(/)

Introduction to Using Thymeleaf in Spring

Last updated: January 8, 2024

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Reviewed by: Grzegorz Piwowarek (https://www.baeldung.com/editor/grzegorz-author)

Spring MVC (https://www.baeldung.com/category/spring/spring-web/spring-mvc)

reference

Spring MVC Basics (https://www.baeldung.com/tag/spring-mvc-basics)

Thymeleaf (https://www.baeldung.com/tag/thymeleaf)

1. Overview

Thymeleaf (http://www.thymeleaf.org/) is a Java template engine for processing and creating HTML, XML, JavaScript, CSS and text.

In this tutorial, we will discuss **how to use Thymeleaf with Spring** along with some basic use cases in the view layer of a Spring MVC application.

The library is extremely extensible, and its natural templating capability ensures we can prototype templates without a back end. This makes development very fast when compared with other popular template engines such as JSP.

2. Integrating Thymeleaf With Spring

First, let's see the configurations required to integrate with Spring. The *thymeleaf-spring* library is required for the integration.

We'll add the following dependencies to our Maven POM file:

```
<dependency>                    (/)
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf</artifactId>
    <version>3.1.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring5</artifactId>
    <version>3.1.2.RELEASE</version>
</dependency>
```

Note that, for a Spring 4 project, we have to use the *thymeleaf-spring4* library instead of *thymeleaf-spring5*.

The *SpringTemplateEngine* class performs all of the configuration steps.

We can configure this class as a bean in the Java configuration file:

```
@Bean
@Description("Thymeleaf Template Resolver")
public ServletContextTemplateResolver templateResolver() {
    ServletContextTemplateResolver templateResolver = new
ServletContextTemplateResolver();
    templateResolver.setPrefix("/WEB-INF/views/");
    templateResolver.setSuffix(".html");
    templateResolver.setTemplateMode("HTML5");

    return templateResolver;
}

@Bean
@Description("Thymeleaf Template Engine")
public SpringTemplateEngine templateEngine() {
    SpringTemplateEngine templateEngine = new SpringTemplateEngine();
    templateEngine.setTemplateResolver(templateResolver());
    templateEngine.setTemplateEngineMessageSource(messageSource());
    return templateEngine;
}
```

The *templateResolver* bean properties *prefix* and *suffix* indicate the location of the view pages within the *webapp* directory and their filename extension, respectively.

The *ViewResolver* interface in Spring MVC maps the view names returned by a controller to actual view objects. *ThymeleafViewResolver* implements the *ViewResolver* interface, and it's used to determine which Thymeleaf views to render, given a view name.

The final step in the integration is to add the *ThymeleafViewResolver* as a bean:

```
@Bean
@Description("Thymeleaf View Resolver")
public ThymeleafViewResolver viewResolver() {
    ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
    viewResolver.setTemplateEngine(templateEngine());
    viewResolver.setOrder(1);
    return viewResolver;
}
```

3. Thymeleaf in Spring Boot

Spring Boot provides auto-configuration for *Thymeleaf* by adding the *spring-boot-starter-thymeleaf* (https://mvnrepository.com/search?q=spring-boot-starter-thymeleaf) dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
    <version>2.3.3.RELEASE</version>
</dependency>
```

No explicit configuration is necessary. By default, HTML files should be placed in the *resources/templates* location.

4. Displaying Values From Message Source (Property Files)

We can use the *th:text="#{key}"* tag attribute to display values from property files.

For this to work, we need to configure the property file as a *messageSource* bean:

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

Here is the Thymeleaf HTML code to display the value associated with the key *welcome.message*:

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

5. Displaying Model Attributes

5.1. Simple Attributes

We can use the *th:text="${attributename}"* tag attribute to display the value of model attributes.

Let's add a model attribute with the name *serverTime* in the controller class:

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

And here's the HTML code to display the value of *serverTime* attribute:

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

5.2. Collection Attributes (7)

If the model attribute is a collection of objects, we can use the *th:each* tag attribute to iterate over it.

Let's define a *Student* model class with two fields, *id* and *name*:

```
public class Student implements Serializable {
    private Integer id;
    private String name;
    // standard getters and setters
}
```

Now we will add a list of students as model attribute in the controller class:

```
List<Student> students = new ArrayList<Student>();
// logic to build student data
model.addAttribute("students", students);
```

Finally, we can use Thymeleaf template code to iterate over the list of students and display all field values:

```
<tbody>
    <tr th:each="student: ${students}">
        <td th:text="${student.id}" />
        <td th:text="${student.name}" />
    </tr>
</tbody>
```

6. Conditional Evaluation

6.1. *if* and *unless*

We use the *th:if="${condition}"* attribute to display a section of the view if the condition is met. And we use the *th:unless="${condition}"* attribute to display a section of the view if the condition is not met.

Let's add a *gender* field to the *Student* model:

```
public class Student implements Serializable {
    private Integer id;
    private String name;
    private Character gender;

    // standard getters and setters
}
```

Suppose this field has two possible values (M or F) to indicate the student's gender.

If we wish to display the words "Male" or "Female" instead of the single character, we could do this using this Thymeleaf code:

```
<td>
    <span th:if="${student.gender} == 'M'" th:text="Male" />
    <span th:unless="${student.gender} == 'M'" th:text="Female" />
</td>
```

6.2. *switch* and *case*

We use the *th:switch* and *th:case* attributes to display content conditionally using the switch statement structure.

Let's rewrite the previous code using the *th:switch* and *th:case* attributes:

```
<td th:switch="${student.gender}">
    <span th:case="'M'" th:text="Male" />
    <span th:case="'F'" th:text="Female" />
</td>
```

7. Handling User Input

We can handle form input using the *th:action="@{url}"* and *th:object="${object}"* attributes. We use *th:action* to provide the form action URL and *th:object* to specify an object to which the submitted form data will be bound.

Individual fields are mapped using the *th:field="*{name}"* attribute, where the *name* is the matching property of the object.

For the *Student* class, we can create an input form:

```
<form action="#" th:action="@{/saveStudent}" th:object="${student}"
method="post">
    <table border="1">
        <tr>
            <td><label th:text="#{msg.id}" /></td>
            <td><input type="number" th:field="*{id}" /></td>
        </tr>
        <tr>
            <td><label th:text="#{msg.name}" /></td>
            <td><input type="text" th:field="*{name}" /></td>
        </tr>
        <tr>
            <td><input type="submit" value="Submit" /></td>
        </tr>
    </table>
</form>
```

In the above code, */saveStudent* is the form action URL and a *student* is the object that holds the form data submitted.

The *saveStudent* method handles the form submission:

```
@RequestMapping(value = "/saveStudent", method = RequestMethod.POST)
public String saveStudent(Model model, @ModelAttribute("student")
Student student) {
    // logic to process input data
}
```

The *@RequestMapping* annotation maps the controller method with the URL provided in the form. The annotated method *saveStudent()* performs the required processing for the submitted form. Finally, the *@ModelAttribute* annotation binds the form fields to the *student* object.

8. Displaying Validation Errors

We can use the *#fields.hasErrors()* function to check if a field has any validation errors. And we use the *#fields.errors()* function to display errors for a particular field. The field name is the input parameter for both these functions.

Let's take a look at the HTML code to iterate and display the errors for each of the fields in the form:

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

Instead of field name, the above functions accept the wild card character *'*' or the constant *all* to indicate all fields. We used the *th:each* attribute to iterate the multiple errors that may be present for each of the fields.

Here's the previous HTML code rewritten using the wildcard *'*':

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

And here we're using the constant *all*:

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

Similarly, we can display global errors in Spring using the *global* constant.

Here's the HTML code to display global errors:

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

Also, we can use the *th:errors* attribute to display error messages.

The previous code to display errors in the form can be rewritten using *th:errors* attribute:

```
<ul>                          (/)
    <li th:errors="*{id}" />
    <li th:errors="*{name}" />
</ul>
```

9. Using Conversions

We use the double bracket syntax *{{}}* to format data for display. This makes use of the *formatters* configured for that type of field in the *conversionService* bean of the context file.

Let's see how to format the name field in the *Student* class:

```
<tr th:each="student: ${students}">
    <td th:text="${{student.name}}" />
</tr>
```

The above code uses the *NameFormatter* class, configured by overriding the *addFormatters()* method from the *WebMvcConfigurer* interface.

For this purpose, our *@Configuration* class overrides the *WebMvcConfigurerAdapter* class:

```
@Configuration
public class WebMVCConfig extends WebMvcConfigurerAdapter {
    // ...
    @Override
    @Description("Custom Conversion Service")
    public void addFormatters(FormatterRegistry registry) {
        registry.addFormatter(new NameFormatter());
    }
}
```

The *NameFormatter* class implements the Spring *Formatter* interface.

We can also use the *#conversions* utility to convert objects for display. The syntax for the utility function is *#conversions.convert(Object, Class)* where *Object* is converted to *Class* type.

Here's how to display *student* object *percentage* field with the fractional part removed:

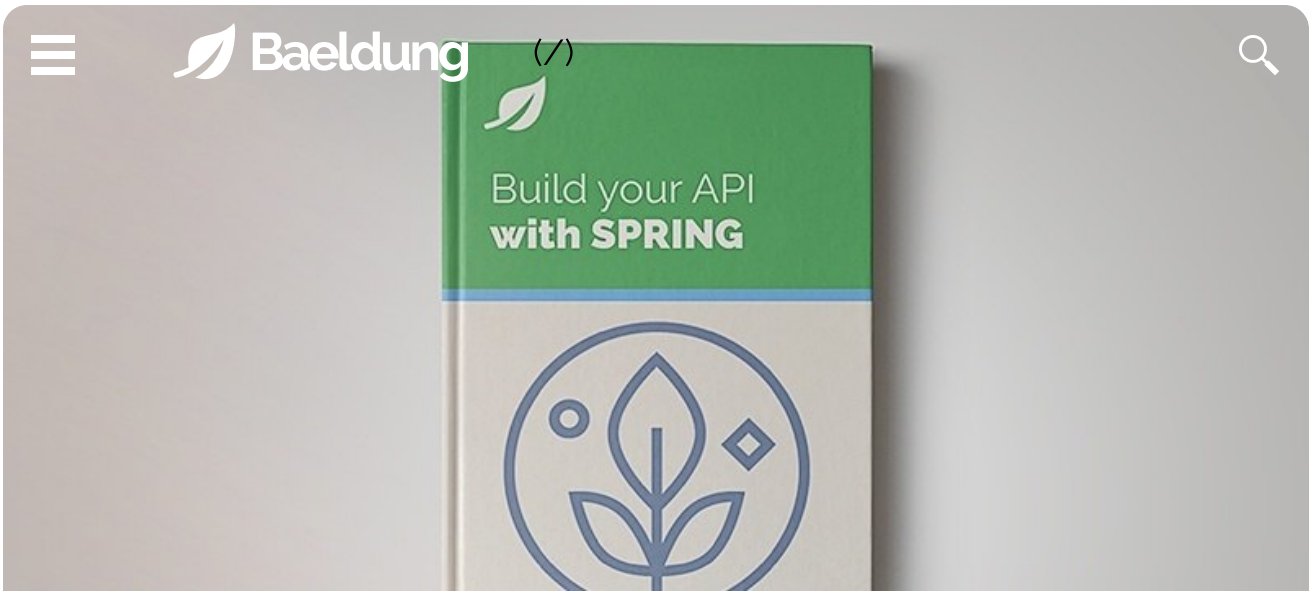
```
<tr th:each="student: ${students}">
    <td th:text="${#conversions.convert(student.percentage,
'Integer')}" />
</tr>
```

10. Conclusion

In this article, we've seen how to integrate and use Thymeleaf in a Spring MVC application.

We have also seen examples of how to display fields, accept input, display validation errors, and convert data for display.

A working version of the code shown in this article is available in the GitHub repository (https://github.com/eugenp/tutorials/tree/master/spring-web-modules/spring-thymeleaf).



Comments are closed on this article!

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)

APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)
OUR PARTNERS (/PARTNERS)
PARTNER WITH BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)