# WHAT'S NEW IN THYMELEAF 2.1

Thymeleaf 2.1 includes a lot of new powerful features. Here they are:

- Ease of use:
    - Render view fragments directly from controllers
- Form error handling:
    - New **`th:errorclass`** for adding CSS class to form fields in error
    - Additional form validation error reporting options
- Security:
    - Transparent integration with Spring's **`RequestDataValueProcessor`**

# NEW FEATURES IN THYMELEAF CORE

## Same-template fragments

Thymeleaf now allows including fragments from the same template. There are two available syntaxes: one specifying no template name:

```
<div th:include="::fragment_name">...</div>
```

...and another one specifying the keyword **`this`**:

```
<div th:include="this :: fragment_name">...</div>
```

## Parameterizable fragment signatures

In order to create a more *function-like* mechanism for the use of template fragments, fragments defined with **`th:fragment`** can now specify a set of parameters:

```
<div th:fragment="frag (onevar,twovar)">
    <p th:text="${onevar} + ' - ' + ${twovar}">.
</div>
```

This requires the use of one of these two syntaxes to call the fragment from **`th:include`**, **`th:substituteby`** or **`th:replace`**:

```
<div th:include="::frag (${value1},${value2})">...</di
```

```
<div th:include="::frag (onevar=${value1},twovar=${val
```

Note that order is not important in the last option:

```
<div th:include="::frag (twovar=${value2},onevar=${val
```

### *Fragment local variables without fragment signature*

Even if fragments are defined without signature, like this:

```
<div th:fragment="frag">
    ...
</div>
```

We could use the second syntax specified above to call them (and only the second one):

```
<div th:include="::frag (onevar=${value1},twovar=${val
```

This would be, in fact, equivalent to a combination of **`th:include`** and **`th:with`**:

```
<div th:include="::frag" th:with="onevar=${value1},two
```

But the difference is, this will also work with **`th:substituteby`** (and the new **`th:replace`**). Until now these attributes didn't work well with **`th:with`** because they effectively removed the host tag before **`th:with`** got executed:

```
<!-- th:with will never be executed. Variables w
<div th:replace="::frag" th:with="onevar=${value
```

**Note** that this specification of local variables for a fragment —no matter whether it has a signature or not— does not cause an initialization of the context to zero. Fragments will still be able to access every context variable being used at the calling template like they currently are.

## More powerful DOM selector syntax

DOM Selector syntax has been enhanced to include more selection possibilities and easier selection of DOM fragments.

These selectors are a way to specify fragments already available since thymeleaf 2.0:

```
<div th:include="mytemplate :: [//div[@class='content'
```

Thymeleaf 2.1 adds to these Selectors some syntax features borrowed from CSS and jQuery, in order to make them more powerful and easy to use:

- Now **`x`** is exactly equivalent to **`//x`** (search an element with name or *reference* "**x**" at any depth level).
- Allowed selectors without element name/reference, as long as it includes a specification of arguments. So **`[@class='oneclass']`** is now a valid selector that looks for any elements (*tags*) with a **`class`** attribute with value **`"oneclass"`**.
- Attribute selection enhancements:
  - Besides "**=**" (equal)., other comparison operators are now also valid: "**!=**" (not equal), "**^=**" (starts with) and "**$=**" (ends with). For example: **`x[@class^='section']`** means *elements with name **`x`** and a value for attribute class that starts with **`section`***.
  - Attributes can be specified both starting with **`@`** (XPath-style) and without (jQuery-style). So **`x[z='v']`** is now equivalent to **`x[@z='v']`**.
  - Multiple-attribute modifiers can now be joined both with "**and**" (XPath-style) and also by chaining multiple modifiers (jQuery-style). So **`x[@z1='v1' and @z2='v2']`** is actually equivalent to **`x[@z1='v1'][@z2='v2']`** (and also to **`x[z1='v1'][z2='v2']`**).
- Direct selectors:
  - **`x.oneclass`** is now equivalent to **`x[class='oneclass']`**.
  - **`.oneclass`** is now equivalent to **`[class='oneclass']`**.
  - **`x#oneid`** is now equivalent to **`x[id='oneid']`**.
  - **`#oneid`** is now equivalent to **`[id='oneid']`**.
  - **`x%oneref`** means nodes -not just *elements*- with name **`x`** that match reference **`oneref`** according to a specified **`DOMSelector.INodeReferenceChecker`** implementation.
  - **`%oneref`** means nodes -not just *elements*- with any name that match reference **`oneref`** according to a specified **`DOMSelector.INodeReferenceChecker`** implementation.
    - Note this is actually equivalent to simply **`oneref`** because references can be used instead

of element names.

  - Direct selectors and attribute selectors can be mixed: **`a.external[@href^='https']`**.
- Specific feature: DOM Selectors now understand the **`class`** attribute to be multivalued, and therefore allow the application of selectors on this attribute even if the element has several class values. For example, **`div[class='two']`** will match **`<div class="one two three"/>`**

So now the above DOM Selector expression:

```
<div th:include="mytemplate :: [//div[@class='content'
```

...could be written as:

```
<div th:include="mytemplate :: [div.content]">...</div
```

### *Fragment specifications adapted to new DOM Selectors*

Also, in order to better take advantage of the new selector syntax, the syntax of the fragment inclusion attributes (like the above **`th:include`**) has been modified to *convert every fragment selection into a DOM selection*, so that brackets **`[...]`** are no longer needed (though *allowed*).

So the following, with no brackets, is now equivalent to the bracketed selector seen above:

```
<div th:include="mytemplate :: div.content">...</div>
```

So, summarizing, this:

```
<div th:replace="mytemplate :: myfrag">...</div>
```

Will look for a **`th:fragment="myfrag"`** fragment signature. But would also look for tags with name **`myfrag`** if they existed (which they don't, in HTML). Note the difference with:

```
<div th:replace="mytemplate :: .myfrag">...</div>
```

...which will actually look for any elements with **`class="myfrag"`**, without caring about **`th:fragment`** signatures.

## Literal tokens

Thymeleaf 2.1 allows for a little bit of simplification in Standard Expressions (i.e. outside OGNL or SpringEL variable expressions), thanks to the introduction of *literal tokens*.

These tokens work exactly the same as literals (`'...'`), but they only allow letters (`A-Z` and `a-z`), numbers (`0-9`), brackets (`[` and `]`), dots (`.`), hyphens (`-`) and underscores (`_`). So no whitespaces, no commas, etc.

The nice part? tokens don't need any quotes surrounding them. So we can now do this:

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

...instead of:

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

## Boolean and null tokens

Boolean and null tokens can now be used (they are *reserved tokens*). The following is now valid:

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

Note the difference with what was allowed in 2.0, when the only way to do this was to specify the `false` token inside the OGNL/SpringEL expression and therefore let these expression engines evaluate it (not thymeleaf).

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

The `null` token also works as expected:

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

## Literal substitutions

The new literal substitutions in Thymeleaf Standard Expressions allow the easy formatting of strings which may contain values from variables without the need to append literals with `'...' + '...'`.

These substitutions must be surrounded by vertical bars (`|`), like:

```
<span th:text="|Welcome to our application, ${user.nam
```

Which is actually equivalent to:

```
<span th:text="'Welcome to our application, ' + ${user
```

Literal substitutions can be combined with other types of expressions:

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

**Note**: Note: only variable expressions are allowed inside `|...|` literal substitutions. No other literals (`'...'`), boolean/numeric tokens, conditional expressions etc. are.

## Protocol-relative URLs

Thymeleaf 2.1 allows now protocol-relative URLs, like:

```
<script th:src="@{//ajax.googleapis.com/ajax/libs/jque
```

## Parser-level comment blocks

Parser-level comment blocks are code that will be simply removed from the template when thymeleaf parses it. They should look like this:

```
<!--/* This code will be removed at thymeleaf parsing
```

Thymeleaf will remove absolutely everything between **`<!--/*`** and **`*/-->`**, so these comment blocks can be used not only for template comments that shouldn't appear in the final result, but also for displaying code when a template is statically open, knowing that it will be removed when thymeleaf processes it:

```
<!--/*-->
<div>
    you can see me only before thymeleaf process
</div>
<!--*/-->
```

This might come very handy for prototyping tables with a lot of

`<tr>`'s, for example:

```
<table>
    <tr th:each="x : ${xs}">
        ...
    </tr>
    <!--/*-->
    <tr>
        ...
    </tr>
    <tr>
        ...
    </tr>
    <!--*/-->
</table>
```

Note that this feature is *dialect-independent*. So it will be available for us even if we don't use the Standard Dialects.

## Prototype-only comment blocks

As an evolution of parser-level comment blocks, Thymeleaf 2.1 allows the definition of special comment blocks marked to be comments when the template is open statically (i.e. as a *prototype*), but *considered normal markup* by thymeleaf when executing the template.

```
<span>hello!</span>
<!--/*/
  <div th:text="${...}">
    ...
  </div>
/*/-->
<span>goodbye!</span>
```

Thymeleaf's parsing system will simply remove the `<!--/*/` and `/*/-->` markers, but not its contents, which will be left therefore *uncommented*. So when executing the template, thymeleaf will actually see this:

```
<span>hello!</span>

  <div th:text="${...}">
    ...
  </div>

<span>goodbye!</span>
```

As with parser-level comment blocks, note that this feature is also *dialect-independent*.

## Improved `th:remove` attribute

The `th:remove` attribute can take now any Thymeleaf Standard Expression, as long as it returns one of the following String values:

- `all`
- `tag`
- `body`
- `all-but-first`
- `none`

This means removals could now be conditional, like:

```
<a href="/something" th:remove="${condition}? tag : no
```

Also note that `th:remove` could consider `null` a synonym to `none`, so that the following works exactly as the example above:

```
<a href="/something" th:remove="${condition}? tag">Lin
```

If `${condition}` is false, null will be returned, and thus no removal will be performed.

## Synthetic `th:block` tag

Thymeleaf 2.1 adds the first *element processor* to the Standard Dialects (until now, all processors were *attribute-based*): `th:block`

`th:block` is a mere attribute container that allows template developers to specify whichever attributes they want, executes them, and then simply dissapears without a trace. So it could be useful, for example, when creating iterated tables that require more than one `<tr>` for each element:

```
<table>
  <th:block th:each="user : ${users}">
    <tr>
        <td th:text="${user.login}">...</td>
        <td th:text="${user.name}">...</td>
    </tr>
    <tr>
        <td colspan="2" th:text="${user.address
    </tr>
  </th:block>
</table>
```

And especially useful when used in combination with *prototype-only comment blocks*:

```
<table>
    <!--/*/ <th:block th:each="user : ${users}"
    <tr>
        <td th:text="${user.login}">...</td>
        <td th:text="${user.name}">...</td>
    </tr>
    <tr>
        <td colspan="2" th:text="${user.address
    </tr>
    <!--/*/ </th:block> /*/-->
</table>
```

Note how this solution allows templates to be valid HTML (no need to add forbidden **`<div>`** blocks inside **`<table>`**), and still work OK when open statically in browsers as prototypes!

## Support for HTML5-friendly attribute and element names

Thymeleaf 2.1 adds a completely new syntax we can use to apply processors to our templates, more *HTML5-friendly*.

```
<table>
    <tr data-th-each="user : ${users}">
        <td data-th-text="${user.login}">...</td
        <td data-th-text="${user.name}">...</td>
    </tr>
</table>
```

The **`data-{prefix}-{name}`** syntax is the standard way to write *custom attributes* in HTML5, without requiring developers to use any namespaced names like **`th:*`**. Thymeleaf 2.1 makes this syntax automatically available to all our dialects (not only the Standard ones).

There is also a new syntax to specify custom tags: **`{prefix}-{name}`**, which follows the *W3C Custom Elements* specification (a part of the larger *W3C Web Components* spec). We can use this, for example, for the new **`th:block`** element (or also **`th-block`**):

```
<!-- ========================================
<!-- Fully HTML5-valid, both prototype and work
<!-- ========================================
<table>
    <!--/*/ <th-block data-th-each="user : ${us
    <tr>
        <td data-th-text="${user.login}">...</t
        <td data-th-text="${user.name}">...</td
    </tr>
    <tr>
        <td colspan="2" data-th-text="${user.ad
    </tr>
```

```
        <!--/*/ </th-block> /*/-->
    </table>
```

**Important**: this new syntax is an **addition** to the namespaced **`th:*`** one, it does not replace it. There is no intention at all to deprecate the namespaced syntax in the future.

## New `th:assert` for in-template assertions

A new *attribute processor* is now available: **`th:assert`**. This attribute can specify a comma-separated list of expressions which should be evaluated and produce **`true`** for every evaluation, raising an exception if not.

```
<div th:assert="${onevar},(${twovar} != 43)">...</div>
```

This comes in handy for validating parameters at a fragment signature:

```
<header th:fragment="contentheader(title)" th:assert="
```

## New `th:replace` as a synonym of `th:substituteby`

For semantic reasons, a new **`th:replace`** has been introduced as a full synonym of **`th:substituteby`** (in fact, it was *sneakily* added in 2.0.18):

```
<div th:replace="mytemplate :: .myfrag">...</div>
```

Template developers are now recommended to use **`th:replace`** instead of **`th:substituteby`**, as the latter will probably be deprecated (not *removed*) in thymeleaf 3.0.

## Reuse variables in `th:with`

As a minor optimization, the **`th:with`** attribute now allows *reusing* variables defined in the same attribute:

```
<div th:with="company=${user.company + ' Co.'},account
```

## New Features in Spring Integration

### More integrated acccess to beans from expressions

Thymeleaf now allows us to access beans registered in our Spring application context in the standard way defined by Spring EL, which is using the syntax **`@beanName`**:

```
<div th:text="${@authService.getUserName()}">...</div>
```

Until now, we could access beans with the thymeleaf-specific syntax **`beans.beanName`**:

```
<div th:text="${beans.authService.getUserName()}">...<
```

Note that this latter syntax is now considered deprecated in favour of the standard Spring EL one (**`@beanName`**).

### Integration of Spring type conversion infrastructure

Spring 3 introduced a type conversion system more general than *Property Editors*: the *Spring Type Conversion System* (see [docs.spring.io]). This system is mainly based on *Converters* (X-to-Y one-way conversion) and *Formatters* (X-to-String two-way conversion), easily implemented by means of their corresponding interfaces, and registered at a *conversion service* in the application context.

Thymeleaf now seamlessly integrates with our *conversion service* thanks to the introduction of double-bracket expressions, which apply conversion on their result:

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

So we can now have a formatter like this:

```
public class CalendarFormatter implements Forma

    private static final SimpleDateFormat SDF =

    public CalendarFormatter() {
        super();
    }

    public Calendar parse(String text, Locale l
        synchronized (SDF) {
            final Date date = SDF.parse(text);
            final Calendar cal = Calendar.getIn
            cal.setTimeInMillis(date.getTime())
            return cal;
        }
    }

    public String print(Calendar object, Locale
        synchronized (SDF) {
            return SDF.format(object.getTime())
        }
    }

}
```

Registered in our application context like this:

```
<mvc:annotation-driven conversion-service="conv

<bean id="conversionService"
      class="org.springframework.format.support
    <property name="formatters">
        <set>
            <bean class="mycompany.myapp.Calend
        </set>
    </property>
</bean>
```

Thymeleaf will allow us to use our formatter whenever we need it. So given a **`Calendar`** variable in context with name **`onedate`**, the following:

```
<p th:text="${{onedate}}">...</p>
<p th:text="|The date is ${{onedate}}|">...</p>
```

Results in:

```
<p>1492-10-12</p>
<p>The date is 1492-10-12</p>
```

### *Conversion in Spring forms*

Besides, whenever the converted (double-bracketed) expression refers to a Spring-bound object (e.g. a form-backing bean) thymeleaf will not only apply the conversion service but also the property editors and also any formatting annotations. So given:

```
<form th:object="${obj}">
    <p>Date: <span th:text="*{{date}}">somedate<
</form>
```

If that date field in **`${obj}`** has an annotation like:

```
@DateTimeFormat(pattern = "yyyy-MM-dd")
private Date date;
```

The result will be:

```
<form>
    <p>Date: <span>1492-10-12</span></p>
</form>
```

### *The #conversions utility object*

Besides this double-bracket syntax, a new expression utility object has been added, allowing the manual execution of the conversion service whenever needed:

```
<p th:text="${'Val: ' + #conversions.convert(val
```

Syntax for this utility object:

- **`#conversions.convert(Object,Class)`**: converts the object to the specified class.
- **`#conversions.convert(Object,String)`**: same as above, but specifying the target class as a String (note the **`java.lang.`** package can be ommitted).

## Spring-resource based template resolver

Thymeleaf now includes a new **`ITemplateResolver`** implementation, besides the standard ones. This new implementation is called **`org.thymeleaf.spring3.templateresolver.SpringResourceTemplateResolver`**. It can be specified at the application context like:

```
...
```

```
<bean id="templateResolver"
      class="org.thymeleaf.spring3.templatereso
    <property name="templateMode" value="HTML5"
    <property name="cacheable" value="true" />
</bean>

<bean id="templateEngine"
      class="org.thymeleaf.spring3.SpringTempla
    <property name="templateResolver" ref="temp
</bean>
...
```

This new implementation delegates on Spring's own resource resolution mechanism (**`ApplicationContext.getResource(resourceName)`**), so templates can be now selected in the same ways Spring itself allows to specify resources, like:

```
@RequestMapping("/")
public String index() {
    ...
    return "classpath:appres/templates/index.htm
}
```

## Render view fragments directly from controllers

Thymeleaf now allows specifying template fragments whenever a view name is returned after controller execution. So given the following template called **`myTemplate`**:

```
<!DOCTYPE html>
<html>
  ...
  <body>
    ...
    <div th:fragment="myFrag">
      <ul th:each="p : ${products}">
        <li th:text="${p.name}">One product</li
      </ul>
    </div>
    ...
  </body>
</html>
```

A Spring MVC controller can render the **`myFrag`** fragment with:

```
@RequestMapping("/frag")
public String showFrag(final ModelMap model) {
    model.addAttribute("products", this.productR
    return "myTemplate :: myFrag";
}
```

This is a very useful feature for controller methods meant to be called via AJAX, which can now just render the part of the HTML they really need.

Besides, the new DOM Selector syntax provides a lot of power, and there is actually no need to specify **`th:fragment`** at all:

```
...
<div id="content">
    ...
</div>
...
```

...and then:

```
@RequestMapping("/frag")
public String showFrag() {
    ...
    return "myTemplate :: #content";
}
```

## New `th:errorclass` for adding CSS class to form fields in error

Until now, whenever we wanted to apply a specific CSS class to an input field in a form when there were errors for that field, we needed to use the **`th:class`** or **`th:classappend`** attributes like this:

```
<input type="text" th:field="*{age}"
       th:class="${#fields.hasErrors('age')}? 'error'"
```

In Thymeleaf 2.1, in order to simplify this structure, a new **`th:errorclass`** attribute processor has been introduced. This processor will read the name of the field from the **`name`** or **`th:field`** attribute in the same tag, and apply the specified class if such field has errors.

```
<input type="text" th:field="*{age}" th:errorclass="'e
```

Note the **`'error'`** literal is in fact a token, so no single quotes are really needed:

```
<input type="text" th:field="*{age}" th:errorclass="er
```

The result is much more concise. Note also that **`th:errorclass`** works like **`th:classappend`**, not **`th:class`**. So the specified class will in fact be *appended* to any existing ones. So:

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

Will result, if the age field has errors, in:

```
<input type="text" id="age" name="age" value="23" clas
```

## Additional form validation error reporting options

Thymeleaf now implements a more comprehensive expression support for form validation errors (besides all the previously supported artifacts):

- Field errors:
    - Conditional: **`${#fields.hasErrors('*{fieldName}')}`**
    - Error list: **`${#fields.errors('*{fieldName}')}`**
- Global errors:
    - Conditional: **`${#fields.hasGlobalErrors()}`**, **`${#fields.hasErrors('global')}`**
    - Error list: **`${#fields.globalErrors()}`**, **`${#fields.errors('global')}`**
- All errors:
    - Conditional: **`${#fields.hasAnyErrors()}`**, **`${#fields.hasErrors()}`**, **`${#fields.hasErrors('all')}`**
    - Error list: **`${#fields.allErrors()}`**, **`${#fields.errors()}`**, **`${#fields.errors('all')}`**

And now form validation errors can also be accessed outside forms themselves, by prepending the form-backing bean name (and using **`${...}`** instead of **`*{...}`**):

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

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

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

## Transparent integration with Spring's `RequestDataValueProcessor`

Thymeleaf now seamlessly integrates with Spring's **`RequestDataValueProcessor`** interface. This interface allows the interception of link URLs, form URLs and form field values before they are written to the markup result, as well as transparently adding hidden form fields that enable security features like e.g. protection agains CSRF *(Cross-Site Request Forgery)*.

See the interface definition here, and also this article as an example of its use for CSRF.

An implementation of **`RequestDataValueProcessor`** can be easily configured at the Application Context:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/sc
       xmlns:xsi="http://www.w3.org/2001/XMLSch
       xsi:schemaLocation="http://www.springfra
                           http://www.springframework

    ...

    <bean name="requestDataValueProcessor"
          class="net.example.requestdata.proces

</beans>
```

...and Thymeleaf uses it this way:

- **`th:href`** and **`th:src`** call **`RequestDataValueProcessor.processUrl(...)`** before rendering the URL.
- **`th:action`** calls **`RequestDataValueProcessor.processAction(...)`** before rendering the form's **`action`** attribute, and additionally it detects when this attribute is being applied on a **`<form>`** tag —which should be the only place, anyway—, and in such case calls **`RequestDataValueProcessor.getExtraHiddenFields(...)`** and adds the returned hidden fields just before the closing **`</form>`** tag.
- **`th:value`** calls **`RequestDataValueProcessor.processFormFieldValue(...)`** for rendering the value it refers to, unless there is a

`th:field` present in the same tag (in which case `th:field` will take care).

- `th:field` calls `RequestDataValueProcessor.processFormFieldValue(...)` for rendering the value of the field it applies to (or the tag body if it is a `<textarea>`).

Note this feature will only be available for Spring versions 3.1 and newer.

Follow @thymeleaf