Spring Boot Thymeleaf

What is Thymeleaf?

The**Thymeleaf** is an open-source Java library that is licensed under the **Apache License 2.0**. It is a **HTML5/XHTML/XML** template engine. It is a **server-side Java template**engine for both web (servlet-based) and non-web (offline) environments. It is perfect for modern-day HTML5 JVM web development. It provides full integration with Spring Framework.

It applies a set of transformations to template files in order to display data or text produced by the application. It is appropriate for serving XHTML/HTML5 in web applications.

The goal of Thymeleaf is to provide a **stylish** and **well-formed**way of creating templates. It is based on XML tags and attributes. These XML tags define the execution of predefined logic on the DOM (Document Object Model) instead of explicitly writing that logic as code inside the template. It is a substitute for **JSP**.

The architecture of Thymeleaf allows the **fast** **processing** of templates that depends on the caching of parsed files. It uses the least possible amount of I/O operations during execution.

Why we use Thymeleaf?

JSP is more or less similar to HTML. But it is not completely compatible with HTML like Thymeleaf. We can open and display a Thymeleaf template file normally in the browser while the JSP file does not.

Thymeleaf supports variable expressions (${...}) like Spring EL and executes on model attributes, asterisk expressions (\*{...}) execute on the form backing bean, hash expressions (#{...}) are for internationalization, and link expressions (@{...}) rewrite URLs.

Like JSP, Thymeleaf works well for Rich HTML emails.

AD

What kind of templates can the Thymeleaf process?

Thymeleaf can process six types of templates (also known as **Template Mode**) are as follows:

* XML
* Valid XML
* XHTML
* Valid XHTML
* HTML5
* Legacy HTML5

Except for the Legacy HTML5 mode, all the above modes refer to **well-defined XML** files. It allows us to process HTML5 files with features such as **standalone tags, tag attributes without value, or not written between quotes**.

To process files in this specific mode, Thymeleaf performs a transformation that converts files into a **well-formed XML**file (valid HTML5 file).

Note: In Thymeleaf, validation is available only for XHTML and XML template.

Thymeleaf also allows us to define our own mode by specifying both a way to parse templates in this mode. In this way, whatever that can be modeled as a DOM tree could effectively be processed as a template by Thymeleaf.

The Standard Dialect

Thymeleaf is a template engine framework that allows us to define the DOM nodes. The DOM nodes processed in the templates.

An object that applies logic to a DOM node is called **processor**. A set of processors, along with some extra artifacts, is called the **dialect**. The dialect that contains the Thymeleaf's core library is called the **Standard Dialect.**

If we want to define our own processing logic while taking advantage of the library's advanced features, we can define our own dialects. In a template engine, we can configure several dialects at a time.

The Thymeleaf integration packages (thymeleaf-spring3 and thymeleaf-spring4) define a dialect called **SpringStandard Dialect**. The Standard Dialect and SpringStandard are almost the same. But the Standard Dialect has some small changes that make better use of some features in Spring Framework.

For example, use Spring Expression Language instead of Thymeleaf's standard ONGL (Object-Graph Navigation Language).

The Standard Dialect can process template in any mode. But it is perfect for web-oriented template modes (HTML5 and XHTML). It supports and validates the following XHTML specifications:

AD

* XHTML 1.0 Transitional
* XHTML 1.0 Strict
* XHTML 1.0 Frameset
* XHTML 1.1.

AD

Standard Dialect processor is the attribute processor that allows browsers to display HTML5/XHTML template files before being processed. It is because they ignore the additional attributes.

For example, when a JSP file uses tag library, it includes a fragment of code not displayable by a browser like:

1. **<form:inputText** name="student.Name" value="${student.name}" **/>**

The Thymeleaf Standard Dialect allows us to achieve the same functionality with the following code.

1. **<input** type="text" name=" student Name" value="Thomas" th:value="${student.name}" **/>**

The above code also allows us to define a **value** attribute in it (**Thomas**). The value will be displayed when the prototype is opened in the browser. The attribute will be substituted by the value resulting from the evaluation of **${student.name}** during the Thymeleaf processing of the template.

It allows a designer and developer to work on the same template file that reduces the efforts required to transform a static prototype into a working template file. It is known as **Natural Templating.**

AD

Thymeleaf Features

* It works on both web and non-web environments.
* Java template engine for HTML5/ XML/ XHTML.
* Its high-performance parsed template cache reduces I/O to the minimum.
* It can be used as a template engine framework if required.
* It supports several template modes: XML, XHTML, and HTML5.
* It allows developers to extend and create custom dialect.
* It is based on modular features sets called dialects.
* It supports internationalization.

AD

Thymeleaf Implementation

We can implement Thymeleaf template engine by adding **spring-boot-starter-thymeleaf** dependency in our application's pom.xml file. Spring Boot configures template engine to read template file from **/resource/templates.**

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-thymeleaf**</artifactId>**
4. **</dependency>**

AD

Spring Boot Thymeleaf Example

Let's create a Spring Boot application and implement Thymeleaf template.

**Step 1:** Open Spring Initializr [http://start.spring.io](http://start.spring.io/).

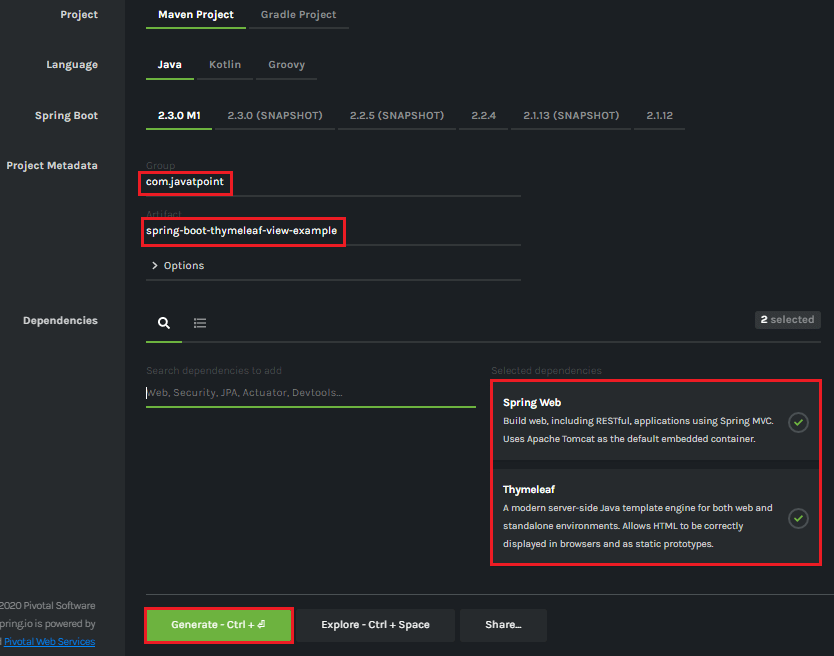
**Step 2:** Select the Spring Boot version **2.3.0.M1.**

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint**.

**Step 3:** Provide the **Artifact** Id. We have provided **spring-boot-thymeleaf-view-example.**

**Step 5:** Add the dependencies **Spring Web** and **Thymeleaf**.

**Step 6:** Click on the **Generate** button. When we click on the Generate button, it wraps the specifications in a **Jar** file and downloads it to the local system.



**Step 7: Extract** the Jar file and paste it into the STS workspace.

**Step 8: Import** the project folder in STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-thymeleaf-view-example -> Finish

It takes some time to import.

**Step 9:** Create a model class in the package **com.javatpoint**. We have created a model class with the name **User.**

In this class, we have defined two variables **name** and **email**and generate**Getters and Setters.**

**User.java**

1. **package** com.javatpoint;
2. **public** **class** User
3. {
4. String name;
5. String email;
6. **public** String getName()
7. {
8. **return** name;
9. }
10. **public** **void** setName(String name)
11. {
12. **this**.name = name;
13. }
14. **public** String getEmail()
15. {
16. **return** email;
17. }
18. **public** **void** setEmail(String email)
19. {
20. **this**.email = email;
21. }
22. }

**Step 10:** Create a controller class. We have created a controller class with the name **DemoController**.

**DemoController.java**

1. package com.javatpoint;
2. import org.springframework.web.bind.annotation.ModelAttribute;
3. import org.springframework.web.bind.annotation.RequestMapping;
4. import org.springframework.web.bind.annotation.RequestMethod;
5. import org.springframework.web.servlet.ModelAndView;
6. import org.springframework.stereotype.Controller;
7. @Controller
8. public class DemoController
9. {
10. @RequestMapping("/")
11. public String index()
12. {
13. return"index";
14. }
15. @RequestMapping(value="/save", method=RequestMethod.POST)
16. public ModelAndView save(@ModelAttribute User user)
17. {
18. ModelAndView modelAndView = new ModelAndView();
19. modelAndView.setViewName("user-data");
20. modelAndView.addObject("user", user);
21. return modelAndView;
22. }
23. }

In the next step, we will create the Thymeleaf templates.

**Step 11:** Inside the **templates** (src/main/resources/templates) folder of the project create a Thymeleaf template with the name **user-data**.

Right-click on the templates folder -> New -> Other -> HTML File -> Next -> Provide the File name -> Finish

Note: Do not forget to implement the following in the template file.

1. **<html** lang="en" xmlns:th="http://www.thymeleaf.org"**>**

**user-data.html**

1. **<html** xmlns:th="https://thymeleaf.org"**>**
2. **<table>**
3. **<tr>**
4. **<td><h4>**User Name: **</h4></td>**
5. **<td><h4** th:text="${user.name}"**></h4></td>**
6. **</tr>**
7. **<tr>**
8. **<td><h4>**Email ID: **</h4></td>**
9. **<td><h4** th:text="${user.email}"**></h4></td>**
10. **</tr>**
11. **</table>**
12. **</html>**

**Step 12:** Similarly, create an **HTML** file in the folder templates. We have created an HTML file with the name **index**.

**index.html**

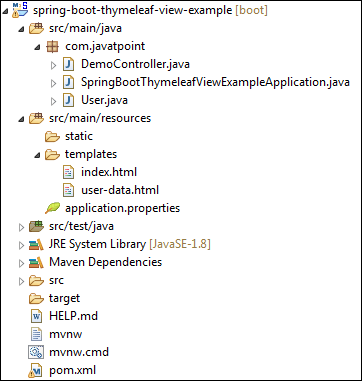
1. **<html** lang="en"**>**
2. **<head>**
3. **<title>**Index Page**</title>**
4. **</head>**
5. **<body>**
6. **<form** action="save" method="post"**>**
7. **<table>**
8. **<tr>**
9. **<td><label** for="user-name"**>**User Name**</label></td>**
10. **<td><input** type="text" name="name"**></input></td>**
11. **</tr>**
12. **<tr>**
13. **<td><label** for="email"**>**Email**</label></td>**
14. **<td><input** type="text" name="email"**></input></td>**
15. **</tr>**
16. **<tr>**
17. **<td></td>**
18. **<td><input** type="submit" value="Submit"**></input></td>**
19. **</tr>**
20. **</table>**
21. **</form>**
22. **</body>**
23. **</html>**

**Step 13:** Open the **application.properties** file and add the following properties in it.

**application.properties**

1. spring.thymeleaf.cache=false
2. spring.thymeleaf.suffix: .html

After creating all the files, folders, and packages, the project directory looks like the following:



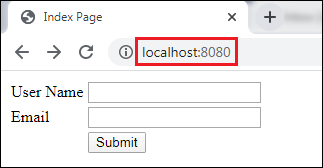
Let's run the application.

**Step 14:** Open the **SpringBootThymeleafViewExampleApplication.java**file and run it as Java Application.

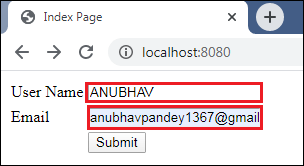
**SpringBootThymeleafViewExampleApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. @SpringBootApplication
5. **public** **class** SpringBootThymeleafViewExampleApplication
6. {
7. **public** **static** **void** main(String[] args)
8. {
9. SpringApplication.run(SpringBootThymeleafViewExampleApplication.**class**, args);
10. }
11. }

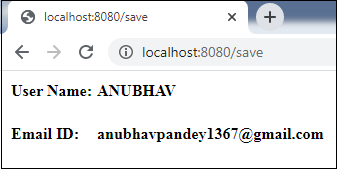
**Step 15:** Now, open the browser and invoke the URL http://localhost:8080. It shows the output, as shown below.



Provide the **User Name** and **Email** and click on the **Submit** button.



After clicking on the **Submit** button, the URL changes to http://localhost:8080/save and shows the show the user-data, as shown below.



In this section, we have discussed the Thymeleaf view. If we want to make the view more attractive, we can add **CSS** and **JS** files in the application. These files must be located under the **src/main/resources/static** folder.