Web-based information systems 1

Thymeleaf templating techniques

Michiel Noback (NOMI)
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Introduction

discussed techniques:

- texts
- variables (of the context/model)
- conditionals
- loops
- urls/links
- predefined variables
- …

Expression types and implicit objects

Resource expressions: #{ }

`#{message.in.resource.bundle}` evaluates to texts stored in properties files

This expression:

```
<h3 th:text="#{page.title}">_offline_</h3>
```

combined with this property:

```
page.title=get your phrase-of-the-day
```

will evaluate to

```
<h3>get your phrase-of-the-day</h3>
```

Unescaped text: th:utext

th:text 'escapes' the text in the message. If you want this message:

```
favourite=Thymeleaf is my <b>favourite</b> templating engine!
```

To display correctly, us th:utext:

```
<h3 th:utext="#{page.title}">_offline_</h3>
```

So you'll get this

```
<p>Thymeleaf is my <b>favourite</b> templating
engine!</p>
```

Instead of this

```
<p>Thymeleaf is my &lt;b&gt;favourite&lt;/b&gt;
templating engine!</p>
```

Messages with variables

It is easy to insert variables into your messages. Give the message identifier an argument, and catch it with `{0}`:

```
animal=The {0} is the scariest animal there is!
```

```
<p th:text="#{animal('black widow')}"></p>
<p th:text="#{animal('rattlesnake')}"></p>
```

```
The black widow is the scariest animal there is!
The rattlesnake is the scariest animal there is!
```

(Use `{1}`, `{2}` … for more variables)

Variable expressions: ${}

This servlet code:

```
ctx.setVariable("warning",
    "you must be over 18 to drive a car");
```

With this Thymeleaf expression

```
<p th:text="${warning}">_offline_</p>
```

will evaluate to

 you must be over 18 to drive a car

Java beans

- A Java bean is a special kind of class, coded to a set of rules. That's why you can use them so conveniently in EL
  - They provide a no-arg constructor (e.g. `public Person(){}`)
  - They provide public standard-named getters and setters for class attributes; e.g. for String property `name` you must provide the methods `public String getName(){return "Fred"}` and `public void setName(String name){}`
  - The class does not necessary have to have the actual property (private String name); only the getters and setters are required!

Flexibility of **${}**

`${}` can be used in many ways:

- Chaining of calls
  - `${object.property.property}`
- List access
  - `${list[1]}` or `${list[${selected_index}]}`
- Map access
  - `${map['foo']}` or
  - `${map[${selected_item}].property}`
- Regular method calls
  - `${person.createCompleteNameWithSeparator('-')}`
- Combine
  - `<h4 th:text="#{'phrase.' + ${phrase_type} + '.' + ${phrase_num}}">_phrase_</h4>`

Built-in-variable expressions: ${# }

- Thymeleaf provides some predefined variables that can be accessed using **`${#variableName}`** syntax; here are some:
  - `#locale`: the context locale.
  - `#httpServletRequest`: (only in Web Contexts) the HttpServletRequest object.
    - access via `${param.key}` in Spring
  - `#dates`: utility methods for java.util.Date objects: formatting, component extraction, etc.
  - `#numbers`: utility methods for formatting numeric objects.
  - `#strings/objects/bools/arrays/lists/sets`: utility methods for these objects

Session attributes: `${session.var}`

- Thymeleaf provides easy access to session attributes (sessions will be dealt with in a separate presentation):
- `${session.user.email}` will display the 'email' property of the User object registered on the 'session' object with

```
session.setAttribute("logged_in_user",
  new User("Henk",
           "henk@example.com",
           Role.USER));
```

- Result: ”henk@example.com”

Locally scoped variables: *{}

- With a locally scoped object, you can access it through the *{} syntax:
- Given the User object of the session example, you can list its properties as follows

```
<ul th:object="${session.logged_in_user}">
    <li>Name: <span th:text="*{name}">_name_</span></li>
    <li>Email: <span th:text="*{email}">_eml_</span></li>
    <li>Role: <span th:text="*{role}">_role_</span></li>
</ul>
```

- Result:
  - Name: Henk
  - Email: henk@example.com
  - Role: USER

Creating links: @{ }

- Using the @{ } syntax, you can create links with a path that is relative to the deployment context

```
<a th:href="@{/give.phrase(phrase_category=bullshit)}"
href="#">get a bullshit phrase</a>
```

- Will result in:

```
<a href="/give.phrase?phrase_category=bullshit">
get a bullshit phrase</a>
```

Inline expressions

- Sometimes it is not desirable to have your variables expressed within tag attributes. Instead, you want raw html text.
- This is especially the case with Javascript
- Here are its variants:

Inline expressions: th:text

- This is the simple th:text equivalent:

```
<p>
Simple text: access a variable attribute: Hello,
<span>[[${session.logged_in_user.name}]]!</span>
</p>
```

Inline expressions: Javascript

- This is the simple th:text equivalent:

```
<button id="say_hello">Say hello!</button>

<script th:inline="javascript">
    var user = [[${session.logged_in_user.name}]];
    document.getElementById("say_hello").onclick =
          function() {
              alert("Hi, " + user);
          }
</script>
```

Flow control

The Movie class

- The following series of examples all make use of the Movie class which is shown on the next slide
- The movies list is registered on the model as:

```
ctx.setVariable("movies",
                    Movie.getAllMovies());
```

```java
public class Movie {
    private String title;
    private int year;
    private double rating = -1;
    private Set<String> mainActors = new HashSet<>();

    //constructors omitted
    //getters & setters omitted

    public List<String> getMainActors() {
        List<String> actors = new ArrayList<>();
        actors.addAll(mainActors);
        actors.sort(String::compareTo);
        return actors;
    }

    public void addActor(String actor) {
        mainActors.add(actor);
    }
    public static List<Movie> getAllMovies() {
        //serves some movies
    }
}
```

Iteration: "th: each"

- Iteration is as simple as in Java code:

```
<table>
  <thead>
    <tr>
      <th>Title</th><th>Year of release</th><th>IMDB rating</th>
  </tr>
</thead>
  <tbody>
      <tr th:each="movie:${movies}">
      <td th:text="${movie.title}">_title_</td>
      <td th:text="${movie.year}">_year_</td>
      <td th:text="${movie.rating}">_rating_</td>
    </tr>
  </tbody>
</table>
```

`th:each` with status variable

- Thymeleaf also provides a mechanism that stores the state of the iteration process. It has several useful properties
  - index: the current iteration index, starting at zero
  - count: the number of elements processed so far
  - size: the total number of elements in the list
  - even/odd: checks if the current iteration index is even or odd
  - first: checks if the current iteration is the first one
  - last: checks if the current iteration is the last one

th:each with status variable

```
<tr th:each="movie, it_stat:${movies}"
    th:class="${it_stat.first} ?
        'first' : (${it_stat.even} ? 'even' : 'odd')">
    <td th:text="${it_stat.count} + ' / ' + ${it_stat.size}"></td>
    <td th:text="${movie.title}">_title_</td>
    <td th:text="${movie.year}">_year_</td>
    <td th:text="${movie.rating}">_rating_</td>
</tr>
```

Pos	Title	Year of release	IMDB rating
1 / 8	The Shawshank Redemption	1994	9.2
2 / 8	The Dark Knight	2008	9.0
3 / 8	Pulp Fiction	1994	8.9
4 / 8	Fight Club	1999	8.8
5 / 8	Forrest Gump	1994	8.7
6 / 8	Inception	2010	8.7
7 / 8	One Flew Over the Cuckoo's Nest	1975	8.7
8 / 8	The Usual Suspects	1995	8.5

Iteration with sorters

- Using sorters is supported with the `#lists` utility object:

```
ctx.setVariable("movies_year_sorter",
    (Comparator<Movie>) (o1, o2)
        -> Integer.compare(o1.getYear(), o2.getYear()));
```

```
<tr th:each="movie:
            ${#lists.sort(movies, movies_year_sorter)}">
```

Conditionals: ternary

- The simplest form is the ternary:

```
<table>
  <thead>
    <tr>
      <th>Title</th><th>Year of release</th><th>IMDB rating</th>
  </tr>
</thead>
  <tbody>
      <tr th:each="movie:${movies}">
      <td th:text="${movie.title}">_title_</td>
      <td th:text="${movie.year}">_year_</td>
      <td th:text="${movie.rating}">_rating_</td>
    </tr>
  </tbody>
</table>
```

Conditionals: if

- Besides the ternary operator structure shown in the previous example, regular if test can be applied.
- Here is a test for the rating of the movie; only movies with a rating higher than 9 are displayed:

```
<ul>
    <li th:each="movie:${movies}"
        th:text="${movie.title} + ' (' + ${movie.year}
             + ') - ' + ${movie.rating}"
        th:if="${movie.rating >= 9}">_movie_
    </li>
</ul>
```

What is true?

The specified expression evaluates to **true** following these rules:

- If value is not null:
  - If value is a boolean and is true.
  - If value is a number and is non-zero
  - If value is a character and is non-zero
  - If value is a String and is not ?false?, ?off? or ?no?
  - If value is not a boolean, a number, a character or a String.
- If value is null, th:if will evaluate to false

Conditionals: switch

- Here you see a combination of iteration and switch/case to build an ordered list.

```
<ol><li th:each="user:${users}”
      th:switch="${user.role.toString()}">
    <span th:case="${'ADMIN’}”
          th:text="${user.name} + ' (' + ${user.role} + ‘)
                manages all accounts'">_admin_</span>
    <span th:case="${'USER’}”
          th:text="${user.name} + ' (' + ${user.role} + ') can
                browse and share all site content'">_user_</span>
    <span th:case="${'GUEST’}”
          th:text="${user.name} + ' (' + ${user.role} + ') can
                only see our front page'">_guest_</span>
    <span th:case="*"
          th:text="${user.name} + '(' + ${user.role} + ') we
                do not know this role'">_unknown_</span>
  </li></ol>
```


Accessing
attributes

Attributes you can access

You have seen attributes such as "th:text" a lot by now. But there are many many more. Have a look at The Thymeleaf docs for a complete listing.

Here are the main ones:

- th:action - the action of a form
- th:class - the style class of an element
- th:href - the url a link will point to
- th:id - the ID of an element
- th:name - the name of an element
- th:rel - url for style sheets
- th:src - the source of an element (e.g. image)
- th:title - the title
- th:value - the value of a (form) element, e.g.
- th:attr - a generic means to define any attribute, e.g. th:attr="data-my-attr=#{subscribe.submit}"

Attributes example

Here is an example demonstrating some of these (result on next slide)

```
<select th:name="${name_of_bird_selector}">
   <option th:each="bird_cat:${bird_groups}"
         th:value="${bird_cat}"
         th:text="'The ' + ${bird_cat}"
         th:id="${bird_cat + '_option'}"
         value="_bird_cat_">_bird_cat_</option>
</select>
```

```
<select th:name="${name_of_bird_selector}">
   <option th:each="bird_cat:${bird_groups}"
         th:value="${bird_cat}"
         th:text="'The ' + ${bird_cat}"
         th:id="${bird_cat + '_option'}"
         value="_bird_cat_">_bird_cat_</option>
</select>
```

```
<select name="fav_birds">
   <option id="raptors_option"
         value="raptors">The raptors</option>
   <option id="songbirds_option"
         value="songbirds">The songbirds</option>
   <option id="waders_option"
         value="waders">The waders</option>
   <option id="wildfowl_option"
         value="wildfowl">The wildfowl</option>
</select>
```


Using
fragments

Why fragments?

- Parts of your pages will be reused in other pages.
- This is especially the case for banners, footers and headers.
- Instead of copy-and-paste, Thymeleaf offers a nice mechanism for reusing page fragments.
- The strategy of choice here is to create a single Thymeleaf html page called template or fragments and to include elements of
  this page in others.

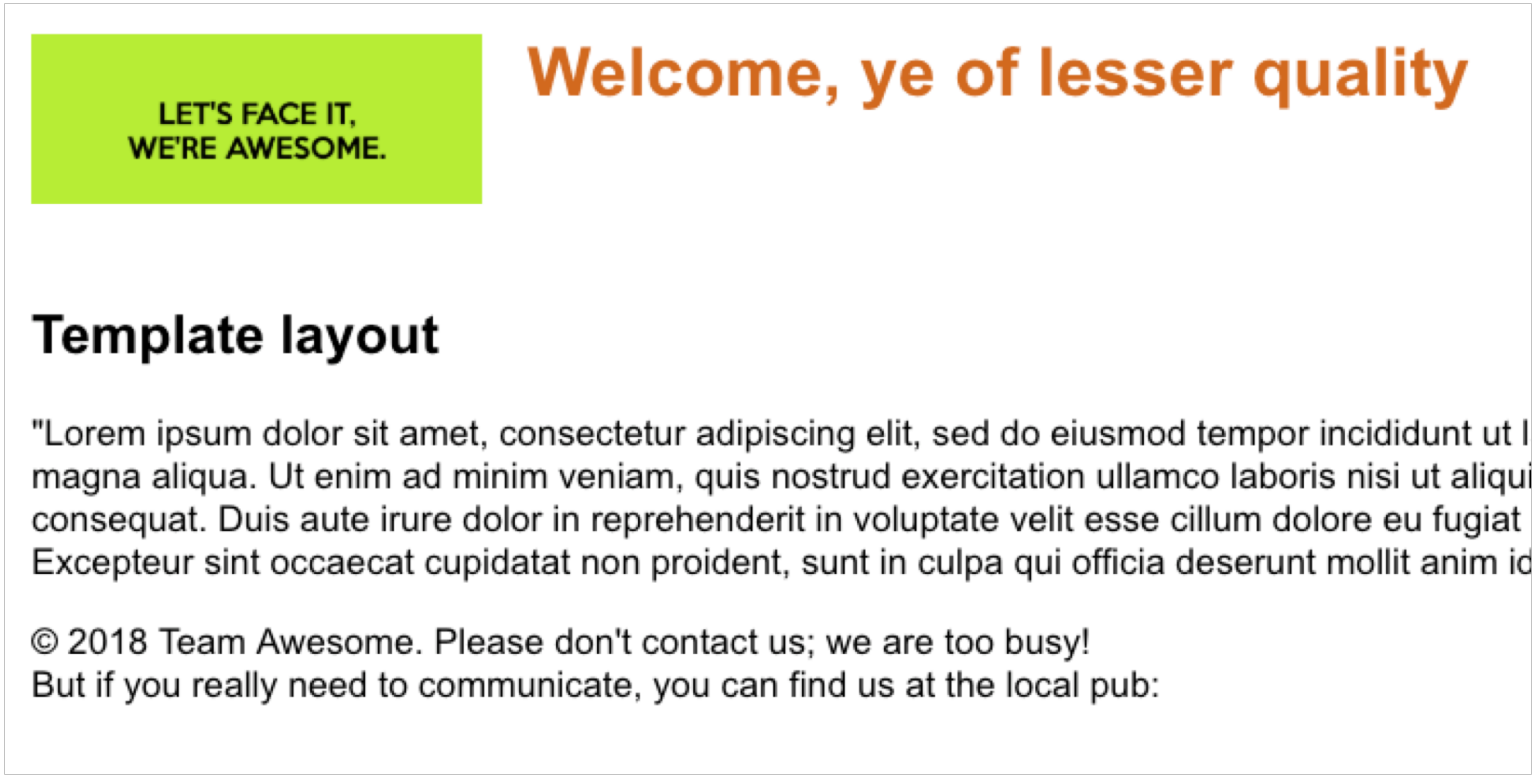
template.html

```
<!DOCTYPE html SYSTEM ”...">
<html ...>
<head th:fragment="headerFragment">
    <title>_TITLE_</title>
    <link rel="stylesheet" th:href="@{/css/main.css}" href="../../css/main.css">
</head>
<body>
    <div th:fragment="banner">
        <img id="logo" th:src="@{/graphics/Awesome.png}" />
        <h1 id="banner_title">Welcome, ye of lesser quality</h1>
    </div>

    <div>
        <h2>Template layout</h2>
        <p>
            "Lorem ipsum dolor sit amet, ..."
        </p>
    </div>

  <div th:fragment="footer(pub)">
      &copy; 2018 Team Awesome. Please don't contact us; we are too busy!<br/>
      But if you really need to communicate, you can find us at the local pub:
      <span th:text="${pub}">_PUB_</span><br />
  </div>
</body></html>
```

template.html



Using fragments

This is how you insert the fragment

```
<!– insert with fragment expression -->
<div th:insert="~{template :: banner}"></div>
<!– insert with fragment expression simplified -->
<div th:insert="template :: banner"></div>

<!– insert with argument to the fragment -->
<div th:insert="~{template :: footer(${the_pub})}"></div>
```

Using fragments: advanced

You can even insert a variant of a fragment depending on the value of some variable

```
<div th:fragment="menu" class="admin">
    <a href="#">my secret content</a>
</div>
<div th:fragment="menu" class="normaluser">
    <a href="#">my regular content</a>
</div>
```

```
<!– insert menu variant depending on role of user -->
<div th:insert="~{template ::
     (${users[0].getRole().toString() == 'ADMIN'} ?
     'menu.admin' : 'menu.normaluser')}">
</div>
```

insert / replace

- The difference between `th:insert` and th:replace
  - th:insert is the simplest: it will simply insert the specified fragment as the body of its host tag.
  - th:replace actually replaces its host tag with the specified fragment.