Tutorial: Thymeleaf + Spring 3

Document Version: 20110710 – 10 July 2011

Project Version: 1.0.0-beta5

Project web site: http://www.thymeleaf.org

CONTENTS

1 Integrating Thymeleaf with Spring 3

Thymeleaf offers a set of Spring integrations that allow you to use it as a full-featured substitute for JSP in Spring MVC applications.

These integrations are a separate library called `thymeleaf-spring3` and packed in a separate .jar file (`thymeleaf-spring3-{version}.jar`), which has to be added to your classpath in order to use these integrations in your applications.

This will allow you to:

- Make the mapped methods in your Spring MVC `@Controller` objects forward to templates managed by Thymeleaf, exactly like you do with JSPs.
- Use **Spring Expression Language** (Spring EL) instead of OGNL in your templates.
- Create forms in your templates that are completely integrated with your form-backing beans and result bindings, including the use of property editors and validation error handling.
- Display internationalization messages from messages files managed by Spring (through the usual `MessageSource` objects).

Note that in order to fully understand this tutorial, you should have first gone through the *"Using Thymeleaf"* tutorial, which explains the Standard Dialect in depth.

2 The Spring Thymeleaf Dialect

In order to achieve an easier and better integration, Thymeleaf provides a dialect which specifically implements all the needed features for it to work correctly with Spring.

This specific dialect is based on the Thymeleaf Standard Dialect and is implemented by a class called `org.thymeleaf.spring3.dialect.SpringThymeleafDialect`, which in fact extends from `org.thymeleaf.standard.StandardDialect`.

Besides all the features already present in the Standard Dialect – and therefore inherited –, the Spring Thymeleaf Dialect presents the following specific features:

- Use of Spring Expression Language (Spring EL) as a variable expression language, instead of OGNL. Consequently, all `${...}` and `*{...}` expressions will be evaluated by Spring's Expression Language engine.
- A new beans special variable that allows you to access any bean in your application context inside variable expressions: `${beans.myBean.doSomething()}`
- New attributes for form processing: `th:field` and `th:errors`, besides a new implementation of `th:object` that allows it to be used for form command selection.
- New DTDs for validation, including these new attributes, as well as new corresponding DOCTYPE translation rules.

Note that *you shouldn't use this dialect directly in a normal TemplateEngine object* as a part of its configuration. Instead, you should instance a new template engine class that performs all the required configuration steps: `org.thymeleaf.spring3.SpringThymeleafTemplateEngine`.

An example bean configuration:

```
<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.spring3.SpringThymeleafTemplateEngine">
  <property name="templateResolver" ref="templateResolver" />
</bean>
```

3 Views and View Resolvers

3.1 Views and View Resolvers in Spring MVC

There are two interfaces in Spring MVC that conform the core of its templating system:

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

Views model pages in our applications and allow us to modify and predefine their behaviour by defining them as beans. Views are in charge of rendering the actual HTML interface, usually by the execution of some template engine like JSP (or Thymeleaf).

ViewResolvers are the objects in charge of obtaining view objects for a specific operation and locale. Tipically, controllers ask ViewResolvers to forward to a view with a specific name (a String returned by the controller method), and then all the view resolvers in the application execute in ordered chain until one of them is able to resolve that view, in which case a View object is returned and control is passed to it for the renderization of HTML.

> Note that not all pages in our applications have to be defined as Views, but only those which behaviour we wish to be non-standard or configured in a specific way (for example, by wiring some special beans to it. If a ViewResolver is asked a view that has no corresponding bean –which is the common case–, a new View object is created ad hoc and returned.

A typical configuration for a JSP+JSTL ViewResolver in a Spring MVC application looks like this:

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

A quick look at its properties is enough to know about how it's configured:

- `viewClass` establishes the class of the View instances. This is needed for a JSP resolver, but it will not be needed at all when we're working with Thymeleaf.
- `prefix` and `suffix` work in a similar way to the attributes of the same names in Thymeleaf's TemplateResolver objects.
- `order` establishes the order in which the ViewResolver will be queried in the chain.
- `viewNames` allows the definition (with wildcards) of the view names that will be resolved by this ViewResolver.

3.2 Views and View Resolvers in Thymeleaf

Thymeleaf offers implementations for the two interfaces mentioned above:

- `org.thymeleaf.spring3.view.ThymeleafView`
- `org.thymeleaf.spring3.view.ThymeleafViewResolver`

These two classes will be in charge of processing Thymeleaf templates as a result of your controllers' executions.

Configuration of the View Resolver is very similar to that of JSP:

```
<bean class="org.thymeleaf.spring3.view.ThymeleafViewResolver">
  <property name="templateEngine" ref="templateEngine" />
  <property name="order" value="1" />
  <property name="viewNames" value="*.html,*.xhtml" />
</bean>
```

The `templateEngine` parameter is, of course, the `SpringThymeleafTemplateEngine` object we defined in the previous chapter. The other two (`order` and `viewNames`) are both optional, and have the same meaning as in the JSP ViewResolver we saw before.

Note that we do not need `prefix` or `suffix` parameters, because this are already specified in the Template Resolver (which in turn is passed to the Template Engine).

And what if we wanted to define a `View` bean and add some static variables to it? Easy:

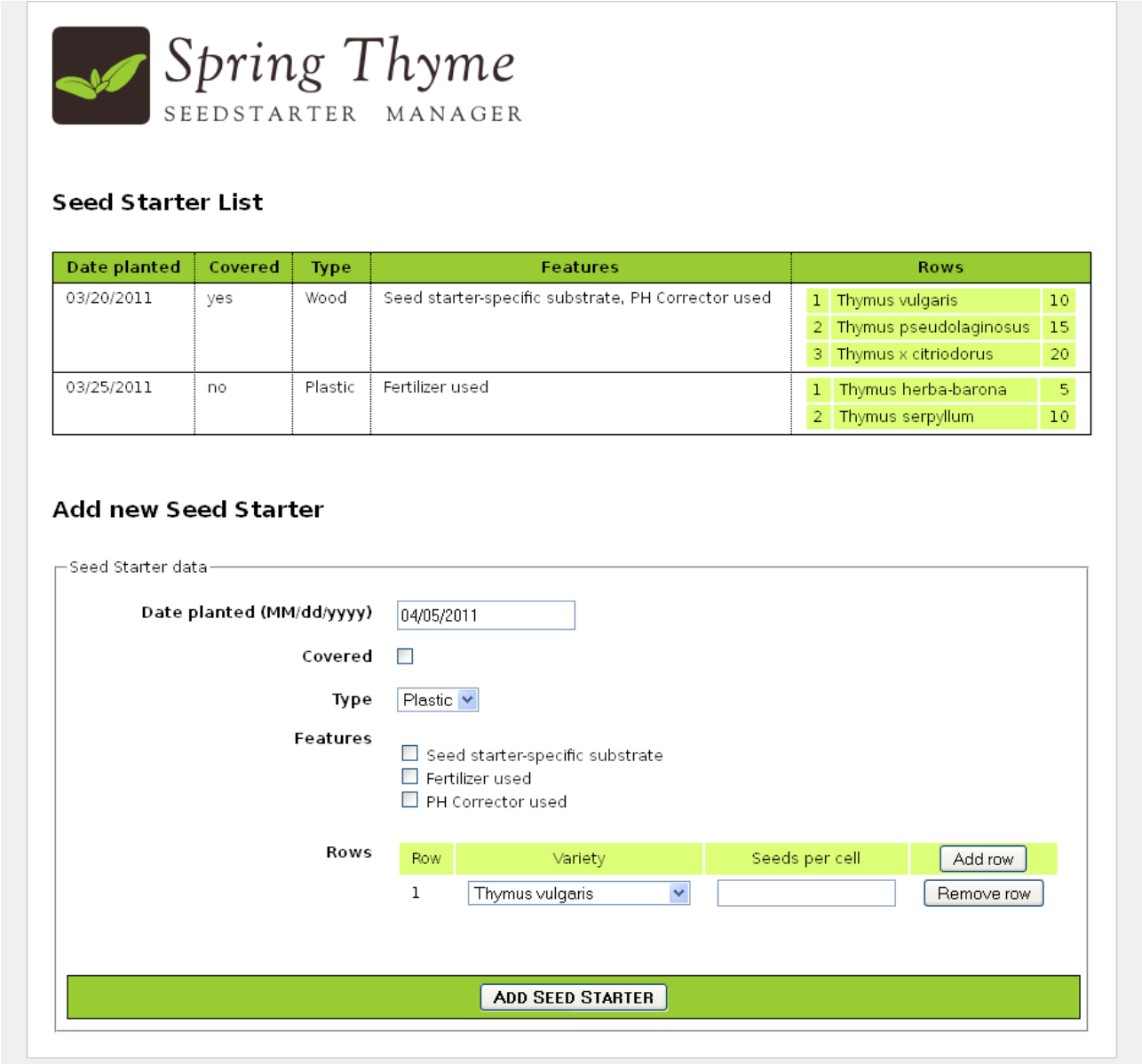
```
<bean name="main" class="org.thymeleaf.spring3.view.ThymeleafView">
  <property name="staticVariables">
    <map>
      <entry key="footer" value="Some company: &lt;b&gt;ACME&lt;/b&gt;" />
    </map>
  </property>
</bean>
```

4 Spring Thyme Seed Starter Manager

4.1 The Concept

At Thymeleaf we're huge fans of thyme, and every spring we prepare our seed starting kits with good soil and our favourite seeds, place them under the Spanish sun and patiently wait for our new plants to grow.

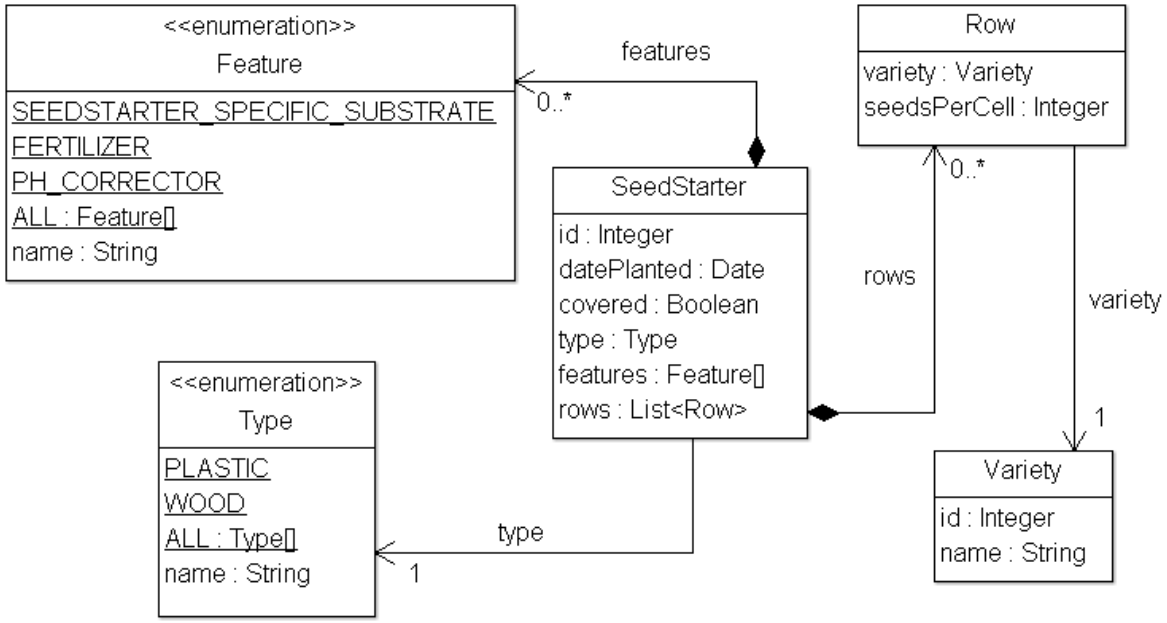
But this year we got fed up with sticking labels to the seed starter containers for knowing which seed was in each cell of the container, so we decided to prepare an application using Spring MVC and Thymeleaf to help us catalogue our starters: The *Spring Thyme SeedStarter Manager*.



In a similar way to the Good Thymes Virtual Grocery application we developed in the *"Using Thymeleaf"* tutorial, the STSM will allow us to exemplify the most important aspects of the integration of Thymeleaf as a template engine for Spring MVC.

4.2 Business Layer

We will need a very simple business layer for our application. First of all, let's have a look at our model entities:



A couple of very simple service classes will provide the required business methods. Like:

```
@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);
    }

}
```

And:

```
@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);
    }

}
```

4.3 Spring MVC configuration

Next we need to set up the Spring MVC configuration for the application, which will include not only the standard Spring MVC artifacts like resource handling or annotation scanning, but also the creation of the Template Engine and View Resolver instances.

```
<?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 />
  <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>


  <!-- ************************************************************** -->
  <!--  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.spring3.SpringThymeleafTemplateEngine">
    <property name="templateResolver" ref="templateResolver" />
  </bean>

  <bean class="org.thymeleaf.spring3.view.ThymeleafViewResolver">
```

```
    <property name="templateEngine" ref="templateEngine" />
  </bean>


</beans>
```

Important: Note that we have selected HTML5 as a template mode.

4.4 The Controller

Of course, we will also need a controller for our application. As the STSM will only contain one web page with a list of seed starters and a form for adding new ones, we will write only one controller class for all the server interactions:

```
@Controller
public class SeedStarterMngController {

    @Autowired
    private MessageSource messageSource;

    @Autowired
    private VarietyService varietyService;

    @Autowired
    private SeedStarterService seedStarterService;

    ...

}
```

Now let's see what we can add to this controller class.

Model Attributes

First we will add some model attributes that we will need in the page:

```
@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();
}
```

Property Editors

Property Editors in a Spring MVC application allow forms to contain fields of types other than String, and define the way in which values of these types will be converted to and from String

so that they can be shown in form inputs and transmitted over the network when users click submit.

We will need two property editors: one for java.util.Date objects (locale-dependent) and a second one for Variety objects.

```
@InitBinder
public void initDateBinder(final WebDataBinder dataBinder, final Locale locale) {
    final String dateformat =
        this.messageSource.getMessage("date.format", null, locale);
    final SimpleDateFormat sdf = new SimpleDateFormat(dateformat);
    sdf.setLenient(false);
    dataBinder.registerCustomEditor(Date.class, new CustomDateEditor(sdf, false));
}


@InitBinder
public void initVarietyBinder(final WebDataBinder dataBinder) {
    dataBinder.registerCustomEditor(Variety.class, new VarietyPropertyEditor(this.varietyService));
}
```

Mapped methods

And now the most important part of a controller, the mapped methods: one for showing the form page, and other for processing the addition of new Seed Starter objects.

```
@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 Listing Seed Starter Data

The first thing that our `/WEB-INF/templates/seedstartermng.html` page will show is a listing with the seed starters currently stored. For this we will need some externalized messages and also some expression evaluation on model attributes. Like this:

```
<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 : ${allSeedStarters}">
        <td th:text="${#dates.format(sb.datePlanted, #messages.msg('date.format'))}">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>
```

Lots to see here. Let's have a look at each fragment separately.

First of all, this section will only be shown if there are any seed starters. We achieve that with a `th:unless` attribute and the `#lists.isEmpty(...)` function.

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

Note that all utility objects like `#lists` are available in Spring EL expressions just as they were in OGNL expressions in the Standard Dialect.

The next thing to see is a lot of internationalized (externalized) texts, like:

```
  <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>
        …
```

This being a Spring MVC application, we already defined a `MessageSource` bean in our spring XML configuration (`MessageSource` objects are the standard way of managing externalized texts in Spring MVC):

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

...and that `basename` property indicates that we will have files like `Messages_es.properties` or `Messages_en.properties` in our classpath. Let's have a look at the Spanish version:

```
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
```

In the first column of the table listing we will show the date when the seed starter was prepared. In order to do that we will obtain the format pattern as an externalized text by using the `#mesages.msg(...)` function and then use it for formatting the `Date` object.

```
<td th:text="${#dates.format(sb.datePlanted, #messages.msg('date.format'))}">13/01/2011</td>
```

Next is showing whether the seed starter container is covered or not, by transforming the value of the boolean `covered` bean property into an internationalized “yes” or “no” with a conditional expression:

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

Now we have to show the type of seed starter container. Type is a java enum with two values (`WOOD` and `PLASTIC`), and that's why we defined two properties in our `Messages` file called `seedstarter.type.WOOD` and `seedstarter.type.PLASTIC`.

But in order to obtain the internationalized names of the types, we will need to add the '`seedstarter.type.`' prefix to the enum value by means of an expression, which result we will then use as the message key:

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

The most difficult part of this listing is the *features* column. In it we want to display all the features of our container –that come in the form of an array of `Feature` enums–, separated by commas. Like *“Electric Heating, Turf”*.

Note that this is particularly difficult because these enum values also need to be externalized, as we did with Types. The flow is then:

1. Prepend the corresponding prefix to all the elements of the `features` array.
2. Obtain the externalized messages corresponding to all the keys from step 1.
3. Join all the messages obtained in step 2, using a comma as a delimiter.

For achieving this, we create the following code:

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

The last column of our listing will be quite simple, in fact. Even if it has a nested table for showing the contents of each row in the container:

```
<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>
```

6 Creating a Form

6.1 Handling the command object

Command object is the name Spring MVC gives to form-backing beans, this is, to objects that model a form's fields and provide getter and setter methods that will be used by the framework for establishing and obtaining the values input by the user at the browser side.

Thymeleaf requires you to specify the command object by using a `th:object` attribute in your `<form>` tag:

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

    ...

</form>
```

This is quite consistent with any other uses of th:object, but in fact this specific scenario adds some limitations in order to correctly integrate with Spring MVC's infrastructure:

- Values for `th:object` attributes in form tags must be variable expressions (`${...}`) specifying only the name of a model attribute, without property navigation. This means that an expression like `${seedStarter}` is valid, but `${seedStarter.data}` would not be.
- Once inside the `<form>` tag, no other `th:object` attribute can be specified. This is consistent with the fact that HTML forms cannot be nested.

6.2 Inputs

Let's see now how to add an input to our form:

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

As you can see, we are introducing a new attribute here: `th:field`. This is a very important feature for Spring MVC integration because it does all the heavy work of binding your input with a property in the form-backing bean. You can see it as an equivalent of the `path` attribute in a `<form:input>` tag from Spring MVC's JSP tag library.

The `th:field` attribute behaves differently depending on whether it is attached to an `<input>`, `<select>` or `<textarea>` tag (and also depending on the specific `type` of `<input>` tag). In this case (input[type=text]), the above line of code is similar to:

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

...but in fact it is a little bit more than that, because `th:field` will take care of the *property editor* we have defined for `java.util.Date` objects and use it correspondingly to show the property value (which `th:value` will not do).

Values for `th:field` attributes must be selection expressions (`*{...}`), which makes sense given the fact that they will be evaluated on the form-backing bean and not on the context variables (or *model attributes* in Spring MVC jargon).

Contrary to the ones in `th:object`, these expressions *can* include property navigation (in fact any expression allowed for the `path` attribute of a `<form:input>` JSP tag will be allowed here).

> Note that `th:field` also understands the new types of `<input>` element introduced by HTML5 like `<input type="datetime" ... />`, `<input type="color" ... />`, etc., effectively adding complete HTML5 support to Spring MVC.

6.3 Checkbox fields

`th:field` also allows us to define checkbox inputs. Let's see an example from our HTML page:

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

Note there's some fine stuff here besides the checkbox itself, like an externalized label and also the use of the `#ids.next('covered')` function for obtaining the value that will be applied to the id attribute of the checkbox input.

Why do we need this dynamic generation of an `id` attribute for this field? Because checkboxes are potentially multi-valued, and thus their `id` values will always be suffixed a sequence number (by internally using the `#ids.seq(...)` function) in order to ensure that each of the checkbox inputs for the same property has a different id value.

We can see this more easily if we look at such a multi-valued checkbox field:

```
<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>
```

Note that we've added a `th:value` attribute this time, because the `features` field is not a boolean like `covered` was, but instead is an array of values.

Let's see the HTML output generated by this code:

```
<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>
```

We can see here how a sequence suffix is added to each input's `id` attribute, and how the `#ids.prev(...)` function allows us to retrieve the last sequence value generated for a specific input `id`.

> 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.

6.4 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>
```

6.5 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.

6.6 Dynamic fields

Thanks to the advanced form-field binding capabilities in Spring MVC, we can use complex Spring EL expressions to bind dynamic form fields to our form-backing bean. This will allow us to create new Row objects in our SeedStarter bean, and to add those rows' fields to our form at user request.

In order to do this, we will need a couple of new mapped methods in our controller, which will add or remove a row from our SeedStarter depending on the existence of specific request parameters:

```
@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";
}
```

And now we can add a dynamic table to our form:

```
<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>
      <th>
        <button type="submit" name="addRow" th:text="#{seedstarter.row.add}">Add row</button>
      </th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="row,rowStat : *{rows}">
      <td th:text="${rowStat.count}">1</td>
      <td>
        <select th:field="*{rows[__${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>
```

Quite a lot of things to see here, but not much we should not understand by now... except for one *strange* thing:

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

    ...

</select>
```

If you recall from the *"Using Thymeleaf"* tutorial, that __${...}__ syntax is a *preprocessing expression*, which is an inner expression that is evaluated before actually evaluating the whole expression. But why that way of specifying the row index? Wouldn't it be enough with:

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

    ...

</select>
```

...well, actually, no. The problem is that Spring EL does not evaluate variables inside array index brackets, so when executing the above expression we would obtain an error telling us that `rows[rowStat.index]` (instead of `rows[0]`, `rows[1]`, etc) is not a valid position in the `rows` collection. That's why preprocessing is needed here.

Let's have a look at a fragment of the resulting HTML after pressing "Add Row" a couple of times:

```
<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>
```

7 Validation and error messages

Most of our forms will need to show validation messages in order to inform the user of the errors he/she has made.

Thymeleaf offers two tools for this: a couple of functions in the #fields object and the th:errors attribute.

Functions first: let's set a class to a field if it has an error:

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

As you can see, the `#fields.hasErrors(...)` function receives the field expression as a parameter, and returns a boolean telling whether any validation errors exist for that field.

Let's show the error messages themselves:

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

The new function is `#fields.errors(...)`, and the star (`'*'`) parameter tells the function we are looking for errors on any field. `#fields.errors(...)` returns a list of (externalized) error messages.

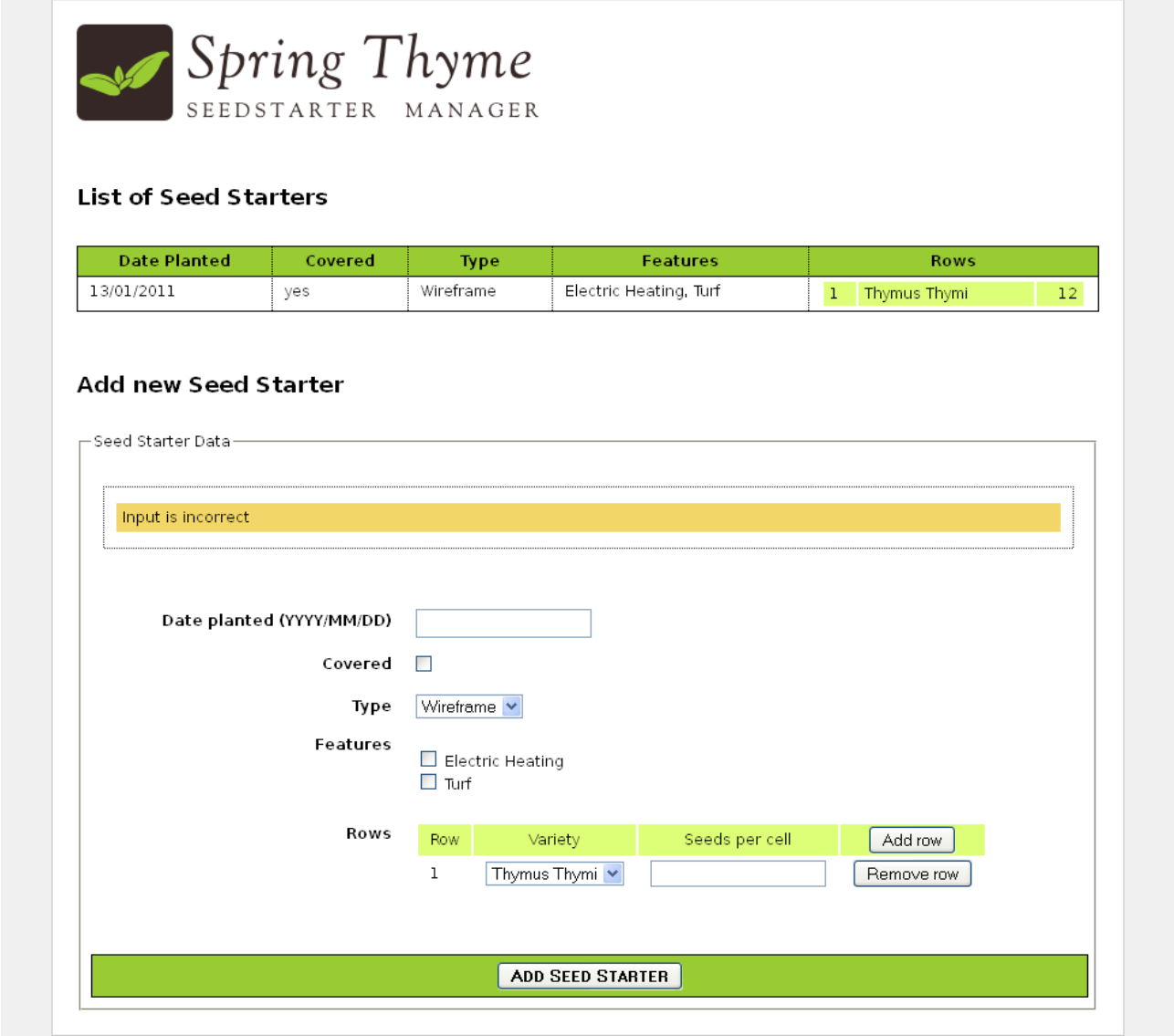
`th:errors` attribute works in a similar way to the `#fields.errors(...)` function, but listing all errors for the specified selector separated by `<br />`:

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

8 It's still a prototype!

Our application is ready now. But let's have a second look at the `.html` page we created...

One of the nicest consequences of working with Thymeleaf is that after all this functionality we have added to our HTML, we can still use it as a prototype. Let's open `seedstartermng.html` directly in our browser without executing our application:



There it is! It's not a working application, it's not real data... but it is a perfectly valid prototype made up of perfectly displayable HTML code. Try to do that with JSP!