores “`v1`” e “`v2`”, respectivamente.
- `x[i]` significa elemento com nome `x` posicionado no número `i` entre seus irmãos.
- `x[@z="v"][i]` significa elementos com nome `x`, atributo `z` com valor “`v`” e posicionados no número `i` entre seus irmãos que também atendem a esta condição.

Mas uma sintaxe mais concisa também pode ser usada:

- `x` é exatamente equivalente a `//x` (pesquisar um elemento com nome ou referência `x` em qualquer nível de profundidade, sendo uma [referência th:ref](#) um ou um `th:fragment` atributo).
- Seletores também são permitidos sem nome/referência de elemento, desde que incluem uma especificação de argumentos. Portanto `[@class='oneclass']`, é um seletor válido que procura quaisquer elementos (tags) com um atributo de classe com value “`oneclass`”.

Recursos avançados de seleção de atributos:

- Além de `=` (igual), outros operadores de comparação também são válidos: `!=` (diferente), `^=` (começa com) e `$=` (termina com). Por exemplo: `x[@class^='section']` significa elementos com nome `x` e um valor para o atributo `class` que começa com `section`.
- Os atributos podem ser especificados começando com `@` (estilo XPath) e sem (estilo jQuery). Então `x[z='v']` é equivalente a `x[@z='v']`.
- Modificadores de múltiplos atributos podem ser unidos tanto com `and` (estilo XPath) quanto encadeando múltiplos modificadores (estilo jQuery). Então `x[@z1='v1' and @z2='v2']`, na verdade, é equivalente a `x[@z1='v1'][@z2='v2']` (e também a `x[z1='v1'][z2='v2']`).

Seletores diretos do tipo jQuery:

- `x.oneclass` é equivalente a `x[class='oneclass']`.
- `.oneclass` é equivalente a `[class='oneclass']`.
- `x#oneid` é equivalente a `x[id='oneid']`.
- `#oneid` é equivalente a `[id='oneid']`.
- `x%oneref` significa `<x>` tags que possuem um atributo `th:ref="oneref"` ou `th:fragment="oneref"`.
- `%oneref` significa qualquer tag que tenha um atributo `th:ref="oneref"` ou `th:fragment="oneref"`. Observe que isso é equivalente simplesmente `oneref` porque referências podem ser usadas em vez de nomes de elementos.
- Seletores diretos e seletores de atributos podem ser misturados: `a.external[@href^='https']`.

Portanto, a expressão do seletor de marcação acima:

```
<div th:insert="~{mytemplate :: //div[@class='content']}>...</div>
```

Poderia ser escrito como:

```
<div th:insert="~{mytemplate :: div.content}">...</div>
```

Examinando um exemplo diferente, este:

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

Procurará uma `th:fragment="myfrag"` assinatura de fragmento (ou `th:ref` referências). Mas também procuraria tags com nome, `myfrag` se existissem (o que não existe, em HTML). Observe a diferença com:

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

...que irá realmente procurar por quaisquer elementos com `class="myfrag"`, sem se preocupar com `th:fragment` assinaturas (ou `th:ref` referências).

Correspondência de classe multivalorada

Os Seletores de Marcação entendem que o atributo de classe é **multivalorado**, e portanto permitem a aplicação de seletores neste atributo mesmo que o elemento possua vários valores de classe.

Por exemplo, `div.two` irá corresponder `<div class="one two three" />`