Documentação Thymeleaf
Idioma: Português
Data: 01/01/2024
Versão do Thymeleaf: 3.1.2.RELEASE
Site Oficial do Projeto: https://www.thymeleaf.org/doc/tutorials/3.1/usingthymeleaf.html
Tradutor: Olival Paulino dos Santos Neto

1 Introdução Thymeleaf

1.1 O que é Thymeleaf

O Thymeleaf é um moderno motor de template Java do lado do servidor, adequado tanto para ambientes web quanto autônomos, capaz de processar HTML, XML, JavaScript, CSS e até mesmo texto simples.

O principal objetivo do Thymeleaf é fornecer uma maneira elegante e altamente sustentável de criar templates. Para alcançar isso, ele se baseia no conceito de Modelos Naturais para injetar sua lógica nos arquivos de template de uma maneira que não afeta o uso do template como um protótipo de design. Isso melhora a comunicação entre design e equipes de desenvolvimento.

O Thymeleaf também foi projetado desde o início com padrões da web em mente - especialmente o HTML5 - permitindo que você crie templates totalmente validados, se isso for uma necessidade para você.

1.2 Quais tipos de templates o Thymeleaf pode processar?

O Thymeleaf permite processar seis tipos de templates, cada um chamado de Modo de Template:

- HTML
- XML
- TEXTO
- JAVASCRIPT
- CSS
- RAW

Existem dois modos de template de marcação (HTML e XML), três modos de template textual (TEXTO, JAVASCRIPT e CSS) e um modo de template sem operação (RAW).

O modo de template **HTML** permitirá qualquer tipo de entrada HTML, incluindo HTML5, HTML 4 e XHTML. Nenhuma validação ou verificação de conformidade será realizada, e o código/estrutura do template será respeitado na maior medida possível na saída.

O modo de template **XML** permitirá entrada XML. Neste caso, espera-se que o código seja bem formado - sem tags não fechadas, sem atributos não citados, etc. - e o analisador lançará exceções se violações de conformidade forem encontradas. Note que nenhuma validação (contra um DTD ou XML Schema) será realizada.

O modo de template **TEXTO** permitirá o uso de uma sintaxe especial para templates de natureza não marcada. Exemplos de tais templates podem ser emails de texto ou documentação com modelo. Observe que templates HTML ou XML também podem ser processados como TEXTO, caso em que não serão analisados como marcação, e cada tag, DOCTYPE, comentário, etc., será tratado como simples texto.

O modo de template JAVASCRIPT permitirá o processamento de arquivos JavaScript em uma aplicação Thymeleaf. Isso significa poder usar dados do modelo dentro de arquivos JavaScript da mesma maneira que pode ser feito em arquivos HTML, mas com integrações específicas do JavaScript, como escape especializado ou scripting natural. O modo de template JAVASCRIPT é considerado um modo textual e, portanto, usa a mesma sintaxe especial do modo de template TEXTO.

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

O modo de template RAW simplesmente não processará templates. Destina-se a ser usado para inserir recursos não modificados (arquivos, respostas de URL, etc.) nos templates em processamento. Por exemplo, recursos externos e não controlados em formato HTML podem ser incluídos nos templates da aplicação, sabendo com segurança que qualquer código Thymeleaf que esses recursos possam incluir não será executado.

1.3 Dialetos: O Padrão Dialético

O Thymeleaf é um mecanismo de template extremamente extensível (na verdade, poderia ser chamado de um framework de mecanismo de template) que permite definir e personalizar a maneira como seus templates serão processados em detalhes refinados.

Um objeto que aplica alguma lógica a um artefato de marcação (uma tag, algum texto, um comentário ou um mero espaço reservado se os templates não forem de marcação) é chamado de processador, e um conjunto desses processadores - mais alguns artefatos extras, talvez - é normalmente o que compõe um dialeto. Por padrão, a biblioteca principal do Thymeleaf fornece um dialeto chamado Dialeto Padrão, que deve ser suficiente para a maioria dos usuários.

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

Este tutorial abrange o Dialeto Padrão. Cada atributo e recurso de sintaxe que você aprenderá nas páginas seguintes é definido por este dialeto, mesmo que isso não seja mencionado explicitamente.

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

Os pacotes de integração oficiais thymeleaf-spring3 e thymeleaf-spring4 ambos definem um dialeto chamado "SpringStandard Dialect", que é principalmente o mesmo que o Dialeto Padrão, mas com pequenas adaptações para aproveitar melhor alguns recursos no Spring Framework (por exemplo, usando a Spring Expression Language ou SpringEL em vez de OGNL). Portanto, se você é um usuário do Spring MVC, não está perdendo tempo, pois quase tudo o 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 arquivos de template HTML mesmo antes de serem processados, porque eles simplesmente ignorarão os atributos adicionais. Por exemplo, enquanto um JSP usando bibliotecas de tags poderia incluir um fragmento de código não diretamente exibível por um navegador, como:

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

O Dialeto Padrão do Thymeleaf nos permitiria alcançar a mesma funcionalidade com:

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

Isso não só 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 de ${user.name} durante o processamento do template.

Isso ajuda seu designer e desenvolvedor a trabalharem no mesmo arquivo de template e reduzir o esforço necessário para transformar um protótipo estático em um arquivo de template funcional. A capacidade de fazer isso é uma característica chamada de Modelos Naturais.

2 A Good Thymes Mercearia Virtual

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

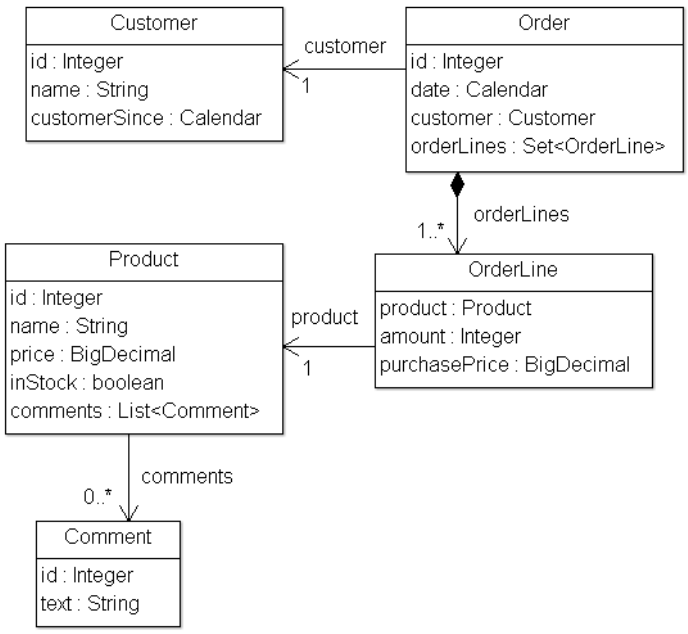
- javax.* based: https://github.com/thymeleaf/thymeleaf/tree/3.1-master/examples/core/thymeleaf-examples-gtvg-javax
- jakarta.* based: https://github.com/thymeleaf/thymeleaf/tree/3.1-master/examples/core/thymeleaf-examples-gtvg-jakarta

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 diversos cenários para destacar as muitas funcionalidades do Thymeleaf.

Para começar, precisamos de um conjunto simples de entidades de modelo para nossa aplicação: Produtos (Product) que são vendidos para Clientes (Customer) por meio da criação de Pedidos(Order). Também gerenciaremos Comentários (Comment) sobre esses Produtos:



Exemplo de modelo de aplicativo

Nossa aplicação também terá uma camada de serviço bem simples, composta por objetos Service 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 IGTVGController 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 interface IGTVGController:

```
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 interface IGTVGController, recuperando dados dos serviços e processando templates utilizando o objeto ItemplateEngine. No final, ficará assim:

Exemplo da página home do aplicativo

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