1. Overview

The Spring web framework is built around the MVC (Model-View-Controller) pattern, which makes it easier to separate concerns in an application. This allows for the possibility to use different view technologies, from the well established JSP technology to a variety of template engines.

In this article, we're going to take a look at the main template engines that can be used with Spring, their configuration, and examples of use.

2. Spring View Technologies

Given that concerns in a Spring MVC application are cleanly separated switching from one view technology to another is primarily a matter of configuration.

To render each view type, we need to define a *ViewResolver* bean corresponding to each technology. This means that we can then return the view names from @*Controller* mapping methods in the same way we usually return JSP files.

In the following sections, we're going to go over more traditional technologies like *Java Server Pages*, as well as the main template engines that can be used with Spring: *Thymeleaf*, *Groovy*, *FreeMarker, Jade.*

AD

For each of these, we will go over the configuration necessary both in a standard Spring application and an application built using *Spring Boot*.

3. *Java Server Pages*

JSP is one of the most popular view technologies for Java applications, and it is supported by Spring out-of-the-box. For rendering JSP files, a commonly used type of *ViewResolver* bean is *InternalResourceViewResolver*:

```
@EnableWebMvc
@Configuration
public class ApplicationConfiguration implements WebMvcConfigurer {
    @Bean
    public ViewResolver jspViewResolver() {
        InternalResourceViewResolver bean = new InternalResourceViewResolver();
        bean.setPrefix("/WEB-INF/views/");
        bean.setSuffix(".jsp");
        return bean;
    }
  }
```

Next, we can start creating JSP files in the */WEB-INF/views* location:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
    <head>
        <meta http-equiv="Content-Type" 
          content="text/html; charset=ISO-8859-1">
        <title>User Registration</title>
    </head>
    <body>
        <form:form method="POST" modelAttribute="user">
            <form:label path="email">Email: </form:label>
            <form:input path="email" type="text"/>
            <form:label path="password">Password: </form:label>
            <form:input path="password" type="password" />
            <input type="submit" value="Submit" />
        </form:form>
    </body>
  </html>
```

If we are adding the files to a *Spring Boot* application, then instead of in the *ApplicationConfiguration* class, we can define the following properties in an *application.properties* file:

```
spring.mvc.view.prefix: /WEB-INF/views/
  spring.mvc.view.suffix: .jsp
```

Based on these properties, *Spring Boot* will auto-configure the necessary *ViewResolver*.

4. *Thymeleaf*

Thymeleaf is a Java template engine which can process HTML, XML, text, JavaScript or CSS files. Unlike other template engines, *Thymeleaf* allows using templates as prototypes, meaning they can be viewed as static files.

4.1. Maven Dependencies

To integrate *Thymeleaf* with Spring, we need to add the *thymeleaf* and *thymeleaf-spring4* dependencies:

```
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf</artifactId>
    <version>3.0.11.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring5</artifactId>
    <version>3.0.11.RELEASE</version>
  </dependency>
```

If we have a Spring 4 project, then we need to add *thymeleaf-spring4*.

4.2. Spring Configuration

Next, we need to add the configuration which requires a *SpringTemplateEngine* bean, as well as a *TemplateResolver* bean that specifies the location and type of the view files.

The *SpringResourceTemplateResolver* is integrated with Spring's resource resolution mechanism:

```
@Configuration
@EnableWebMvc
public class ThymeleafConfiguration {

    @Bean
    public SpringTemplateEngine templateEngine() {
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();
        templateEngine.setTemplateResolver(thymeleafTemplateResolver());
        return templateEngine;
    }

    @Bean
    public SpringResourceTemplateResolver thymeleafTemplateResolver() {
        SpringResourceTemplateResolver templateResolver
          = new SpringResourceTemplateResolver();
        templateResolver.setPrefix("/WEB-INF/views/");
        templateResolver.setSuffix(".html");
        templateResolver.setTemplateMode("HTML5");
        return templateResolver;
    }
  }
```

Also, we need a *ViewResolver* bean of type *ThymeleafViewResolver*:

```
@Bean
public ThymeleafViewResolver thymeleafViewResolver() {
    ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
    viewResolver.setTemplateEngine(templateEngine());
    return viewResolver;
  }
```

4.3. *Thymeleaf* Templates

Now we can add an HTML file in the *WEB-INF/views* location:

```
<html>
    <head>
        <meta charset="ISO-8859-1" />
        <title>User Registration</title>
    </head>
    <body>
        <form action="#" th:action="@{/register}"
          th:object="${user}" method="post">
            Email:<input type="text" th:field="*{email}" />
            Password:<input type="password" th:field="*{password}" />
            <input type="submit" value="Submit" />
        </form>
    </body>
  </html>
```

Thymeleaf templates are very similar in syntax to HTML templates.

Some of the features that are available when using *Thymeleaf* in a Spring application are:

- support for defining forms behavior
- binding form inputs to data models
- validation for form inputs
- displaying values from message sources
- rendering template fragments

You can read more about using *Thymeleaf* templates in our article Thymeleaf in Spring MVC.

4.4. *Thymeleaf* in *Spring Boot*

Spring Boot will provide auto-configuration for *Thymeleaf* by adding the *spring-boot-starter-thymeleaf* dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
    <version>2.5.6</version>
  </dependency>
```
No explicit configuration is necessary. By default, HTML files should be placed in the *resources/templates* location.

5. *FreeMarker*

FreeMarker is a Java-based template engine built by the *Apache Software Foundation*. It can be used to generate web pages, but also source code, XML files, configuration files, emails and other text-based formats.

AD

The generation is done based on template files written using the *FreeMarker Template Language*.

5.1. Maven Dependencies

To start using the templates in our project, we need the *freemarker* dependency:

```
<dependency>
    <groupId>org.freemarker</groupId>
    <artifactId>freemarker</artifactId>
    <version>2.3.23</version>
  </dependency>
```

For Spring integration, we also need the *spring-context-support* dependency:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>5.2.8.RELEASE</version>
  </dependency>
```

5.2. Spring Configuration

Integrating *FreeMarker* with Spring MVC requires defining a *FreemarkerConfigurer* bean which specifies the location of the template files:

```
@Configuration
@EnableWebMvc
public class FreemarkerConfiguration {

    @Bean
    public FreeMarkerConfigurer freemarkerConfig() {
        FreeMarkerConfigurer freeMarkerConfigurer = new FreeMarkerConfigurer();
        freeMarkerConfigurer.setTemplateLoaderPath("/WEB-INF/views/");
        return freeMarkerConfigurer;
    }
  }
```

Next, we need to define an appropriate *ViewResolver* bean of type *FreeMarkerViewResolver*:

```
@Bean
public FreeMarkerViewResolver freemarkerViewResolver() {
    FreeMarkerViewResolver resolver = new FreeMarkerViewResolver();
    resolver.setCache(true);
    resolver.setPrefix("");
    resolver.setSuffix(".ftl");
    return resolver;
  }
```

5.3. *FreeMarker* Templates

We can create an HTML template using *FreeMarker* in the *WEB-INF/views* location:

```
<#import "/spring.ftl" as spring/>
<html>
    <head>
        <meta charset="ISO-8859-1" />
        <title>User Registration</title>
    </head>
    <body>
        <form action="register" method="post">
            <@spring.bind path="user" />
            Email: <@spring.formInput "user.email"/>
            Password: <@spring.formPasswordInput "user.password"/>
            <input type="submit" value="Submit" />
        </form>
    </body>
  </html>
```

In the example above, we have imported a set of macros defined by Spring for working with forms in *FreeMarker*, including binding form inputs to data models.

Also, the *FreeMarker Template Language* contains a large number of tags, directives, and expressions for working with collections, flow control structures, logical operators, formatting and parsing strings, numbers and many more features.

5.4. *FreeMarker* in *Spring Boot*

In a *Spring Boot* application, we can simplify the needed configuration by using the *spring-boot-starter-freemarker* dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-freemarker</artifactId>
    <version>2.5.6</version>
  </dependency>
```

This starter adds the necessary auto-configuration. All we need to do is start placing our template files in the *resources/templates* folder.

6. *Groovy*

Spring MVC views can also be generated using the Groovy Markup Template Engine. This engine is based on a builder syntax and can be used for generating any text format.

6.1. Maven Dependencies

The *groovy-templates* dependency needs to be added to our *pom.xml*:

```
<dependency>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>groovy-templates</artifactId>
    <version>2.4.12</version>
  </dependency>
```

6.2. Spring Configuration

The integration of the *Markup Template Engine* with Spring MVC requires defining a *GroovyMarkupConfigurer* bean and a *ViewResolver* of type *GroovyMarkupViewResolver*:

```
@Configuration
@EnableWebMvc
public class GroovyConfiguration {

    @Bean
    public GroovyMarkupConfigurer groovyMarkupConfigurer() {
        GroovyMarkupConfigurer configurer = new GroovyMarkupConfigurer();
        configurer.setResourceLoaderPath("/WEB-INF/views/");
        return configurer;
    }

    @Bean
    public GroovyMarkupViewResolver thymeleafViewResolver() {
        GroovyMarkupViewResolver viewResolver = new GroovyMarkupViewResolver();
        viewResolver.setSuffix(".tpl");
        return viewResolver;
    }
}
```

6.3. *Groovy Markup* Templates

Templates are written in the Groovy language and have several characteristics:

- they are compiled into bytecode
- they contain support for fragments and layouts
- they provide support for internationalization
- the rendering is fast

Let's create a Groovy template for our “User Registration” form, which includes data bindings:

```
yieldUnescaped '<!DOCTYPE html>'
html(lang:'en') {
    head {
        meta('http-equiv':'"Content-Type" ' +
          'content="text/html; charset=utf-8"')
        title('User Registration')
    }
    body {
        form (id:'userForm', action:'register', method:'post') {
            label (for:'email', 'Email')
            input (name:'email', type:'text', value:user.email?:'')
            label (for:'password', 'Password')
            input (name:'password', type:'password', value:user.password?:'')
            div (class:'form-actions') {
                input (type:'submit', value:'Submit')
            }
        }
    }
  }
```

6.4. *Groovy Template Engine* in *Spring Boot*

Spring Boot contains auto-configuration for the *Groovy Template Engine*, which is added by including the *spring-boot-starter-groovy-templates* dependency:

AD

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-groovy-templates</artifactId>
    <version>2.5.6</version>
  </dependency>
```

The default location for the templates is */resources/templates*.

7. *Jade4j*

Jade4j is the Java implementation of the *Pug* template engine (originally known as *Jade*) for Javascript. *Jade4j* templates can be used for generating HTML files.

7.1. Maven Dependencies

For Spring integration, we need the spring-jade4j dependency:

```xml
<dependency>
    <groupId>de.neuland-bfi</groupId>
    <artifactId>spring-jade4j</artifactId>
    <version>1.2.5</version>
  </dependency>
```

7.2. Spring Configuration

To use *Jade4j* with Spring, we have to define a *SpringTemplateLoader* bean that configures the location of the templates, as well as a *JadeConfiguration* bean:

```
@Configuration
@EnableWebMvc
public class JadeTemplateConfiguration {

    @Bean
    public SpringTemplateLoader templateLoader() {
        SpringTemplateLoader templateLoader
          = new SpringTemplateLoader();
        templateLoader.setBasePath("/WEB-INF/views/");
        templateLoader.setSuffix(".jade");
        return templateLoader;
    }

    @Bean
    public JadeConfiguration jadeConfiguration() {
        JadeConfiguration configuration
          = new JadeConfiguration();
        configuration.setCaching(false);
        configuration.setTemplateLoader(templateLoader());
        return configuration;
    }
  }
```

Next, we need the usual *ViewResolver* bean, in this case of type *JadeViewResolver*:

```
@Bean
public ViewResolver viewResolver() {
    JadeViewResolver viewResolver = new JadeViewResolver();
    viewResolver.setConfiguration(jadeConfiguration());
    return viewResolver;
  }
```

7.3. *Jade4j* Templates

Jade4j templates are characterized by an easy-to-use whitespace-sensitive syntax:

```
doctype html
html
  head
    title User Registration
  body
    form(action="register" method="post" )
      label(for="email") Email:
      input(type="text" name="email")
      label(for="password") Password:
      input(type="password" name="password")
        input(type="submit" value="Submit")
```

The project also provides a very useful interactive documentation, where you can view the output of your template as you write it.

Spring Boot does not provide a *Jade4j* starter, so in a *Boot* project, we would have to add the same Spring configuration as defined above.

8. Other Template Engines

Besides the template engines described so far, there are quite a few more available which may be used.

Let's review some of them briefly.

Velocity is an older template engine, which is very complex but has the disadvantage that Spring has deprecated its use since version 4.3 and removed completely in Spring 5.0.1.

JMustache is a template engine which can be easily integrated into a Spring Boot application by using the *spring-boot-starter-mustache* dependency.

Pebble contains support for Spring and *Spring Boot* within its libraries.

Other templating libraries such as *Handlebars* or *React*, running on top of a *JSR-223* script engine such as *Nashorn,* can also be used.