GeekBrains

Основы веб-разработки на Spring Framework

Thymeleaf

Шаблонизатор Thymeleaf. Интеграция с фреймворком Spring. Выражения. Операторы.

Оглавление

Thymeleaf

Thymeleaf — это современный серверный движок Java-шаблонов для веб- и автономных сред, способный обрабатывать HTML, XML, JavaScript, CSS и простой текст. Основная цель Thymeleaf — предоставить элегантный и удобный способ создания шаблонов. Он основывается на концепции Natural Templates, внедряя свою логику в файлы шаблона таким образом, чтобы он не влиял на использование прототипа дизайна. Использование Thymeleaf улучшает дизайн и способствует более тесному взаимодействию между группами backend- и frontend-разработчиков.

Thymeleaf был разработан с учетом стандартов Web, особенно HTML5, что позволяет создавать полностью проверенные шаблоны. Обособленно, без запуска приложения, шаблон можно проверить HTML-валидатором на соответствие стандарту HTML5. Шаблон может быть открыт в браузере как обычный html-файл (чем он и является), при этом он будет правильно отображен как валидная веб-страница. Thymeleaf обеспечивает интеграцию со Spring Framework.

Типы используемых шаблонов

Из коробки Thymeleaf позволяет обрабатывать шесть режимов шаблонов:

- HTML;
- XML;
- TEXT;
- JAVASCRIPT;
- CSS;
- RAW.

По умолчанию используется режим HTML.

Диалекты Thymeleaf

Чтобы добиться более простой и удобной интеграции, Thymeleaf предоставляет диалект Thymeleaf Spring, который специально реализует все необходимые функции для правильной работы со Spring.

Официальные пакеты интеграции **thymeleaf-spring3** и **thymeleaf-spring4** определяют диалект **SpringStandard Dialect**, который в основном совпадает со Standard Dialect, но содержит и небольшие изменения, позволяющие лучше использовать некоторые функции Spring Framework.

Помимо всех функций, уже присутствующих в стандартном диалекте и, следовательно, унаследованных, в диалекте **SpringStandard Dialect** представлены следующие особенности:

- в качестве языка выражений используется **Spring Expression Language** (SpEL), а не OGNL. Поэтому все выражения вида **${...}** и ***{...}** будут вычислены с помощью SpEL;
- есть доступ к любому бину в приложении с использованием SpEL, например **${@myBean.doSomething()}**;
- помимо новой реализации **th:object** имеются новые атрибуты для обработки формы: **th:field**, **th:errors** и **th:errorclass**.

Мы будем рассматривать диалект Thymeleaf Spring, так как он позволяет использовать Thymeleaf как полную замену JSP в приложениях Spring.

Thymeleaf — это механизм шаблонов Java для обработки и создания HTML, XML, JavaScript, CSS и текста. На этом уроке мы обсудим, как работать с Thymeleaf и Spring, рассмотрим базовые варианты использования на уровне представления приложения Spring MVC. Библиотека Thymeleaf расширяема, и ее естественная возможность шаблонирования гарантирует, что шаблоны могут быть прототипированы независимо — без использования сервера приложений. Это делает разработку очень быстрой по сравнению с другими популярными движками шаблонов, такими как JSP.

Шаблоны Thymeleaf выглядят как валидный статический HTML. В работающем приложении атрибуты пространства имен **th:** будут динамически вычислены и представлены как тело тега. Или будет выполнено дополнительное действие, например, как цикл ниже:

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
...
<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod: ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price, 1, 2)}">0.99</td>
    </tr>
  </tbody>
</table>
```

Интеграция со Spring

Thymeleaf предлагает набор возможностей для интеграций со Spring, чтобы использовать его как полнофункциональную замену JSP в приложениях Spring MVC. Это позволяет:

- создавать сопоставленные методы в объектах Spring MVC — **@Controller** для шаблонов, управляемых Thymeleaf;
- использовать Spring Expression Language (Spring EL) в шаблонах;
- создавать формы в шаблонах, которые полностью интегрированы с бинами обработки форм;
- реализовывать интернационализацию сообщений с помощью их файлов, управляемых Spring;
- маршрутизировать шаблоны, используя собственные механизмы разрешения ресурсов Spring.

Чтобы включить Thymeleaf в проект, требуется библиотека **thymeleaf-spring**. Для этого необходимо добавить следующие зависимости к файлу **Maven pom.xml**:

```
<dependency>
   <groupId>org.thymeleaf</groupId>
   <artifactId>thymeleaf-spring5</artifactId>
   <version>3.0.10.RELEASE</version>
```

```
</dependency>
```

Класс **SpringTemplateEngine** выполняет все этапы настройки. Вы можете настроить этот класс как компонент в конфигурационном файле Java:

```
  @Bean
  public SpringResourceTemplateResolver templateResolver() {
    SpringResourceTemplateResolver templateResolver = new
SpringResourceTemplateResolver ();
    viewResolver.setPrefix("/WEB-INF/templates/");
    viewResolver.setSuffix(".html");
    return templateResolver;
  }

  @Bean
  public TemplateEngine templateEngine() {
    TemplateEngine templateEngine = new SpringTemplateEngine();
    templateEngine.setTemplateResolver(templateResolver());
    return templateEngine;
  }

  @Bean
  public ThymeleafViewResolver thymeleafViewResolver() {
    ThymeleafViewResolver thymeleafViewResolver = new ThymeleafViewResolver ();
    thymeleafViewResolver.setTemplateEngine(templateEngine());
    thymeleafViewResolver.setEncoding("UTF-8");
    return thymeleafViewResolver;
  }
```

Свойства **prefix** и **suffix** бина **templateResolver** указывают расположение файлов шаблонов внутри (относительно) папки **webapp** и расширение файлов. Интерфейс **ViewResolver** в Spring MVC транслирует имена представлений, возвращаемые контроллером, в реальные объекты просмотра. **ThymeleafViewResolver** реализует интерфейс **ViewResolver** и используется, чтобы определять виды Thymeleaf для рендеринга с учетом имени представления.

Вариация Spring Boot

В случае Spring Boot проекта Thymeleaf добавляется соответствующим стартером:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Кроме библиотеки стартер добавляет небольшой «мешочек магии» в виде класса ThymeleafAutoConfiguration, что избавляет нас написания вручную бинов конфигурации, рассмотренных выше. А класс ThymeleafProperties определяет по умолчанию следующие опции, конфигурируемые в файле **application.properties**:

```
# THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.cache=true                    # кеширование

# Check template exists before rendering it.
spring.thymeleaf.check-template=true

# Check templates location exists.
spring.thymeleaf.check-template-location=true
spring.thymeleaf.content-type=text/html        # Content-Type value.

# Enable MVC Thymeleaf view resolution.
spring.thymeleaf.enabled=true
spring.thymeleaf.encoding=UTF-8                # Template encoding.

# список отключенных представлений
spring.thymeleaf.excluded-view-names=
spring.thymeleaf.mode=HTML5                    # режим шаблонов
spring.thymeleaf.prefix=classpath:/templates/ # путь к шаблонам, Prefix

# расширение файлов шаблонов, Suffix
spring.thymeleaf.suffix=.html
spring.thymeleaf.template-resolver-order=      # порядок поиска в цепочке

# закрытый список используемых шаблонов
spring.thymeleaf.view-names=
```

Отображение строк из файлов message.properties и интернационализация

Атрибут **th:text="# {key}"** может использоваться для отображения значений из файлов свойств. Для этого файл свойств должен быть указан при конфигурировании бина **messageSource**:

```
@Bean
@Description("Spring Message Resolver")
public ResourceBundleMessageSource messageSource() {
  ResourceBundleMessageSource messageSource = new
                                          ResourceBundleMessageSource();
  messageSource.setBasename("messages");
  return messageSource;
}
```

Отметим, что Spring Boot выполняет данное конфигурирование самостоятельно.

Код HTML-шаблона для отображения значения, связанного с ключом **welcome.message**:

```
<span th:text="#{welcome.message}" />
```

Строки сообщений могут быть параметризированы:

```
<span th:text="#{welcome.message(user.name)}" />
```

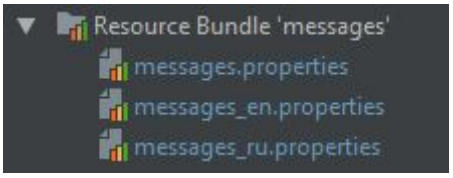
Файл **messages.properties**:

```
welcome.message=Welcome {0}!
```

По умолчанию приложение **Spring Boot** будет искать файлы сообщений, содержащие ключи, и значения интернационализации в папке **src/main/resources**.

Файл для локали по умолчанию будет иметь имя **messages.properties**, а файлы для каждой локали — **messages_XX.properties**, где **XX** — код локали. Если файла с запрошенным кодом не существует, будет использован файл локали по умолчанию.

Интернационализация достигается добавлением суффикса языкового кода к имени файла свойств:



Файл **messages_ru.properties**:

```
welcome.message=Добро пожаловать {0}!
```

Отображение атрибутов модели

Атрибут **th:text = "${attributename}"** используется для отображения значения атрибутов модели. Добавим атрибут **model** с именем **serverTime** в класс контроллера:

```
model.addAttribute("serverTime", dateFormat.format(new Date()));
```

Код HTML-шаблона для отображения значения атрибута **serverTime**:

```
Current time is <span th:text="${serverTime}" />
```

Отображение атрибутов коллекций

Если атрибут **model** представляет собой коллекцию объектов, атрибут **th:each** может использоваться для перебора этой коллекции. Определим класс модели **Item**:

```
// новость: шапка. текст, дата, источник
public class Item {
  private final String header;
  private final String text;
```

Теперь определим список новостей как атрибут модели в классе контроллера:

```
// ... логика наполнения списка опущена
@ModelAttribute("items")
public List<Item> populateItems() {
  return items;
}
```

Используем шаблон **Thymeleaf**, чтобы вывести список элементов и отобразить значения полей каждого пользователя:

```
<table>
  <tr th:each="item: ${items}">
    <td th:text="${item.header}"/>
    <td th:text="${item.text}"/>
  </tr>
</table>
```

Форматированный вывод

Для форматированного вывода пользовательских типов можно воспользоваться конструкцией **{{...}}**:

```
<td th:text="${{item.date}}"/>
```

И определить бин пользовательского форматтера в конфигурации приложения:

```
@SpringBootApplication
public class ThymeleafApplication {
  ...
  @Bean
  public DateFormatter dateFormatter() {
    return new DateFormatter();
  }
```

```
public class DateFormatter implements Formatter<Date> {
  @Autowired
  private MessageSource messageSource;

  public Date parse(final String text, final Locale locale)
                      throws ParseException {
    final SimpleDateFormat dateFormat = createDateFormat(locale);
    return dateFormat.parse(text);
  }

  public String print(final Date object, final Locale locale) {
    final SimpleDateFormat dateFormat = createDateFormat(locale);
    return dateFormat.format(object);
  }

  private SimpleDateFormat createDateFormat(final Locale locale) {
    final String format = this.messageSource.getMessage("date.format",
                                                        null, locale);
    final SimpleDateFormat dateFormat = new SimpleDateFormat(format);
```

```
    dateFormat.setLenient(false);
    return dateFormat;
  }
}
```

Обработка форм

Действие формы может быть указано с помощью атрибута **th:action**. Атрибут **th:object** связывает данную форму с указанным объектом модели. Отдельные поля отображаются с использованием атрибута **th:field="*{name}"**, где **name** — имя свойства объекта, для получения и заполнения которого будет использованы геттеры и сеттеры.

Следующий пример шаблона:

```
<form action="#" th:action="@{/some}" th:object="${item}" method="post">
  <input type="text" th:field="*{date}" />
  <input type="text" th:field="*{header}" />
  <input type="text" th:field="*{text}" />
</form>
```

… отображается в:

```
<form action="/some" method="post">
  <input type="text" id="date" name="date" value="10.10.2017" />
  <input type="text" id="header" name="header" value="" />
  <input type="text" id="text" name="text" value="" />
</form>
```

Заметим на примере свойства **date**, что **th:field** автоматически использует доступные форматтеры пользовательских типов.

Валидация полей формы

Функция **#fields.hasErrors()** может использоваться для проверки наличия ошибок. Функция **#fields.errors()** отображает ошибки определенного поля. Для обеих этих функций входным параметром будет имя поля — в его качестве могут выступать предопределенные константы «*» и «**all**».

Пример:

```
<input type="text" th:field="*{date}" />
<p th:if="${#fields.hasErrors('date')}" th:errors="*{date}">Incorrect date</p>
```

```
<ul>
    <li th:each="err : ${#fields.errors('all')}" th:text="${err}" />
</ul>
```

Упрощение стилизации CSS на основе ошибок

Применительно к тегу поля формы (**input**, **select**, **textarea**...) атрибут **th:errorclass** будет читать имя поля, которое должно быть проверено, из любого существующего имени или **th:** атрибутов поля в том же теге. А затем добавит указанный класс CSS в тег, если в этом поле есть связанные ошибки.

Пример:

```
<input type="text" th:field="*{date}" th:errorclass="error" />
```

Если в поле будет ошибка, **date** будет отображено, как предписывает следующий HTML-код:

```
<input type="text" id="date" name="date" value="2017-10-10" class="error" />
```

Условные выражения

Атрибут **th:if="${condition}"** используется для отображения раздела представления, если условие истинно.

Пример:

```
<div id="comments" th:if="${!items.empty}">
    <h3>Comments</h3>
    <ul th:each="item : ${items}">
        <li><span th:text="${item}">comment</span></li>
    </ul>
</div>
```

** ** и все его содержимое будет отображено только при непустом списке **items**.

Атрибут **th:unless="${condition}"** — это антипод **th:if**. Действие будет выполнено при ложном значении условия.

Пример:

```
<a href="comments.html" th:href="@{/comments}" th:unless="${items.empty}">view
comments</a>
```

Ссылка **<a href=...>** будет отображаться только до тех пор (**unless**), пока **items** не пуст.

Практическое задание

1. Добавить навигацию и пагинацию по страницам в таблице товаров.
2. Реализовать фильтр товаров по диапазону цен на основе двух полей (минимальная и максимальная цена) и кнопки («Фильтровать»).
3. * Добавить возможность редактировать существующие товары.

Дополнительные материалы

1. https://www.thymeleaf.org/documentation.html

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Официальная документация. Tutorial: Using Thymeleaf.
2. Официальная документация. Tutorial: Thymeleaf + Spring.