Thymeleaf

Fala, Dev! Seja bem-vindo ao Do Back ao Front - Programação Full Stack. Meu nome é Olival e resolvi assumir o desafio de traduzir a parte prática e essencial desta documentação. Se você está começando com Thymeleaf, esse documento vai te ajudar bastante.

Mas, antes de começar a lê-lo na integra, gostaria de recomendar que assistisse a esse vídeo, onde explico um pouco na prática. Você não precisa fazer a prática, só assistir e ver como funciona, para depois vir aqui e aprofundar, tirar dúvidas, e usar de acordo com a sua finalidade.

Assista aqui: https://youtu.be/Ddb8j-cyDgY?si=RayhEkgLi8lNjurc

Observação: Thymeleaf neste vídeo é usado com Spring Boot, Java. Aqui abaixo, deixo o link do meu curso de Spring Boot com Java e MySQL, onde te explico em detalhes como criar uma API, e depois, você pode usar o Thymeleaf para criar as páginas web, os templates, para criar a sua aplicação Full Stack. Lembrando que está na promoção. Com um detalhe, na compra deste curso de java web, no valor de 22.90, você pode adquirir qualquer outro curso de java no valor de 9.90/cada, incluindo os cursos de java para desktop, java no frontend para desktop, javascript, inteligência artificial generativa, para acelerar ainda mais os estudos, a imersão java na prática do básico ao avançado com orientação a objetos e banco de dados, que é uma ótima base para quem está começando a estudar java do zero, e a mentoria de carreira backend, onde você fica por dentro do cenário de vagas e estudos de programação. Tudo com acesso vitalício, certificados e aulas gravadas. Confira os links abaixo:

Cursos de Java:

- Java Web com Spring Boot: https://pay.kiwify.com.br/mXnHYBK
- Java na Prática: https://pay.kiwify.com.br/gmU7HPU
- Java no Frontend com Swing: https://pay.kiwify.com.br/RXED8AQ
- Mentoria de Programação Backend Java: https://pay.kiwify.com.br/EbMHbry
- Inteligência Artificial Generativa: https://pay.kiwify.com.br/I4CthI2
- JavaScript na Prática: https://pay.kiwify.com.br/PFtSYRv

Aproveite que logo, logo, sai a nova versão da apostila, com mais conteúdos atualizados. Ah! Inscreva-se no canal que toda terça e quinta tem conteúdo de java novo. Inscreva-se aqui:

Não esqueça de compartilhar com os amigos. Bons estudos.

3. Usando Textos

3.1 Boas-Vindas

Nossa primeira tarefa será criar uma página inicial para o 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 arquivo `/WEB-INF/templates/home.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>

    <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 ele não inclui nenhuma tag que não seja HTML (os navegadores ignoram todos os atributos que não entendem, como `th:text`). Mas você também pode perceber que este template não é realmente um documento HTML5 válido, porque esses atributos não padrão que estamos usando na forma `th:*` não são permitidos pela especificação HTML5. Na verdade, estamos até adicionando um atributo `xmlns:th` à nossa tag `<html>`, algo absolutamente fora do padrão HTML5:

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

...o que não tem influência alguma no processamento do template, mas funciona como um encantamento que impede que nosso IDE reclame sobre a falta de uma definição de namespace para todos esses atributos `th:*`.

Então, e se quiséssemos tornar este template válido segundo o HTML5? Simples: mude para a sintaxe de atributo `data` do Thymeleaf, usando o prefixo `data-` para nomes de atributos e separadores de hífen (-) em vez de dois pontos (:):

```
<!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 personalizados prefixados com `data-` são permitidos pela especificação HTML5, então, com o código acima, nosso template seria um documento HTML5 válido. Ambas as notações são completamente equivalentes e intercambiáveis, mas, para o bem da simplicidade e da concisão dos exemplos de código, este tutorial usará a notação de namespace (`th:*`). Além disso, a notação `th:*` é mais geral e permitida em todos os modos de template do Thymeleaf (XML, TEXT...), enquanto a notação `data-` é permitida apenas no modo HTML.

Usando th:text e externalizando texto

Externalizar texto é o processo de extrair fragmentos de código do template para fora dos arquivos de template, para que possam ser mantidos em arquivos separados (geralmente arquivos `.properties`) e que possam ser facilmente substituídos por textos equivalentes escritos em outras línguas (um processo chamado internacionalização ou simplesmente i18n). Fragmentos de texto externalizados são geralmente chamados de "mensagens".

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

** Bem-vindo à nossa loja de mercearia! **

O que podemos ver aqui são, na verdade, duas características diferentes do Dialeto Padrão do Thymeleaf:

1. O atributo `th:text`, que avalia sua expressão de valor e define o resultado como o conteúdo da tag hospedeira, substituindo efetivamente o texto "Bem-vindo à nossa loja de mercearia!" que vemos no código.
2. A expressão `#{home.welcome}`, especificada na Sintaxe de Expressão Padrão, que instrui que o texto a ser usado pelo atributo `th:text` deve ser a mensagem com a chave `home.welcome`, correspondente a qualquer localidade com a qual estamos processando o template.

Agora, onde está esse texto externalizado? A localização do texto externalizado no Thymeleaf é totalmente configurável e dependerá da implementação específica de `org.thymeleaf.messageresolver.IMessageResolver` que está sendo usada. Normalmente, será usada uma implementação baseada em arquivos `.properties`, 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 o nosso motor de templates durante a inicialização, o que significa que nossa aplicação está usando o Resolvedor de Mensagens Padrão, implementado por `org.thymeleaf.messageresolver.StandardMessageResolver`. O resolvedor de mensagens padrão espera encontrar mensagens para `/WEB-INF/templates/home.html` em arquivos `.properties` na mesma pasta e com o mesmo nome do template, 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 português (Brasil).
- `/WEB-INF/templates/home.properties` para textos padrão (se a localidade não for correspondida).

Vamos dar uma olhada no nosso arquivo `home_es.properties`:

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

Isso é tudo o que precisamos para fazer o Thymeleaf processar nosso template. Vamos então criar nosso controlador Home.

Contexto

Para processar nosso template, vamos criar uma classe `HomeController` que implementa a interface `IGTVGController` que vimos anteriormente:

```
public class HomeController implements IGTVGController {

    public void process(
            final HttpServletRequest request, final HttpServletResponse response,
            final ServletContext servletContext, final ITemplateEngine templateEngine)
            throws Exception {

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

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

    }

}
```

A primeira coisa que vemos é a criação de um contexto. Um contexto do Thymeleaf é um **objeto** que implementa a interface `org.thymeleaf.context.IContext`. Os contextos devem conter todos os dados necessários para a execução do motor de templates em um mapa de variáveis, além de referenciar a localidade que deve ser usada para as 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 dessa interface, `org.thymeleaf.context.IWebContext`, destinada a ser usada em aplicações web baseadas na Servlet API (como o SpringMVC).

```
public interface IWebContext extends IContext {

    public HttpServletRequest getRequest();
    public HttpServletResponse getResponse();
    public HttpSession getSession();
    public ServletContext getServletContext();

}
```

A biblioteca principal do Thymeleaf oferece uma implementação para cada uma dessas 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, precisamos usá-lo, pois o uso de um `ServletContextTemplateResolver` exige que utilizemos um contexto que implemente `IWebContext`.

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

Apenas três desses quatro argumentos do construtor são obrigatórios, pois a localidade padrão do sistema será usada se nenhuma for especificada (embora você nunca deva permitir que isso aconteça em aplicações reais). Existem algumas expressões especializadas que poderemos usar para obter os parâmetros da requisição e os atributos de requisição, sessão e aplicação do `WebContext` em nossos templates. Por exemplo:

- `${x}` retornará uma variável `x` armazenada no contexto do Thymeleaf ou como um atributo de requisição.
- `${param.x}` retornará um parâmetro de requisição chamado `x` (que pode ter múltiplos valores).
- `${session.x}` retornará um atributo de sessão chamado `x`.
- `${application.x}` retornará um atributo do contexto do servlet chamado `x`.

Executando o Template Engine

Com nosso objeto de contexto pronto, agora podemos instruir o motor de templates a processar o template (pelo seu nome) usando o contexto, e passar a ele um `response writer` para que a resposta possa ser escrita nele:

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

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 não Escapado

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

home.welcome=Welcome to our fantastic grocery store!

Se executarmos este template como antes, obteremos:

** Welcome to our fantastic grocery store! **

O que não é exatamente o que esperávamos, porque nossa tag `<b>` foi escapada e, portanto, será exibida no navegador. Este é o comportamento padrão do atributo `th:text`. Se quisermos que o Thymeleaf respeite nossas tags HTML e não as escape, teremos que usar um atributo diferente: `th:utext` (para "texto não escapado"):

** Welcome to our grocery store! **

Isso exibirá nossa mensagem exatamente como a queremos:

** Welcome to our fantastic grocery store! **

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
```

Em primeiro lugar, teremos que modificar nosso controlador para que adicionemos essa data como uma variável de contexto:

```
public void process(
            final HttpServletRequest request, final HttpServletResponse response,
            final ServletContext servletContext, final ITemplateEngine templateEngine)
            throws Exception {

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

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

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

}
```

4 Sintaxe de Expressão Padrão.

Faremos uma pequena pausa no desenvolvimento da nossa loja virtual de supermercado para aprender sobre uma das partes mais importantes do Dialeto Padrão do Thymeleaf: a sintaxe de expressão padrão do Thymeleaf. Já vimos dois tipos de valores de atributo válidos expressos nessa sintaxe: expressões de mensagem e expressões de variável:

** Welcome to our grocery store! **

** Today is: 13 february 2011 **

Mas há mais tipos de expressões e detalhes mais interessantes para aprender sobre as que já conhecemos. Primeiro, vamos ver um resumo rápido das características da Sintaxe de Expressão Padrão:

Expressões Simples:

- **Expressões de Variável:** ${...}
- **Expressões de Variável de Seleção:** *{...}
- **Expressões de Mensagem:** #{...}
- **Expressões de URL de Link:** @{...}
- **Expressões de Fragmento:** ~{...}

Literals:

- **Literals de Texto:** 'um texto', 'Outro!',…
- **Literals Numéricos:** 0, 34, 3.0, 12.3,…
- **Literals Booleanos:** true, false
- **Literals Nulos:** null
- **Tokens Literais:** um, algumtexto, principal,…

Operações de Texto:

- **Concatenação de String:** +
- **Substituições Literais:** |O nome é ${name}|

Operações Aritméticas:

- **Operadores binários:** +, -, *, /, %
- **Sinal de menos (operador unário):**

Operações Booleanas:

- **Operadores binários:** and, or
- **Negação Boolean:** !, not

Comparações e Igualdades:

- **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:

- **No-Operation: _**

Todas essas características podem ser combinadas e aninhadas:

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

4.1 Mensagens.

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

** Welcome to our grocery store! **

…a isso:

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

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

** ¡Bienvenido a nuestra tienda de comestibles, John Apricot! **

Isso significa que precisaríamos adicionar um parâmetro à nossa mensagem. Assim como isto:

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

Os parâmetros são especificados de acordo com a sintaxe padrão do <code>java.text.MessageFormat</code>, o que significa que você pode formatar números e datas conforme especificado na documentação da API para classes no pacote <code>java.text.*</code>. Para especificar um valor para nosso parâmetro, e dado um atributo de sessão HTTP chamado user, poderíamos ter:

** Welcome to our grocery store, Sebastian Pepper! **

Observe que o uso de <code>th
</code> aqui significa que a mensagem formatada não será escapada. Este exemplo assume que <code>user.name</code> já está escapado. Vários parâmetros podem ser especificados, separados por vírgulas. A chave da mensagem em si pode vir de uma variável:

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

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 informações detalhadas sobre a sintaxe e os recursos do OGNL, você deve ler o **Guia da Linguagem OGNL**. Em aplicações habilitadas para Spring MVC, o OGNL será substituído pelo SpringEL, mas sua sintaxe é muito semelhante à do OGNL (na verdade, exatamente a mesma para a maioria dos casos comuns).

A partir da sintaxe do OGNL, sabemos que a expressão em:

** Today is: 13 february 2011. **

…é, na verdade, equivalente a isto:

ctx.getVariable("today");

Mas o OGNL permite que criemos expressões muito mais poderosas, e é assim que isso:

** Welcome to our grocery store, Sebastian Pepper! **

…obtém o nome do usuário executando:

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

Mas a navegação de métodos getter é apenas um dos recursos do OGNL. Vamos ver mais:

```
/*
 * 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('-')}
```

Objetos Básicos de Expressão

Ao avaliar expressões OGNL nas variáveis de contexto, alguns objetos são disponibilizados para expressões, proporcionando 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**: o locale de contexto.
- **#request**: (apenas em Contextos Web) o objeto HttpServletRequest.
- **#response**: (apenas em Contextos Web) o objeto HttpServletResponse.
- **#session**: (apenas em Contextos Web) o objeto HttpSession.
- **#servletContext**: (apenas em Contextos Web) o objeto ServletContext.

Assim, podemos fazer isso:

Established locale country: US.

Você pode ler a referência completa desses objetos no **Apêndice A - Objetos de Utilidade de Expressão**. Além desses objetos básicos, o Thymeleaf nos oferecerá um conjunto de objetos de utilidade que nos ajudarão a realizar tarefas comuns em nossas expressões.

- **#execInfo**: informações sobre o template 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 partes de URLs/URIs.
- **#conversions**: métodos para executar o serviço de conversão configurado (se houver).
- **#dates**: métodos para objetos java.util.Date: formatação, extração de componentes, etc.
- **#calendars**: análogo a #dates, mas para objetos java.util.Calendar.
- **#numbers**: métodos para formatar objetos numéricos.
- **#strings**: métodos para objetos String: contém, startsWith, prepend/append, etc.
- **#objects**: métodos para objetos em geral.
- **#bools**: métodos para avaliação booleana.
- **#arrays**: métodos para arrays.
- **#lists**: métodos para listas.
- **#sets**: métodos para conjuntos.
- **#maps**: métodos para mapas.
- **#aggregates**: métodos para criar agregados em arrays ou coleções.
- **#ids**: métodos para lidar com atributos de id que podem ser repetidos (por exemplo, como resultado de uma iteração).

Você pode verificar quais funções são oferecidas por cada um desses objetos de utilidade no **Apêndice B**.

Reformatando datas em nossa página inicial

Agora que conhecemos esses objetos de utilidade, 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(request, servletContext, request.getLocale());
ctx.setVariable("today", dateFormat.format(cal.getTime()));

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

…podemos fazer apenas isso:

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

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

…e então realizar 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)

As expressões de variáveis podem ser escritas não apenas como ${...}, mas também como *{...}. No entanto, há uma diferença importante: a sintaxe de asterisco avalia expressões em objetos selecionados, em vez de no contexto inteiro. Ou seja, enquanto não houver um objeto selecionado, as sintaxes de cifrão e asterisco fazem exatamente a mesma coisa.

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

```
  <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>
```

Claro, as sintaxes de cifrão e asterisco podem ser misturadas:

```
<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 está em vigor, o objeto selecionado também estará disponível para expressões de cifrão como a variável de expressão `#object`:

```
<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 mencionado, se nenhuma seleção de objeto tiver sido realizada, as sintaxes de cifrão e asterisco são 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 Link URLs

Devido à sua importância, as URLs são elementos de primeira classe em templates de aplicações web, e o Dialeto Padrão do Thymeleaf possui uma sintaxe especial para elas, a sintaxe @: @{...}. Existem diferentes tipos de URLs:

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

O processamento real dessas expressões e sua conversão para as URLs que serão geradas é realizado por implementações da interface `org.thymeleaf.linkbuilder.ILinkBuilder` que são registradas no objeto `ITemplateEngine` em uso.

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

Vamos utilizar essa nova sintaxe. Conheça o atributo `th:href`:

```
<!-- 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 importantes a serem observadas:

- **`th:href` é um atributo modificador**: uma vez processado, ele calculará a URL do link a ser usada e definirá esse valor no atributo `href` da tag `<a>`.
- **Podemos usar expressões para os parâmetros da URL** (como você pode ver em `orderId=${o.id}`). As operações necessárias para codificação dos parâmetros da URL também serão realizadas automaticamente.
- **Se vários parâmetros forem necessários**, eles serão separados por vírgulas: `@{/order/process(execId=${execId},execType='FAST')}`
- **Templates de variáveis também são permitidos nos caminhos de URL**: `@{/order/{orderId}/details(orderId=${orderId})}`
- **URLs relativas começando com `/`** (por exemplo: `/order/details`) serão automaticamente prefixadas com o nome do contexto da aplicação.
- **Se os cookies não estiverem habilitados ou isso ainda não for conhecido**, um sufixo `";jsessionid=..."` pode ser adicionado às URLs relativas para que a sessão seja preservada. Isso é chamado de "URL Rewriting" (reescrita de URL), e o Thymeleaf permite que você adicione seus próprios filtros de reescrita usando o mecanismo `response.encodeURL(...)` da API Servlet para cada URL.

- **O atributo `th:href` permite que tenhamos (opcionalmente) um atributo `href` estático funcional** em nosso template, para que nossos links de template permaneçam navegáveis por um navegador quando abertos diretamente para fins de prototipagem.
- **Assim como na sintaxe de mensagens (`#{...}`)**, as bases de URLs 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 a nossa página inicial

Agora que sabemos como criar URLs de links, que tal adicionar um pequeno menu na 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>
```

Server root relative URLs

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

4.5 Fragments

As expressões de fragmento são uma maneira fácil de representar fragmentos de markup e movê-los entre templates. Isso nos permite replicá-los, passá-los para outros templates como argumentos, e assim por diante. A utilização 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 eles podem ser usados em qualquer lugar, assim como qualquer outra variável:

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

Mais adiante neste tutorial, há uma seção inteira dedicada ao Layout de Template, incluindo uma explicação mais profunda sobre expressões de fragmento.

4.6 Literals

Literals de Texto

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

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

Literals de Números

Literais numéricos são exatamente 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>
```

Literals Lógicos (Booleanos)

Os **literais booleanos** são verdadeiro e falso. Por exemplo:

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

Neste exemplo, o `== false` está escrito fora das chaves, e, portanto, é o Thymeleaf que cuida disso. Se estivesse escrito dentro das chaves, seria responsabilidade dos motores OGNL/SpringEL:

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

Literals Nulos (Null)

O literal nulo também pode ser utilizado:

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

Literals Tokens

Literais numéricos, booleanos e nulos são, de fato, um caso particular de tokens literais. Esses tokens permitem um pouco de simplificação nas Expressões Padrão. Eles funcionam exatamente da mesma forma que os literais de texto ('...'), mas apenas permitem letras (A-Z e a-z), números (0-9), colchetes ([e]), pontos (.), hífens (-) e sublinhados (_). Portanto, não são permitidos espaços em branco, vírgulas, etc. A parte interessante? Os tokens não precisam de aspas ao seu redor. Portanto, podemos fazer isso:

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

em vez de:

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

4.7 Anexando textos

Textos, não importa se são literais ou o resultado da avaliação de expressões variáveis ou de mensagens, podem ser facilmente concatenados 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 que contêm valores de variáveis, sem a necessidade de concatenar literais com '...' + '...'. Essas substituições devem ser cercadas por barras verticais (|), como:

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

Que é equivalente a:

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

Substituições literais podem ser combinadas com outros tipos de expressões:

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

Somente expressões de variável/mensagem (${...}, *{...}, #{...}) são permitidas dentro das substituições literais |...|. Nenhum outro literal ('...'), tokens booleanos/numerais, expressões condicionais, etc. são permitidos.

4.9 Operadores Aritméticos

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

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

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

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

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

4.10 Comparadores e Igualdades

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, portanto, eles devem ser substituídos por < e >.

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

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

4.11 Expressões Condicionais

Expressões condicionais servem para avaliar apenas uma de duas expressões, dependendo do resultado da avaliação de uma condição (que é, ela mesma, outra expressão). Vamos dar uma olhada em um fragmento de exemplo (introduzindo outro modificador de atributo, th:class):

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

Todas as três partes de uma expressão condicional (condição, então e senão) são, elas mesmas, expressões, o que significa que podem ser variáveis (${...}, *{...}), mensagens (#{...}), URLs (@{...}) ou literais ('...'). As 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, caso em que um valor nulo é 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 uma parte "then". É equivalente ao operador Elvis presente em algumas linguagens, como Groovy, e permite especificar duas expressões: a primeira é usada se não for avaliada como nula, mas se for, a segunda é usada. Vamos ver isso em ação na 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 de *{age} for nulo. Isso é, portanto, equivalente a:

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

Assim como 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 No-Operation

O token No-Operation é representado pelo símbolo de sublinhado (_).
A ideia por trás desse token é especificar que o resultado desejado para uma expressão é não fazer nada, ou seja, agir 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 um texto de prototipagem, o que resulta em um código que é ao mesmo tempo 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 duplas chaves para expressões de variável (${...}) e seleção (*{...}) que nos permite aplicar conversão de dados por meio de um serviço de conversão configurado.
Basicamente, funciona assim:

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

Percebeu as duplas chaves ali?: ${{...}}. Isso instrui o Thymeleaf a passar o resultado da expressão user.lastAccessDate para o serviço de conversão e pede que ele realize uma operação de formatação (uma conversão para String) antes de escrever 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, ela será aplicada.

A implementação padrão de IStandardConversionService (a classe StandardConversionService) simplesmente executa .toString() em qualquer objeto convertido para String. Para mais informações sobre como registrar uma implementação de serviço de conversão personalizada, consulte a seção Mais sobre Configuração.

Os pacotes de integração 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, de modo que serviços de conversão e formatadores declarados na configuração do Spring estarão automaticamente disponíveis para as expressões ${{...}} e *{{...}}.

4.15 Pré-processamento

Além de todos esses recursos para o processamento de expressões, o Thymeleaf possui a funcionalidade de pré-processamento de expressões. O pré-processamento é uma execução das expressões realizada antes da normal, que permite a modificação da expressão que será eventualmente executada.
As expressões pré-processadas são exatamente como as normais, mas aparecem cercadas por um símbolo de sublinhado duplo (como **__${expression}__**).

Vamos imaginar que temos uma entrada em i18n Messages_fr.properties contendo uma expressão OGNL chamando um método estático específico de um idioma, como:

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

...e um equivalente em Messages_es.properties:

```
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 meio do pré-processamento) e, em seguida, 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 Definindo Valores de Atributos

Este capítulo explicará a forma como podemos definir (ou modificar) os valores dos atributos em nosso markup.

5.1 Definindo o valor de qualquer atributo

Digamos que nosso site publique uma newsletter e queremos que nossos usuários possam se inscrever nela, então criamos um template /WEBINF/templates/subscribe.html com um formulário:

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

Como acontece com o Thymeleaf, este template começa mais como um protótipo estático do que como um template para uma aplicação web. Primeiro, o atributo action em nosso formulário vincula estaticamente ao próprio arquivo do template, de modo que não há lugar para reescrita útil de URL. Em segundo lugar, o atributo value no botão de envio faz com que exiba um texto em inglês, mas gostaríamos que ele fosse internacionalizado.

Então, entra o atributo th:attr, e sua capacidade de mudar 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 recebe uma expressão que atribui um valor a um atributo. Tendo criado o controlador correspondente e os arquivos de mensagens, o resultado do processamento deste arquivo será:

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

Além dos novos valores de atributo, você também pode ver que o nome do contexto da aplicação foi automaticamente prefixado à base da URL em /gtvg/subscribe, conforme explicado no capítulo anterior. Mas e se quiséssemos definir mais de um atributo ao mesmo tempo? As regras do XML não permitem que você defina um atributo duas vezes em uma tag, então th:attr aceitará uma lista de atribuições separadas por vírgula, como:

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

Dado os arquivos de mensagens necessários, isso irá resultar em:

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

5.2 Definindo valor para atributos específicos

Neste ponto, você pode estar pensando que algo como:

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

… é um pedaço de markup bastante feio. Especificar uma atribuição dentro do valor de um atributo pode ser muito prático, mas não é a maneira mais elegante de criar templates se você tiver que fazer isso o tempo todo. O Thymeleaf concorda com você, e é por isso que th:attr é raramente utilizado em templates. Normalmente, você usará outros atributos th:* cuja tarefa é definir atributos específicos de tags (e não apenas qualquer atributo como th:attr).

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

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

Isso parece muito melhor! Vamos tentar fazer o mesmo para o atributo action na tag form:

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

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

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

Existem muitos atributos como esses, cada um deles direcionando para um atributo HTML5 específico:

th:abbr
th:accept
th:accept-chars
th:accesskey
th:action
th:align
th:alt
th:archive
th:audio
th:autocomplete
th:axis
th:background
th:bgcolor
th:border
th:cellpadding
th:cellspacing
th:challenge
th:charset
th:cite
th:class
th:classid
th:codebase
th:codetype
th:cols
th:colspan
th:compact
th:content
th:contenteditable
th:contextmenu
th:data
th:datetime
th:dir
th:draggable
th:dropzone
th:enctype
th:for
th:form
th:formaction
th:formenctype
th:formmethod
th:formtarget
th: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 Configurando mais de um valor ao mesmo tempo

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 a nossa página inicial do GTVG, isso nos permitirá substituir isso:

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

...ou isso, que é equivalente:

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

...por isso:

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

5.4 Appending and prepending

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

Por exemplo, você pode querer armazenar o nome de uma classe CSS a ser adicionada (não definida, apenas adicionada) a um de 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 template com a variável `cssStyle` definida como "warning", você obterá:

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

Existem também dois atributos de anexação 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 sobrescrever os existentes:

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

(Não se preocupe com o atributo `th:each`. É um atributo de iteração e falaremos sobre ele mais tarde.)

5.5 Atributos booleanos de valor fixo

O HTML tem o conceito de atributos booleanos, atributos que não têm valor e a presença de um significa que o valor é “verdadeiro”. No XHTML, esses atributos aceitam apenas 1 valor, que é o próprio atributo.

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:novalidate
th:nowrap	th:open	th:pubdate
th:readonly	th:required	th:reversed
th:scoped	th:seamless	th:selected

5.6 Definindo 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 processador específico `th:*` tenha sido definido para ele no Dialeto Padrão.

Assim, algo como:

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

Resultará em:

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

5.7 Suporte a nomes de atributos e elementos amigáveis ao HTML5

Também é possível usar uma sintaxe completamente diferente para aplicar processadores aos seus templates de uma maneira mais amigável ao 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 sintaxe `data-{prefix}-{name}` é a maneira padrão de escrever atributos personalizados em HTML5, sem exigir que os desenvolvedores usem nomes com namespace como `th:*`. O Thymeleaf torna essa sintaxe automaticamente disponível 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 (parte da especificação maior de Web Components do W3C). Isso pode ser

usado, por exemplo, para o elemento `th:block` (ou também `thblock`), que será explicado em uma seção posterior.

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

6 Iteration

Até agora, criamos uma página inicial, uma página de perfil de usuário e também uma página para permitir que os usuários se inscrevam em nossa newsletter... mas e quanto aos nossos produtos? Para isso, precisaremos de uma maneira de iterar sobre os itens em uma coleção para construir nossa página de produtos.

6.1 Noções básicas de iteração

Para exibir produtos em nossa página `/WEB-INF/templates/product/list.html`, usaremos uma tabela. Cada um de nossos produtos será exibido em uma linha (um elemento `<tr>`), e para 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-la, uma vez para cada produto. O Dialeto Padrão nos oferece um atributo exatamente para isso: `th:each`.

Usando th:each

Para nossa página de lista de produtos, precisaremos de um método no controlador que recupere a lista de produtos da camada de serviço e a adicione ao contexto do modelo:

```
public void process(
        final HttpServletRequest request, final HttpServletResponse response,
        final ServletContext servletContext, final ITemplateEngine templateEngine)
        throws Exception {

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

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

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

}
```

E então, usaremos `th:each` em nosso modelo para iterar sobre 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>

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

  </body>

</html>
```

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

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

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

Valores Iteráveis

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

- Qualquer objeto que implemente `java.util.Iterable`.
- Qualquer objeto que implemente `java.util.Enumeration`.
- Qualquer objeto que implemente `java.util.Iterator`, cujos valores serão usados conforme são retornados pelo iterador, sem a necessidade de armazenar todos os valores na memória.
- Qualquer objeto que implemente `java.util.Map`. Ao iterar sobre mapas, as variáveis iteradoras serão da classe `java.util.Map.Entry`.
- Qualquer array.
- Qualquer outro objeto será tratado como se fosse uma lista de valor único contendo o próprio objeto

6.2 Mantendo o status da iteração

Quando se utiliza `th:each`, o Thymeleaf oferece um mecanismo útil para acompanhar o status da sua iteração: a variável de status.
As variáveis de status são definidas dentro de um atributo `th:each` e contêm os seguintes dados:

- O índice da iteração atual, começando em 0. Esta é a propriedade `index`.
- O índice da iteração atual, começando em 1. Esta é a propriedade `count`.
- A quantidade total de elementos na variável iterada. Esta é a propriedade `size`.
- A variável iteradora para cada iteração. Esta é a propriedade `current`.
- Se a iteração atual é par ou ímpar. Estas são as propriedades booleanas `even/odd`.
- Se a iteração atual é a primeira. Esta é a propriedade booleano `first`.
- Se a iteração atual é a última. Esta é a propriedade booleano `last`.
  Vamos ver como poderíamos usar isso 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 de status (iterStat neste exemplo) é definida no atributo `th:each` escrevendo seu nome após a variável iter, separada por uma vírgula. Assim como a variável iter, a variável de status também está escopo ao fragmento de código definido pela tag que contém o atributo `th:each`.

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

```
<!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>
```

Note que nossa variável de status de iteração funcionou perfeitamente, aplicando a classe CSS 'odd' apenas às linhas ímpares.

Se você não definir explicitamente uma variável de status, o Thymeleaf sempre criará uma para você, adicionando o 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>
```

6.3 Otimizando através da recuperação preguiçosa de dados

Às vezes, pode ser interessante otimizar a recuperação de coleções de dados (por exemplo, de um banco de dados) para que essas coleções sejam obtidas apenas se realmente forem usadas.

Na verdade, isso pode ser aplicado a qualquer tipo de dado, mas devido ao tamanho que coleções em memória podem ter, a recuperação de coleções destinadas a serem iteradas é o caso mais comum nesse cenário.

Para dar suporte a isso, o Thymeleaf oferece um mecanismo para carregar variáveis de contexto de forma preguiçosa. Variáveis de contexto que implementam a interface **ILazyContextVariable** – provavelmente estendendo a implementação padrão **LazyContextVariable** – serão resolvidas no momento em que forem executadas. Por exemplo:

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

Essa variável pode ser usada sem o conhecimento de sua "preguiça", em um código como:

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

Mas, ao mesmo tempo, nunca será inicializada (seu método loadValue() nunca será chamado) se a condição for avaliada como falsa em um código como:

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

Gostou da tradução até aqui. Aproveite e conheça meus cursos:

Cursos de Java:

- Java Web com Spring Boot: https://pay.kiwify.com.br/mXnHYBK
- Java na Prática: https://pay.kiwify.com.br/gmU7HPU
- Java no Frontend com Swing: https://pay.kiwify.com.br/RXED8AQ
- Mentoria de Programação Backend Java: https://pay.kiwify.com.br/EbMHbry
- Inteligência Artificial Generativa: https://pay.kiwify.com.br/I4CthI2
- JavaScript na Prática: https://pay.kiwify.com.br/PFtSYRv

Quer se conectar comigo no **LinkedIn**? Acesse: https://www.linkedin.com/in/olivalpaulino/

Sucesso nos estudos.