

Thymeleaf



教程: Thymeleaf + Spring

文件版本: 20180605 - 2018年6月5日

项目版本: 3.0.9.RELEASE

项目网站: <https://www.thymeleaf.org>

前言

本教程解释了Thymeleaf如何与Spring Framework集成，尤其是（但不仅仅是）Spring MVC。

请注意，Thymeleaf集成了Spring Framework的3.x和4.x版本，由两个名为 **thymeleaf-spring3** 和的独立库提供 **thymeleaf-spring4**。这些库打包在单独的 **.jar** 文件（ **thymeleaf-spring3-{version}.jar** 和 **thymeleaf-spring4-{version}.jar** ）中，需要添加到类路径中，以便在应用程序中使用Thymeleaf的Spring集成。

本教程中的代码示例和示例应用程序使用**Spring 4.x**及其相应的Thymeleaf集成，但本文的内容对Spring 3.x也有效。如果您的应用程序使用Spring 3.x，您只需在代码示例中替换 **org.thymeleaf.spring4** 包 **org.thymeleaf.spring3**。

1将Thymeleaf与Spring结合起来

Thymeleaf提供了一组Spring集成，允许您将其用作Spring MVC应用程序中JSP的全功能替代品。

这些集成将允许您：

- 使Spring MVC **@Controller** 对象中的映射方法转发到由Thymeleaf管理的模板，就像使用JSP一样。
- 在模板中使用**Spring Expression Language** (Spring EL) 代替OGNL。
- 在模板中创建与表单支持bean和结果绑定完全集成的表单，包括使用属性编辑器，转换服务和验证错误处理。
- 显示来自Spring管理的消息文件的国际化消息（通过常用 **MessageSource** 对象）。

请注意，为了完全理解本教程，您应该首先阅读“*使用Thymeleaf*”教程，该教程深入解释了标准方言。

2 SpringStandard方言

为了实现更容易和更好的集成, Thymeleaf提供了一种方言, 专门实现了与Spring一起正常工作所需的所有功能。

这种特定的方言基于Thymeleaf标准方言, 并在一个叫做的类中实现

`org.thymeleaf.spring4.dialect.SpringStandardDialect`, 实际上是从这个类扩展而来
`org.thymeleaf.standard.StandardDialect`。

除了标准方言中已经存在的所有功能 - 因此继承了 -, `SpringStandardDialect`还引入了以下特定功能:

- 使用Spring Expression Language (Spring EL) 作为变量表达式语言, 而不是OGNL。因此, 所有 `${...}` 和 `*{...}` 表达式都将由Spring的表达式语言引擎进行评估。变量表达式的优先被解释为springEL表达式
- 使用SpringEL的语法访问应用程序上下文中的任何bean: `#{@myBean.doSomething()}`
- 新属性表格处理: `th:field`, `th:errors` 并且 `th:errorclass`, 除了新的实施 `th:object`, 允许它被用于形式命令选择。增加了新标签
- 表达式对象和方法, `#themes.code(...)` 相当于 `spring:theme` JSP自定义标记。
- 表达式对象和方法, `#mvc.uri(...)` 相当于 `spring:mvcUrl(...)` JSP自定义函数 (仅在Spring 4.1+中)。
- 用于验证的新DTD, 包括这些新属性, 以及新的相应DOCTYPE转换规则。

请注意, 您不应将此方言直接用于普通的`TemplateEngine`对象作为其配置的一部分。相反, 您应该实例化一个新的模板引擎类, 它执行所有必需的配置步骤: `org.thymeleaf.spring4.SpringTemplateEngine`。

一个示例bean配置:

```
<bean id="templateResolver"
      class="org.thymeleaf.templateresolver.ServletContextTemplateResolver">
    <property name="prefix" value="/WEB-INF/templates/" />
    <property name="suffix" value=".html" />
    <property name="templateMode" value="HTML5" />
</bean>

<bean id="templateEngine"
      class="org.thymeleaf.spring4.SpringTemplateEngine">
    <property name="templateResolver" ref="templateResolver" />
</bean>
```

3个视图和视图解析器

3.1 Spring MVC中的视图和视图解析器

Spring MVC中有两个接口符合其模板系统的核心:

- `org.springframework.web.servlet.View`
- `org.springframework.web.servlet.ViewResolver`

在我们的应用程序中查看模型页面, 并允许我们通过将它们定义为bean来修改和预定义它们的行为。视图负责呈现实际的HTML接口, 通常通过执行一些模板引擎 (如JSP (或Thymeleaf)) 。

ViewResolvers是负责获取特定操作和区域设置的View对象的对象。通常, 控制器要求ViewResolvers转发到具有特定名称的视图 (由控制器方法返回的字符串), 然后应用程序中的所有视图解析器都在有序链中执行, 直到其中一个能够解析该视图, 其中case返回一个View对象, 并将控制权传递给它以便重新定义HTML。

请注意, 并非我们的应用程序中的所有页面都必须定义为视图, 而只是那些我们希望非标准或以特定方式配置的行为 (例如, 通过将一些特殊bean连接到它。如果要求ViewResolver一个没有相应bean的视图 - 这是常见的情况 - 一个新的View对象是临时创建并返回的。

Spring MVC应用程序中JSP + JSTL ViewResolver的典型配置如下所示:

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
  <property name="prefix" value="/WEB-INF/jsp/" />
  <property name="suffix" value=".jsp" />
  <property name="order" value="2" />
  <property name="viewNames" value="*jsp" />
</bean>
```

快速浏览一下它的属性就足以了解它的配置方式:

- **viewClass** 建立View实例的类。这是JSP解析器所必需的, 但在我们与Thymeleaf合作时根本不需要它。
- **prefix** 并 **suffix** 以与Thymeleaf的TemplateResolver对象中相同名称的属性类似的方式工作。
- **order** 建立在链中查询ViewResolver的顺序。
- **viewNames** 允许定义 (使用通配符) 将由此ViewResolver解析的视图名称。

3.2 Thymeleaf中的视图和视图解析器

Thymeleaf为上面提到的两个接口提供了实现:

- `org.thymeleaf.spring4.view.ThymeleafView`
- `org.thymeleaf.spring4.view.ThymeleafViewResolver`

由于执行控制器, 这两个类将负责处理Thymeleaf模板。

Thymeleaf View Resolver的配置与JSP的配置非常相似:

```
<bean class="org.thymeleaf.spring4.view.ThymeleafViewResolver">
  <property name="templateEngine" ref="templateEngine" />
  <property name="order" value="1" />
  <property name="viewNames" value="*.html,*.xhtml" />
</bean>
```

templateEngine 当然, 该参数是 **SpringTemplateEngine** 我们在前一章中定义的对象。其他两个 (**order** 和 **viewNames**) 都是可选的, 与我们之前看到的JSP ViewResolver具有相同的含义。

请注意, 我们不需要 **prefix** 或 **suffix** 参数, 因为这些已经在模板解析器中指定 (模板解析器又传递给模板引擎)。

如果我们想要定义一个 **View** bean并为它添加一些静态变量怎么办? 简单:

```
<bean name="main" class="org.thymeleaf.spring4.view.ThymeleafView">
  <property name="staticVariables">
    <map>
      <entry key="footer" value="Some company: &lt;b&gt;ACME&lt;/b&gt;" />
    </map>
  </property>
</bean>
```

4 模板分辨率

4.1 基于弹簧的模板分辨率

当与Spring一起使用时, Thymeleaf提供 `ITemplateResolver` 了 `IResourceResolver` 与Spring的资源解析机制完全集成的附加实现和相关联。这些是:

- `org.thymeleaf.spring4.templateresolver.SpringResourceTemplateResolver` 用于解析模板。
- `org.thymeleaf.spring4.resourceresolver.SpringResourceResourceResolver` , 主要供内部使用。

此模板解析程序将允许应用程序使用标准Spring资源解析语法解析模板。它可以配置为:

```
<bean id="templateResolver"
      class="org.thymeleaf.spring4.templateresolver.SpringResourceTemplateResolver">
  <property name="suffix" value=".html" />
  <property name="templateMode" value="HTML5" />
</bean>
```

这将允许您使用视图名称, 如:

```
@RequestMapping("/doit")
public String doIt() {
    ...
    return "classpath:resources/templates/doit";
}
```

请注意, 默认情况下永远不会使用这个基于Spring的资源解析器。除了Thymeleaf核心提供的其他模板解析器实现之外, 它还是一个可供应用程序配置的选项。


5春天百里香种子初学者经理

可以在[Spring Thyme Seed Starter Manager GitHub存储库](#)中找到本指南本章和后续章节中显示的示例的源代码。

5.1概念

在Thymeleaf，我们是百里香的忠实粉丝，每年春天我们都会为我们的种子起始工具准备好土壤和我们最喜欢的种子，将它们放在西班牙的阳光下，耐心地等待我们的新植物生长。

但是今年我们厌倦了在种子起始容器上粘贴标签以了解容器的每个单元格中的哪个种子，因此我们决定使用Spring MVC和Thymeleaf准备一个应用程序来帮助我们起始者进行编目：*Spring Thyme SeedStarter经理*。



Spring Thyme
SEEDSTARTER MANAGER

Seed Starter List

Date planted	Covered	Type	Features	Rows
03/20/2011	yes	Wood	Seed starter-specific substrate, PH Corrector used	<div><div>1</div><div>Thymus vulgaris</div><div>10</div></div> <div><div>2</div><div>Thymus pseudolaginosus</div><div>15</div></div> <div><div>3</div><div>Thymus x citriodorus</div><div>20</div></div>
03/25/2011	no	Plastic	Fertilizer used	<div><div>1</div><div>Thymus herba-barona</div><div>5</div></div> <div><div>2</div><div>Thymus serpyllum</div><div>10</div></div>

Add new Seed Starter

Seed Starter data

Date planted (MM/dd/yyyy)

04/05/2011

Covered

☐

Type

Plastic

Features

☐ Seed starter-specific substrate

☐ Fertilizer used

☐ PH Corrector used

Rows

Row	Variety	Seeds per cell	
1	Thymus vulgaris		<div><div>Add row</div><div>Remove row</div></div>

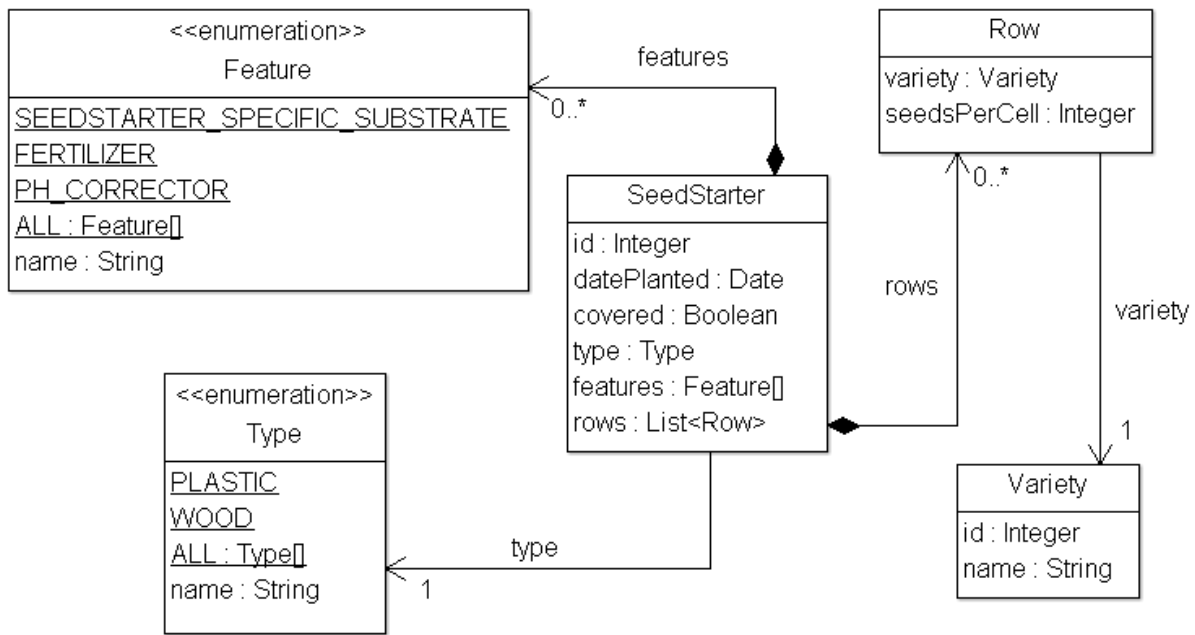
ADD SEED STARTER

STSM首页

与我们在使用Thymeleaf教程中开发的Good Thymes Virtual Grocery应用程序类似，STSM将允许我们举例说明Thymeleaf作为Spring MVC模板引擎集成的最重要方面。

5.2业务层

我们的应用程序需要一个非常简单的业务层。首先, 让我们看看我们的模型实体:



STSM模型

一些非常简单的服务类将提供所需的业务方法。喜欢:

```

@Service
public class SeedStarterService {

    @Autowired
    private SeedStarterRepository seedstarterRepository;

    public List<SeedStarter> findAll() {
        return this.seedstarterRepository.findAll();
    }

    public void add(final SeedStarter seedStarter) {
        this.seedstarterRepository.add(seedStarter);
    }

}
  
```

和:

```

@Service
public class VarietyService {

    @Autowired
    private VarietyRepository varietyRepository;

    public List<Variety> findAll() {
        return this.varietyRepository.findAll();
    }

    public Variety findById(final Integer id) {
        return this.varietyRepository.findById(id);
    }

}
  
```

5.3 Spring MVC配置

接下来, 我们需要为应用程序设置Spring MVC配置, 它不仅包括标准的Spring MVC工件, 如资源处理或注释扫描, 还包括模板引擎和View Resolver实例的创建。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <!-- ***** -->
    <!-- RESOURCE FOLDERS CONFIGURATION -->
    <!-- Dispatcher configuration for serving static resources -->
    <!-- ***** -->
    <mvc:resources location="/images/" mapping="/images/**" />
    <mvc:resources location="/css/" mapping="/css/**" />

    <!-- ***** -->
    <!-- SPRING ANNOTATION PROCESSING -->
    <!-- ***** -->
    <mvc:annotation-driven conversion-service="conversionService" />
    <context:component-scan base-package="thymeleafexamples.stsm" />

    <!-- ***** -->
    <!-- MESSAGE EXTERNALIZATION/INTERNATIONALIZATION -->
    <!-- Standard Spring MessageSource implementation -->
    <!-- ***** -->
    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="Messages" />
    </bean>

    <!-- ***** -->
    <!-- CONVERSION SERVICE -->
    <!-- Standard Spring formatting-enabled implementation -->
    <!-- ***** -->
    <bean id="conversionService"
          class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <property name="formatters">
            <set>
                <bean class="thymeleafexamples.stsm.web.conversion.VarietyFormatter" />
                <bean class="thymeleafexamples.stsm.web.conversion.DateFormatter" />
            </set>
        </property>
    </bean>

    <!-- ***** -->
    <!-- THYMELEAF-SPECIFIC ARTIFACTS -->
    <!-- TemplateResolver <- TemplateEngine <- ViewResolver -->
    <!-- ***** -->

    <bean id="templateResolver"
          class="org.thymeleaf.templateresolver.ServletContextTemplateResolver">
        <property name="prefix" value="/WEB-INF/templates/" />
        <property name="suffix" value=".html" />
        <property name="templateMode" value="HTML5" />
    </bean>
```

```
<bean id="templateEngine"
      class="org.thymeleaf.spring4.SpringTemplateEngine">
  <property name="templateResolver" ref="templateResolver" />
</bean>

<bean class="org.thymeleaf.spring4.view.ThymeleafViewResolver">
  <property name="templateEngine" ref="templateEngine" />
</bean>

</beans>
```

重要提示: 请注意, 我们已选择HTML5作为模板模式。

5.4 财务主任

当然, 我们还需要一个控制器用于我们的应用程序。由于STSM只包含一个包含种子启动器列表的网页和一个用于添加新启动器的表单, 因此我们将只为所有服务器交互编写一个控制器类:

```
@Controller
public class SeedStarterMngController {

    @Autowired
    private VarietyService varietyService;

    @Autowired
    private SeedStarterService seedStarterService;

    ...

}
```

现在让我们看看我们可以添加到这个控制器类。

模型属性

首先, 我们将在页面中添加一些我们需要的模型属性:

```
@ModelAttribute("allTypes")
public List<Type> populateTypes() {
    return Arrays.asList(Type.ALL);
}

@ModelAttribute("allFeatures")
public List<Feature> populateFeatures() {
    return Arrays.asList(Feature.ALL);
}

@ModelAttribute("allVarieties")
public List<Variety> populateVarieties() {
    return this.varietyService.findAll();
}

@ModelAttribute("allSeedStarters")
public List<SeedStarter> populateSeedStarters() {
    return this.seedStarterService.findAll();
}
```

映射方法

现在是控制器最重要的部分，映射方法：一个用于显示表单页面，另一个用于处理添加新的Seed Starter对象。

```
@RequestMapping("/{"/seedstartermng"})
public String showSeedstarters(final SeedStarter seedStarter) {
    seedStarter.setDatePlanted(Calendar.getInstance().getTime());
    return "seedstartermng";
}

@RequestMapping(value="/seedstartermng", params={"save"})
public String saveSeedstarter(
    final SeedStarter seedStarter, final BindingResult bindingResult, final ModelMap model) {
    if (bindingResult.hasErrors()) {
        return "seedstartermng";
    }
    this.seedStarterService.add(seedStarter);
    model.clear();
    return "redirect:/seedstartermng";
}
```

5.5 配置转换服务

为了便于格式化视图层中的对象 **Date** 和 **Variety** 对象，我们 **ConversionService** 在应用程序上下文中注册了一个Spring 实现。再看一遍：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    ...
    <mvc:annotation-driven conversion-service="conversionService" />
    ...

    <!-- ***** -->
    <!-- CONVERSION SERVICE -->
    <!-- Standard Spring formatting-enabled implementation -->
    <!-- ***** -->
    <bean id="conversionService"
        class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <property name="formatters">
            <set>
                <bean class="thymeleafexamples.stsm.web.conversion.VarietyFormatter" />
                <bean class="thymeleafexamples.stsm.web.conversion.DateFormatter" />
            </set>
        </property>
    </bean>

    ...
</beans>
```

该转换服务允许我们注册两个Spring *格式化程序*，即 **org.springframework.format.Formatter** 接口的实现。有关Spring转换基础结构如何工作的更多信息，请参阅spring.io上的文档。

让我们看一下，**DateFormatter** 根据 **date.format** 我们的消息键上的格式字符串格式化日期 **Messages.properties**：

```
public class DateFormatter implements Formatter<Date> {

    @Autowired
    private MessageSource messageSource;

    public DateFormatter() {
        super();
    }
}
```

```

    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;
    }
}

```

在 **VarietyFormatter** 我们之间自动转换 **Variety** 的实体, 我们希望在我们的形式使用它们 (基本上, 通过他们的方式 **id** 字段值) :

```

public class VarietyFormatter implements Formatter<Variety> {

    @Autowired
    private VarietyService varietyService;

    public VarietyFormatter() {
        super();
    }

    public Variety parse(final String text, final Locale locale) throws ParseException {
        final Integer varietyId = Integer.valueOf(text);
        return this.varietyService.findById(varietyId);
    }

    public String print(final Variety object, final Locale locale) {
        return (object != null ? object.getId().toString() : "");
    }
}

```

我们将详细了解这些格式化程序如何影响我们以后显示数据的方式。

6列出种子入门数据

我们的 `/WEB-INF/templates/seedstartermng.html` 页面将显示的第一件事是当前存储的种子启动器的列表。为此，我们需要一些外化消息，还需要对模型属性进行一些表达式评估。喜欢这个：

```
<div class="seedstarterlist" th:unless="${#lists.isEmpty(allSeedStarters)}">

    <h2 th:text="${title.list}">List of Seed Starters</h2>

    <table>
        <thead>
            <tr>
                <th th:text="${seedstarter.datePlanted}">Date Planted</th>
                <th th:text="${seedstarter.covered}">Covered</th>
                <th th:text="${seedstarter.type}">Type</th>
                <th th:text="${seedstarter.features}">Features</th>
                <th th:text="${seedstarter.rows}">Rows</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="sb : ${allSeedStarters}">
                <td th:text="${sb.datePlanted}">13/01/2011</td>
                <td th:text="${sb.covered}? #{bool.true} : #{bool.false}">yes</td>
                <td th:text="${${'seedstarter.type.' + sb.type}}">Wireframe</td>
                <td th:text="${#strings.arrayJoin(
                    #messages.arrayMsg(
                        #strings.arrayPrepend(sb.features, 'seedstarter.feature.')),
                    ', ')}">Electric Heating, Turf</td>

                <td>
                    <table>
                        <tbody>
                            <tr th:each="row,rowStat : ${sb.rows}">
                                <td th:text="${rowStat.count}">1</td>
                                <td th:text="${row.variety.name}">Thymus Thymi</td>
                                <td th:text="${row.seedsPerCell}">12</td>
                            </tr>
                        </tbody>
                    </table>
                </td>
            </tr>
        </tbody>
    </table>
</div>
```

有很多可以看到这里。让我们分别看看每个片段。

首先，只有有任何种子开始时才会显示此部分。我们用th：除非属性和 `#lists.isEmpty(...)` 函数来实现。

```
<div class="seedstarterlist" th:unless="${#lists.isEmpty(allSeedStarters)}">
```

请注意，所有实用程序对象 `#lists` 都可以在Spring EL表达式中使用，就像它们在标准方言中的OGNL表达式中一样。

接下来要看的是很多国际化（外化）文本，例如：

```
<h2 th:text="${title.list}">List of Seed Starters</h2>

<table>
    <thead>
        <tr>
            <th th:text="${seedstarter.datePlanted}">Date Planted</th>
            <th th:text="${seedstarter.covered}">Covered</th>
            <th th:text="${seedstarter.type}">Type</th>
```

```
<th th:text="#{seedstarter.features}">Features</th>
<th th:text="#{seedstarter.rows}">Rows</th>
...
```

这是一个Spring MVC应用程序，我们已经 **MessageSource** 在spring XML配置中定义了一个bean（**MessageSource** 对象是在Spring MVC中管理外化文本的标准方法）：

```
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basename" value="Messages" />
</bean>
```

...那基本名称属性指示，我们将有类似的文件 **Messages_es.properties** 或 **Messages_en.properties** 在我们的类路径。我们来看看西班牙语版本：

```
title.list=Lista de semilleros

date.format=dd/MM/yyyy
bool.true=sí
bool.false=no

seedstarter.datePlanted=Fecha de plantación
seedstarter.covered=Cubierto
seedstarter.type=Tipo
seedstarter.features=Características
seedstarter.rows=Filas

seedstarter.type.WOOD=Madera
seedstarter.type.PLASTIC=Plástico

seedstarter.feature.SEEDSTARTER_SPECIFIC_SUBSTRATE=Sustrato específico para semilleros
seedstarter.feature.FERTILIZER=Fertilizante
seedstarter.feature.PH_CORRECTOR=Corrector de PH
```

在表格列表的第一列中，我们将显示种子启动器的准备日期。但**我们将按照我们在我们中定义的方式显示格式 `DateFormatter`**。为此，我们将使用双括号语法，它将自动应用Spring转换服务。

```
<td th:text="${{sb.datePlanted}}">13/01/2011</td>
```

接下来通过将带有条件表达式的boolean covered bean属性的值转换为国际化的“yes”或“no”来显示是否覆盖了种子入门容器：

```
<td th:text="${sb.covered}? #{bool.true} : #{bool.false}">yes</td>
```

现在我们必须显示种子入门容器的类型。Type是一个Java枚举有两个值（**WOOD** 和 **PLASTIC**），这就是为什么我们定义在我们的两个属性 **Messages** 文件调用 **seedstarter.type.WOOD** 和 **seedstarter.type.PLASTIC**。

但是为了获得类型的国际化名称，我们需要 **seedstarter.type**。通过表达式为枚举值添加前缀，然后我们将使用该结果作为消息键：

```
<td th:text="#{'seedstarter.type.' + sb.type}">Wireframe</td>
```

此列表中最困难的部分是**功能列**。在其中我们希望显示容器的所有功能 - 以 **Feature** 枚举数组的形式出现 - 以逗号分隔。像“**电加热, 草坪**”。

请注意，这尤其困难，因为这些枚举值也需要外部化，就像我们对类型一样。那么流程是：

1. 将相应的前缀添加到 **features** 数组的所有元素。
2. 获取与步骤1中的所有密钥对应的外部化消息。
3. 使用逗号作为分隔符加入在步骤2中获取的所有消息。

为实现此目的，我们创建以下代码：

```
<td th:text="${#strings.arrayJoin(
    #messages.arrayMsg(
        #strings.arrayPrepend(sb.features, 'seedstarter.feature.')),
    ', ')}">Electric Heating, Turf</td>
```

事实上, 我们列表的最后一栏非常简单。即使它有一个嵌套表来显示容器中每行的内容:

```
<td>
  <table>
    <tbody>
      <tr th:each="row,rowStat : ${sb.rows}">
        <td th:text="${rowStat.count}">1</td>
        <td th:text="${row.variety.name}">Thymus Thymi</td>
        <td th:text="${row.seedsPerCell}">12</td>
      </tr>
    </tbody>
  </table>
</td>
```


7 创建表单

7.1 处理命令对象

Command对象是Spring MVC为表单支持bean提供的名称，这是对表单的字段进行建模并提供getter和setter方法的对象，框架将使用这些方法来建立和获取用户在浏览器中输入的值。

Thymeleaf要求您使用标记中的 **th:object** 属性指定命令对象 **<form>**：

```
<form action="#" th:action="@{/seedstartermng}" th:object="${seedStarter}" method="post">
    ...
</form>
```

这与其他用途一致，**th:object**，但实际上这个特定的场景增加了一些限制，以便正确地与Spring MVC的基础架构集成：

- **th:object** 表单标记中的属性值必须是变量表达式（**\${...}**），仅指定模型属性的名称，而不包含属性导航。这意味着表达式 **\${seedStarter}** 是有效的，但 **\${seedStarter.data}** 不是。
- 进入 **<form>** 标记后，不能 **th:object** 指定其他属性。这与HTML表单无法嵌套这一事实是一致的。

7.2 输入

现在让我们看看如何在表单中添加输入：

```
<input type="text" th:field="*{datePlanted}" />
```

正如您所看到的，我们在这里引入了一个新属性：**th:field**。这是Spring MVC集成的一个非常重要的特性，因为它完成了将输入与表单支持bean中的属性绑定的所有繁重工作。您可以将其视为a中path属性的等效项 来自Spring MVC的JSP标记库的标记。

的 **th:field** 属性取决于它是否连接到一个具有不同的行为 **<input>**，**<select>** 或 **<textarea>** 标签（也取决于特定类型的 **<input>** 标签）。在这种情况下（**input[type=text]**），上面的代码行类似于：

```
<input type="text" id="datePlanted" name="datePlanted" th:value="*{datePlanted}" />
```

...但事实上它只是稍微多一些，因为它 **th:field** 也将应用已注册的Spring转换服务，包括 **DateFormatter** 我们之前看到的（即使字段表达式不是双括号）。多亏了这一点，日期将正确显示格式。

th:field 属性的值必须是选择表达式（***{...}**），这是有意义的，因为它们将在表单支持bean上进行评估，而不是在上下文变量（或Spring MVC术语中的模型属性）上进行评估。

与其中的相反 **th:object**，这些表达式可以包括属性导航（事实上，这里允许任何允许 **<form:input>** JSP标记的path属性的表达式）。

请注意，**th:field** 也了解新类型的 **<input>** 通过像HTML5元素引入 **<input type="datetime" ... />**，**<input type="color" ... />** 等等，Spring MVC的有效增加完整的HTML5的支持。

7.3 复选框字段

th:field 还允许我们定义复选框输入。我们从HTML页面看一个例子：

```
<div>
    <label th:for="${#ids.next('covered')}" th:text="#{seedstarter.covered}">Covered</label>
```

```
<input type="checkbox" th:field="*{covered}" />
</div>
```

注意除了复选框本身之外还有一些很好的东西，比如外化标签以及使用该 `#ids.next('covered')` 函数来获取将应用于复选框输入的id属性的值。

为什么我们需要为此字段动态生成id属性？因为复选框可能是多值的，因此它们的id值将始终以序列号为后缀（通过内部使用 `#ids.seq(...)` 函数），以确保同一属性的每个复选框输入具有不同的id值。

如果我们查看这样一个多值复选框字段，我们可以更容易地看到这一点：

```
<ul>
  <li th:each="feat : ${allFeatures}">
    <input type="checkbox" th:field="*{features}" th:value="${feat}" />
    <label th:for="${#ids.prev('features')}"
      th:text="#{'seedstarter.feature.' + feat}">Heating</label>
  </li>
</ul>
```

请注意，这次我们添加了一个 `th:value` 属性，因为features字段不是像覆盖的那样的布尔值，而是一个值数组。

让我们看看这段代码生成的HTML输出：

```
<ul>
  <li>
    <input id="features1" name="features" type="checkbox" value="SEEDSTARTER_SPECIFIC_SUBSTRATE" />
    <input name="_features" type="hidden" value="on" />
    <label for="features1">Seed starter-specific substrate</label>
  </li>
  <li>
    <input id="features2" name="features" type="checkbox" value="FERTILIZER" />
    <input name="_features" type="hidden" value="on" />
    <label for="features2">Fertilizer used</label>
  </li>
  <li>
    <input id="features3" name="features" type="checkbox" value="PH_CORRECTOR" />
    <input name="_features" type="hidden" value="on" />
    <label for="features3">PH Corrector used</label>
  </li>
</ul>
```

我们可以在这里看到如何将序列后缀添加到每个输入的id属性，以及该 `#ids.prev(...)` 函数如何允许我们检索为特定输入id生成的最后一个序列值。

Don't worry about those hidden inputs with `name="_features"` : they are automatically added in order to avoid problems with browsers not sending unchecked checkbox values to the server upon form submission.

Also note that if our features property contained some selected values in our form-backing bean, `th:field` would have taken care of that and would have added a `checked="checked"` attribute to the corresponding input tags.

7.4 Radio Button fields

Radio button fields are specified in a similar way to non-boolean (multi-valued) checkboxes —except that they are not multivalued, of course:

```
<ul>
  <li th:each="ty : ${allTypes}">
    <input type="radio" th:field="*{type}" th:value="${ty}" />
    <label th:for="${#ids.prev('type')}" th:text="#{'seedstarter.type.' + ty}">Wireframe</label>
  </li>
</ul>
```

```

</li>
</ul>

```

7.5 Dropdown/List selectors

Select fields have two parts: the `<select>` tag and its nested `<option>` tags. When creating this kind of field, only the `<select>` tag has to include a `th:field` attribute, but the `th:value` attributes in the nested `<option>` tags will be very important because they will provide the means of knowing which is the currently selected option (in a similar way to non-boolean checkboxes and radio buttons).

Let's re-build the type field as a dropdown select:

```

<select th:field="*{type}">
  <option th:each="type : ${allTypes}"
    th:value="{type}"
    th:text="#{'seedstarter.type.' + type}">Wireframe</option>
</select>

```

At this point, understanding this piece of code is quite easy. Just notice how attribute precedence allows us to set the `th:each` attribute in the `<option>` tag itself.

7.6 Dynamic fields

由于Spring MVC中的高级表单字段绑定功能, 我们可以使用复杂的Spring EL表达式将动态表单字段绑定到表单支持bean。这将允许我们在 **SeedStarter** bean中创建新的Row对象, 并在用户请求时将这些行的字段添加到表单中。

为了做到这一点, 我们需要在控制器中使用几个新的映射方法, 这将 **SeedStarter** 根据特定请求参数的存在添加或删除一行:

```

@RequestMapping(value="/seedstartermng", params={"addRow"})
public String addRow(final SeedStarter seedStarter, final BindingResult bindingResult) {
    seedStarter.getRows().add(new Row());
    return "seedstartermng";
}

@RequestMapping(value="/seedstartermng", params={"removeRow"})
public String removeRow(
    final SeedStarter seedStarter, final BindingResult bindingResult,
    final HttpServletRequest req) {
    final Integer rowId = Integer.valueOf(req.getParameter("removeRow"));
    seedStarter.getRows().remove(rowId.intValue());
    return "seedstartermng";
}

```

现在我们可以添加动态表:

```

<table>
  <thead>
    <tr>
      <th th:text="#{seedstarter.rows.head.rownum}">Row</th>
      <th th:text="#{seedstarter.rows.head.variety}">Variety</th>
      <th th:text="#{seedstarter.rows.head.seedsPerCell}">Seeds per cell</th>
    </tr>
    <tr>
      <td colspan="3">
        <button type="submit" name="addRow" th:text="#{seedstarter.row.add}">Add row</button>
      </td>
    </tr>
  </thead>
  <tbody>
    <tr th:each="row,rowStat : *{rows}">
      <td th:text="${rowStat.count}">1</td>
    </tr>
  </tbody>
</table>

```

```

<td>
  <select th:field="*{rows[__${rowStat.index}__].variety}">
    <option th:each="var : ${allVarieties}"
      th:value="${var.id}"
      th:text="${var.name}">Thymus Thymi</option>
  </select>
</td>
<td>
  <input type="text" th:field="*{rows[__${rowStat.index}__].seedsPerCell}" />
</td>
<td>
  <button type="submit" name="removeRow"
    th:value="${rowStat.index}" th:text="#{seedstarter.row.remove}">Remove row</button>
</td>
</tr>
</tbody>
</table>

```

在这里可以看到很多东西,但现在我们不应该理解太多.....除了一 **strange** 件事:

```

<select th:field="*{rows[__${rowStat.index}__].variety}">

  ...

</select>

```

如果您从“使用Thymeleaf”教程中回忆一下,该 `__${...}__` 语法是一个预处理表达式,它是在实际评估整个表达式之前计算的内部表达式。但为什么这种方式指定行索引?这不足以:

```

<select th:field="*{rows[rowStat.index].variety}">

  ...

</select>

```

.....好吧,实际上,没有。问题在于春EL不计算数组下标括号内的变量,所以执行上述表达式时我们会得到一个错误告诉我们 `rows[rowStat.index]` (而不是 `rows[0]`, `rows[1]` 等) 不在行收集的有效位置。这就是为什么需要预处理的原因。

在按“添加行”几次后,让我们看看生成的HTML片段:

```

<tbody>
<tr>
  <td>1</td>
  <td>
    <select id="rows0.variety" name="rows[0].variety">
      <option selected="selected" value="1">Thymus vulgaris</option>
      <option value="2">Thymus x citriodorus</option>
      <option value="3">Thymus herba-barona</option>
      <option value="4">Thymus pseudolaginosus</option>
      <option value="5">Thymus serpyllum</option>
    </select>
  </td>
  <td>
    <input id="rows0.seedsPerCell" name="rows[0].seedsPerCell" type="text" value="" />
  </td>
  <td>
    <button name="removeRow" type="submit" value="0">Remove row</button>
  </td>
</tr>
<tr>
  <td>2</td>
  <td>
    <select id="rows1.variety" name="rows[1].variety">
      <option selected="selected" value="1">Thymus vulgaris</option>
      <option value="2">Thymus x citriodorus</option>

```

```
<option value="3">Thymus herba-barona</option>
<option value="4">Thymus pseudolaginosus</option>
<option value="5">Thymus serpyllum</option>
</select>
</td>
<td>
  <input id="rows1.seedsPerCell" name="rows[1].seedsPerCell" type="text" value="" />
</td>
<td>
  <button name="removeRow" type="submit" value="1">Remove row</button>
</td>
</tr>
</tbody>
```

8 验证和错误消息

我们的大多数表单都需要显示验证消息，以便告知用户他/她所犯的错误。

Thymeleaf为此提供了一些工具：**#fields** 对象，属性 **th:errors** 和 **th:errorclass** 属性中的几个函数。

8.1 现场错误

让我们看看如果它有一个错误我们如何设置一个特定的CSS类到字段：

```
<input type="text" th:field="*{datePlanted}"
      th:class="${#fields.hasErrors('datePlanted')}? fieldError" />
```

如您所见，该 **#fields.hasErrors(...)** 函数接收字段表达式作为参数（**datePlanted**），并返回一个布尔值，告知该字段是否存在任何验证错误。

我们还可以获取该字段的所有错误并迭代它们：

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

我们也可以使用 **th:errors** 一个专门的属性来代替迭代，这个属性构建一个列表，其中包含指定选择器的所有错误，由 **
** 以下内容分隔：

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

简化基于错误的CSS样式: **th:errorclass**

The example we saw above, *setting a CSS class to a form input if that field has errors*, is so common that Thymeleaf offers a specific attribute for doing exactly that: **th:errorclass**.

Applied to a form field tag (input, select, textarea...), it will read the name of the field to be examined from any existing **name** or **th:field** attributes in the same tag, and then append the specified CSS class to the tag if such field has any associated errors:

```
<input type="text" th:field="*{datePlanted}" class="small" th:errorclass="fieldError" />
```

If **datePlanted** has errors, this will render as:

```
<input type="text" id="datePlanted" name="datePlanted" value="2013-01-01" class="small fieldError" />
```

8.2 All errors

And what if we want to show all the errors in the form? We just need to query the **#fields.hasErrors(...)** and **#fields.errors(...)** methods with the **'*'** or **'all'** constants (which are equivalent):

```
<ul th:if="${#fields.hasErrors('*')}}">
  <li th:each="err : ${#fields.errors('*')}}" th:text="${err}">Input is incorrect</li>
```

```
</ul>
```

As in the examples above, we could obtain all the errors and iterate them...

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

...以及构建 `
` 分离列表:

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

最后。注 `#fields.hasErrors('*')` 等同于 `#fields.hasAnyErrors()` 和 `#fields.errors('*')` 等价于 `#fields.allErrors()`。使用您喜欢的任何语法:

```
<div th:if="${#fields.hasAnyErrors()}">
  <p th:each="err : ${#fields.allErrors()}" th:text="${err}">...</p>
</div>
```

8.3 全局错误

Spring 表单中存在第三种类型的错误: 全局错误。这些错误与表单中的任何特定字段无关, 但仍然存在。

Thymeleaf 提供 `global` 访问这些错误的常量:

```
<ul th:if="${#fields.hasErrors('global')}}">
  <li th:each="err : ${#fields.errors('global')}}" th:text="${err}">Input is incorrect</li>
</ul>

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

.....以及等效 `#fields.hasGlobalErrors()` 和 `#fields.globalErrors()` 便利的方法:

```
<div th:if="${#fields.hasGlobalErrors()}">
  <p th:each="err : ${#fields.globalErrors()}" th:text="${err}">...</p>
</div>
```

8.4 在表单外显示错误

表单验证错误也可以通过使用 `variable` (`${...}`) 而不是 `selection` (`*{...}`) 表达式并在表单支持 bean 的名称前面显示在表单外部:

```
<div th:errors="${myForm}">...</div>
<div th:errors="${myForm.date}">...</div>
<div th:errors="${myForm.*}">...</div>

<div th:if="${#fields.hasErrors('${myForm}')}}">...</div>
<div th:if="${#fields.hasErrors('${myForm.date}')}}">...</div>
<div th:if="${#fields.hasErrors('${myForm.*}')}}">...</div>

<form th:object="${myForm}">
  ...
</form>
```

8.5 丰富的错误对象

Thymeleaf提供了使用 (String) , (String) 和 (boolean) 属性以bean (而不仅仅是字符串) 的形式获取表单错误信息的可能性。 **fieldName message global**

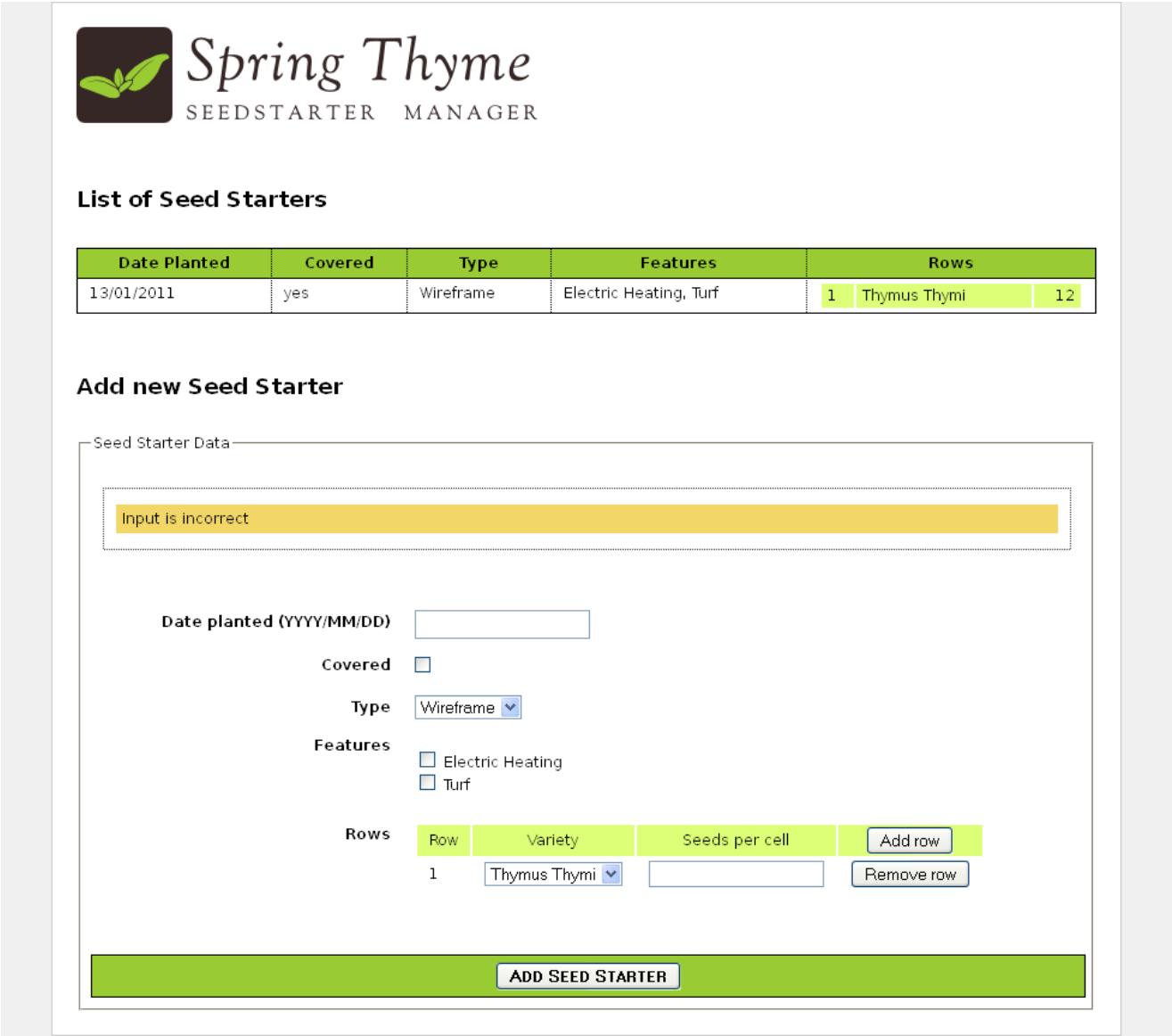
这些错误可以通过 **#fields.detailedErrors()** 实用方法获得:

```
<ul>
  <li th:each="e : ${#fields.detailedErrors()}" th:class="${e.global}? globalerr : fielderr">
    <span th:text="${e.global}? '*' : ${e.fieldName}">The field name</span> |
    <span th:text="${e.message}">The error message</span>
  </li>
</ul>
```


9它仍然是一个原型!

我们的申请现已准备就绪。但是让 `.html` 我们再看一下我们创建的页面.....

使用Thymeleaf最好的后果之一是，在我们添加到HTML之后，我们仍然可以将它用作原型（我们说它是一个*自然模板*）。让我们 `seedstartermng.html` 直接在浏览器中打开而不执行我们的应用程序：



STSM自然模板

它就是！它不是一个有效的应用程序，它不是真正的数据...但它是一个完美有效的原型，由完美可显示的HTML代码组成。尝试用JSP做到这一点！

10转换服务

10.1配置

如前所述, Thymeleaf可以使用在应用程序上下文中注册的转换服务。让我们再看一下它的样子:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    ...
    <mvc:annotation-driven conversion-service="conversionService" />
    ...

    <!-- ***** -->
    <!-- CONVERSION SERVICE -->
    <!-- Standard Spring formatting-enabled implementation -->
    <!-- ***** -->
    <bean id="conversionService"
        class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <property name="formatters">
            <set>
                <bean class="thymeleafexamples.stsm.web.conversion.VarietyFormatter" />
                <bean class="thymeleafexamples.stsm.web.conversion.DateFormatter" />
            </set>
        </property>
    </bean>

    ...

</beans>
```

10.1双括号语法

可以轻松应用转换服务, 以便将任何对象转换/格式化为String。这是通过双括号语法完成的:

- 对于变量表达式: `${{...}}`
- 对于选择表达式: `*{{...}}`

因此, 例如, 给定一个Integer-to-String转换器, 将逗号添加为千位分隔符, 这样:

```
<p th:text="${val}">...</p>
<p th:text="*{{val}}">...</p>
```

.....应该导致:

```
<p>1234567890</p>
<p>1,234,567,890</p>
```

10.2在表格中使用

之前我们看到每个 `th:field` 属性都将始终应用转换服务, 因此:

```
<input type="text" th:field="*{{datePlanted}}" />
```

.....实际上相当于:

```
<input type="text" th:field="*{{datePlanted}}" />
```

请注意, 这是使用单括号语法在表达式中应用转换服务的唯一方案。

10.3 **#conversions** 实用对象

的 **#conversions** 表达效用对象允许转换服务的手动执行任何需要的地方:

```
<p th:text="${'Val: ' + #conversions.convert(val,'String')}">...</p>
```

此实用程序对象的语法:

- **#conversions.convert(Object,Class)**: 将对象转换为指定的类。
- **#conversions.convert(Object,String)**: 与上面相同, 但将目标类指定为String (注意 **java.lang.** 可以省略包)。

11 渲染模板碎片

Thymeleaf提供了仅执行模板的一部分作为执行结果的可能性: *片段*。

这可以是一个有用的组件化工具。例如, 它可以在AJAX调用上执行的控制器上使用, 这可能会返回已经在浏览器中加载的页面的标记片段 (用于更新选择, 启用/禁用按钮.....) 。

通过使用Thymeleaf的*片段规范*: 实现 `org.thymeleaf.fragment.IFragmentSpec` 接口的对象, 可以实现片段渲染。

这些实现中最常见的是 `org.thymeleaf.standard.fragment.StandardDOMSelectorFragmentSpec`, 允许使用DOM选择器指定一个完全类似于 `th:include` 或使用的片段 `th:replace` 。

11.1 在视图bean中指定片段

*视图bean*是 `org.thymeleaf.spring4.view.ThymeleafView` 在应用程序上下文中声明的类的bean。它们允许像这样的片段的规范:

```
<bean name="content-part" class="org.thymeleaf.spring4.view.ThymeleafView">
  <property name="templateName" value="index" />
  <property name="fragmentSpec">
    <bean class="org.thymeleaf.standard.fragment.StandardDOMSelectorFragmentSpec"
      c:selectorExpression="content" />
  </property>
</bean>
```

鉴于上面的bean定义, 如果我们的控制器返回 **content-part** (上面的bean的名称) ...

```
@RequestMapping("/showContentPart")
public String showContentPart() {
    ...
    return "content-part";
}
```

... thymeleaf将仅返回模板的 **content** 片段 `index - /WEB-INF/templates/index.html` 一旦应用前缀和后缀, 该位置可能类似于:

```
<!DOCTYPE html>
<html>
  ...
  <body>
    ...
    <div th:fragment="content">
      Only this will be rendered!!
    </div>
    ...
  </body>
</html>
```

另请注意, 由于Thymeleaf DOM选择器的强大功能, 我们可以在模板中选择一个片段而不需要任何 **th:fragment** 属性。让我们使用该 **id** 属性, 例如:

```
<bean name="content-part" class="org.thymeleaf.spring4.view.ThymeleafView">
  <property name="fragmentSpec">
    <bean class="org.thymeleaf.standard.fragment.StandardDOMSelectorFragmentSpec"
      c:selectorExpression="#content" />
  </property>
  <property name="templateName" value="index" />
</bean>
```

...将完美选择:

```
<!DOCTYPE html>
<html>
  ...
  <body>
    ...
    <div id="content">
      Only this will be rendered!!
    </div>
    ...
  </body>
</html>
```

11.2 在控制器返回值中指定片段

可以使用与in 或属性相同的语法从控制器本身指定片段, 而不是声明视图bean: **th:include th:replace**

```
@RequestMapping("/showContentPart")
public String showContentPart() {
  ...
  return "index :: content";
}
```

当然, DOM Selectors的全部功能也是可用的, 因此我们可以根据标准HTML属性选择我们的片段, 例如 **id="content"** :

```
@RequestMapping("/showContentPart")
public String showContentPart() {
  ...
  return "index :: #content";
}
```

我们还可以使用参数, 例如:

```
@RequestMapping("/showContentPart")
public String showContentPart() {
  ...
  return "index :: #content ('myvalue')";
}
```

12高级集成功能

12.1与 集成 *RequestDataValueProcessor*

Thymeleaf现在可以与Spring的 **RequestDataValueProcessor** 界面无缝集成。此接口允许在将链接URL，表单URL和表单字段值写入标记结果之前截取它们，以及透明地添加隐藏的表单字段，以启用安全功能，例如再次保护CSRF（跨站点请求伪造）。

RequestDataValueProcessor 可以在应用程序上下文中轻松配置实现：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

    ...

    <bean name="requestDataValueProcessor"
          class="net.example.requestdata.processor.MyRequestDataValueProcessor" />

</beans>
```

.....而Thymeleaf则以这种方式使用它：

- **th:href** 并在呈现URL之前 **th:src** 调用 **RequestDataValueProcessor.processUrl(...)**。
- **th:action** **RequestDataValueProcessor.processAction(...)** 在呈现表单 **action** 属性之前调用，此外它还检测何时将此属性应用于 **<form>** 标记 - 无论如何应该是唯一的地方 - 并且在这种情况下调用 **RequestDataValueProcessor.getExtraHiddenFields(...)** 并在结束 **</form>** 标记之前添加返回的隐藏字段。
- **th:value** 要求 **RequestDataValueProcessor.processFormFieldValue(...)** 呈现它引用的值，除非 **th:field** 在同一个标记中存在（在这种情况下 **th:field** 需要注意）。
- **th:field** 要求 **RequestDataValueProcessor.processFormFieldValue(...)** 渲染它应用的字段的值（如果是a，则为标记体 **<textarea>**）。

请注意，此功能仅适用于Spring 3.1及更高版本。

12.1为控制器构建URI

从版本4.1开始，Spring允许直接从视图构建到带注释的控制器链接，而无需知道这些控制器映射到的URI。

在Thymeleaf中，这可以通过 **#mvc.url(...)** 表达式方法来实现，该方法允许通过它们所在的控制器类的大写字母来指定控制器方法，然后是方法本身的名称。这相当于JSP的 **spring:mvcUrl(...)** 自定义函数。

例如，对于：

```
public class ExampleController {

    @RequestMapping("/data")
    public String getData(Model model) { ... return "template" }

    @RequestMapping("/data")
    public String getDataParam(@RequestParam String type) { ... return "template" }

}
```

以下代码将创建一个指向它的链接:

```
<a th:href="${#mvc.url('EC#getData')).build()}">Get Data Param</a>  
<a th:href="${#mvc.url('EC#getDataParam').arg(0,'internal')).build()}">Get Data Param</a>
```

您可以在<http://docs.spring.io/spring-framework/docs/4.1.2.RELEASE/spring-framework-reference/html/mvc.html#mvc-links-to-controllers>上阅读有关此机制的更多信息。从-意见

13 Spring WebFlow集成

13.1 基本配置

该 **thymeleaf-spring4** 集成软件包包括与Spring Webflow的2.3.x整合

WebFlow包含一些AJAX功能，用于在触发特定事件（*转换*）时呈现显示页面的片段，并且为了使Thymeleaf能够参与这些AJAX请求，我们将不得不使用不同的 **ViewResolver** 实现，配置如下：

```
<bean id="thymeleafViewResolver" class="org.thymeleaf.spring4.view.AjaxThymeleafViewResolver">
    <property name="viewClass" value="org.thymeleaf.spring4.view.FlowAjaxThymeleafView" />
    <property name="templateEngine" ref="templateEngine" />
</bean>
```

...然后 **ViewResolver** 可以在您的WebFlow中配置， **ViewFactoryCreator** 如：

```
<bean id="mvcViewFactoryCreator"
    class="org.springframework.webflow.mvc.builder.MvcViewFactoryCreator">
    <property name="viewResolvers" ref="thymeleafViewResolver" />
</bean>
```

从这里开始，您可以在视图状态中指定Thymeleaf模板：

```
<view-state id="detail" view="bookingDetail">
    ...
</view-state>
```

在上面的示例中， **bookingDetail** 是一个以常规方式指定的Thymeleaf模板，可由任何配置的 *模板解析器*理解 **TemplateEngine** 。

13.2 Ajax片段

WebFlow允许通过带有 **<render>** 标记的AJAX呈现片段的规范，如下所示：

```
<view-state id="detail" view="bookingDetail">
    <transition on="updateData">
        <render fragments="hoteldata" />
    </transition>
</view-state>
```

这些片段（ **hoteldata** 在本例中）可以是以标记分隔的逗号分隔列表，其中包含 **th:fragment**：

```
<div id="data" th:fragment="hoteldata">
    This is a content to be changed
</div>
```

始终记住指定的片段必须具有**id**属性，以便在浏览器上运行的Spring JavaScript库能够替换标记。

<render> 也可以使用DOM选择器指定标签：

```
<view-state id="detail" view="bookingDetail">
    <transition on="updateData">
        <render fragments="//div[@id='data']" />
    </transition>
</view-state>
```



```
</transition>
</view-state>
```

.....这意味着 **th:fragment** 不需要:

```
<div id="data">
    This is a content to be changed
</div>
```

至于触发 **updateData** 转换的代码, 它看起来像:

```
<script type="text/javascript" th:src="@{/resources/dojo/dojo.js}"></script>
<script type="text/javascript" th:src="@{/resources/spring/Spring.js}"></script>
<script type="text/javascript" th:src="@{/resources/spring/Spring-Dojo.js}"></script>

...

<form id="triggerform" method="post" action="">
    <input type="submit" id="doUpdate" name="_eventId_updateData" value="Update now!" />
</form>

<script type="text/javascript">
    Spring.addDecoration(
        new Spring.AjaxEventDecoration({formId:'triggerform',elementId:'doUpdate',event:'onclick'}));
</script>
```