SimpleTemplate

A engine de templates do Futurepages

Uma introdução e API

Por Thiago Rabelo

Sumário

1 - Introdução

O Futurepages 2 conta com uma engine para renderização de templates. Esta engine foi batizada com o nome **SimpleTemplate.** Sua estrutura encontra-se no pacote ***org.futurepages.util.template.simpletemplate***. Esta engine suporta tags baseadas em comentários HTML e expressões semelhantes às ELs do JSP.

2 - Usando as tags

Por exemplo vajamos o trecho abaixo:

```
<!--if [teste]-->
      ${teste} passou
<!--endif-->
<!--if [!teste]-->
      ${teste} NÃO passou
<!--endif-->
```

Toda tag deve ser termina da seguinte forma:

```
<!--end[NOME_DA_TAG]-->
```

Por exemplo:

```
<!--if ...-->...<!--endif-->
<!--forEach ...-->...<!--endforEach-->
```

Para que a tag seja considerada realmente um comentário HTML, basta separar a abertura da tag do seu nome:

```
<!--if [teste]-->   // isto é uma tag
<!-- if [teste]--> // isto é um comentário HTML
```

Atualmente a engine conta com 4 tags básicas, sendo elas:

if – Uma tag que avalia o seu corpo se recebe como parâmetro um valor diferente de `false`, `null` ou `0`. Seu parâmetro pode ou não ser envolvido entre colchetes "[]".

forEach – Itera sobre o seu corpo de acordo com a lista passado como parâmetro. Este parâmetro pode tanto ser uma lista que está no contexto, quando pode ser uma sintaxe que gera uma lista automaticamente. Por exemplo:

```
<!--forEach 1..10:pass|var-->...
```

Onde em `1..10:pass|var`

- **1** – é o valor de início da lista.
- **10** – é o valor onde terminam as iterações. Por exemplo, se for `1..10`, a lista começa em 1 e termina em 9.
- **pass** – é o incremento. Por exemplo, se for 1 para lista `1..10`, será: 1, 2, 3, 4, 5, 6, 7, 8, 9 e 10. E se for 2 para a lista `1..10`, será: 1, 3, 5, 7

e 9.

- **var** – é o nome da variável. A cada passo da iteração na lista, o valor corrente é colocado numa variável com o nome "var". Sendo assim ela se torna acessível através da expressão EL: ${var}.

Ex:

```
<!--forEach 1..10:pass|var-->
      o valor atual é ${var}
<!--endforEach-->
```

```
<!--forEach lista|var|contador-->...
```

Onde em lista|var|contador

- **lista** – é a lista que foi passada para o contexto.
- **var** – é o nome da variável. A cada passo da iteração na lista, o valor corrente é colocado numa variável com o nome "var". Sendo assim ela se torna acessível através da expressão EL: ${var}.
- **contador** – é o nome da variável a qual é atribuído um contador. A cada iteração seu calor é acrescentado de 1, sendo que é inicializado com 0.

Ex:

```
<!--forEach lista|var|contador-->
      iteração de número ${contador} – o valor atual é ${var}
<!--endforEach-->
```

set – Adiciona ao contexto uma nova variável. Recebe dois parâmetros separados por um |, sendo o segundo não obrigatório. O primeiro parâmetro é a nova variável onde o será armazenado um valor. O segundo é o valor que se quer armazenar no primeiro parâmetro. Caso o segundo parâmetro não seja especificado, o conteúdo do corpo da tag será usado como o valor a ser armazenado na nova variável. Ex:

```
<!--set new_var | 'hello, world'--><!--endset-->
// neste caso new_var receberá a string 'hello, world'.
```

```
<!--set new_var -->Olá, meu nome é ${nome}<!--endset-->
// já neste caso, a tag set avaliará o seu corpo (Olá, meu nome é ${nome}) e a string gerada será armazenada em new_var.
```

valueFormatter – Formata uma determinada entrada. Usa os formatadores padrões do Futurepages para formatar um determinado valor. Recebe dois parâmetros obrigatórios, separados por um |, sendo o primeiro o valor a qual se quer formatar. E o segundo, uma string contendo o nome do formatador a ser usado. Ex:

```
<!--valueFormatter alguma_data | 'dateTime'--><!--endvalueFormatter-->
<!--valueFormatter algumValor | algumFormatador--><!--endvalueFormatter-->
```

3 – As expressões

Já as expressões, que podem ser parâmetros da tags ou serem usadas dentro do operador ${}, são semelhantes às Expression Language, do JSP. As principais diferenças, são que as expressões usadas na engine de template do Futurepages 2 não suportam, AINDA, o uso de funções, o operador "[]" e o operador ternário

“?:”.

Os operadores disponíveis (na ordem de precedência de cima para baixo, da esquerda para direita) são:

- **(**, **)** (parênteses esquerdo e direito)
- **!** (negação), **+** (positivo), **-** (negativo)
- ****** (potência)
- ***** (multiplicação), **/** (divisão), **%** (resto)
- **+** (soma), **-** (subtração)
- **<** (menor que), **<=** (menor ou igual que), **>** (maior que), **>=** (maior ou igual que)
- **==** (comparação/igual), **!=** (negação da comparação/diferença)
- **^** (ou exclusivo)
- **&&** (e), **||** (ou)

O comportamento dos operadores usados nessa engine tem inspirações em linguagens como JavaScript e Python. Pois, como só é possível saber os tipos que os operadores vão manipular apenas durante o processo de avaliação, eles tem que lidar com os mais variados tipo.

A exemplo disto vamos usar o operador **+** e o operador **&&**. O operado **+** trata seus dois parâmetros como pertencendo a um só tipo. Se ambos forem inteiros (**int**, **long**, **short**, **byte**), seus valores serão convertidos para **long** e o resultado será um valor do tipo **long**. Caso algum dos parâmetros seja ponto flutuante (**float** ou **double**), os dois serão tratados como **double** e o resultado será do tipo **double**.

Se o operador **+** receber um valor de algum tipo que não seja numérico, ele retorna um valor NaN (Not an Number).

Alguns exemplos:

```
${a + b} => considerando que a = 2 e b = 3 => 2 + 3 = 5
${1 + 2} => 3
${a + 5} => considerando que a = 2 => 2 + 5 = 7
${1 + 'Eu sou uma string'} => NaN
```

O operador **&&**, como os demais operadores lógicos (**!**, **^**, **&&** e **||**) considera como verdadeiro qualquer valor diferente de `false`, `null` ou `0`:

```
${true && false} => false
${true && a} => considerando que a = true => true && true = true
${true && 0} => false
${true && null} => false
${true && 0} => false
${true && 1} => true
${true && 'Eu sou uma string'} => true
```

O operador “${ }” aceita expressões mais complexas, como por exemplo: `${a*x**2 + b*x + c > 25.0}`. Que neste exemplo é que uma equação do segundo grau ($ax^2 + bx + c$) testando se seu resultado é maior que 25,0.

Deve ser observado que tais expressões também podem ser usadas como parâmetros de tags. Veja no caso logo abaixo o uso na tag `if`:

```
<!--if [a*x**2 + b*x + c > 25.0]-->
      a*x**2 + b*x + c (${a*x**2 + b*x + c}) é maior que 25.0
<!--endif-->
<!--if [!(a*x**2 + b*x + c > 25.0)]-->
      a*x**2 + b*x + c (${a*x**2 + b*x + c}) não é maior que 25.0
<!--endif-->
```

4 – Como usar

Até agora vimos como é a sintaxe da linguagem. Mas agora veremos como usá-la, ver os erros que podem ser gerados e aprender a estender suas funcionalidades criando novas tags.

Para utilizar a engine, temos que passar por duas etapas: a compilação e a avaliação. Para compilar, basta chamar o método ***compile*** da classe ***TemplateParser***, passando como parâmetro a string que contém o template. Este método retorna um objeto do tipo **TemplateBlockBase** que representa a estrutura do template.

A classe **TemplateParse** encontra-se no pacote ***org.futurepages.util.template.simpletemplate.template***. Já a classe **TemplateBlockBase**, está no pacote ***org.futurepages.util.template.simpletemplate.template***.

Abaixo temos um exemplo:

```
TemplateBlockBase template = TemplateParser.compile(
      "<!--forEach inicio..fim:1|var-->" +
            "O número é ${var} e o dobro é ${2 * var}\n" +
      "<!--endforEach-->"
);
```

Dado que este é o processo de compilação, resta-nos ver como obter a string que resulta da avaliação do template. Para isto, devemos chamar o método ***eval*** do template, passando como parâmetro um **Map<String, Object>** com os valores que deverão ser usados nos templates. Veja logo a seguir:

```
Map<String, Object> valores = new HashMap<String, Object>();
valores.put("inicio", 1);
valores.put("fim", 10);

String output = template.eval(valores);
System.out.println(output);
```

A saída para o exemplo anterior seria:

```
O número é 1 e o dobro é 2
O número é 2 e o dobro é 4
O número é 3 e o dobro é 6
O número é 4 e o dobro é 8
O número é 5 e o dobro é 10
O número é 6 e o dobro é 12
O número é 7 e o dobro é 14
O número é 8 e o dobro é 16
O número é 9 e o dobro é 18
```

O método **TemplateParse.compile**, pode lançar uma exceção chamada **TemplateException**. Que contém as informações sobre o erro e sua localização no template. Por exemplo, vamos pegar no exemplo acima e, propositalmente, causar um erro. No lugar de ${2 * var}, colocaremos ${2 * }. O operador * espera dois parâmetros, um na esquerda e um na direita, mas neste caso "esquecemos" de colocar o valor da direita.

```
TemplateBlockBase template = TemplateParser.compile(
	"<!--forEach inicio..fim:1|var-->" +
		"O número é ${var} e o dobro é ${2 * }\n" +
	"<!--endforEach-->"
);
```

Sendo assim, o método ***compile*** lança uma execeção do tipo **TemplateException**. Nós podemos usar esta exceção para identificar onde está o problema. Para isto basta chamar o método **getMessage** da exceção lançada:

```
try {
	TemplateBlockBase template = ... // omitido
	... // omitido
} catch (TemplateException ex) {
	System.out.println(ex.getMessage()); // imprimindo mensagem de erro
}
```

Neste caso a saída será:

```
Expected right expression after 2º token (*):1
2 *
  ^
```

O :1 indica a linha em que o erro ocorre.

A exceção também pode indicar erro de aninhamento. Caso seja esquecido o fechamento da tag, como se segue a baixo:

```
try {
	TemplateBlockBase template = TemplateParser.compile(
		"<!--forEach inicio..fim:1|var-->" +
			"O número é ${var} e o dobro é ${2 * }\n"
	);
	... // omitido
} catch (TemplateException ex) {
	System.out.println(ex.getMessage()); // imprimindo mensagem de erro
}
```

Já, neste caso a mensagem de erro seria:

```
Template nesting error: forEach:0
```

Onde, forEarch:0 indica que a tag <!--forEach ...--> da linha 0 não foi fechada (<!--endforEach-->).

5 – Criando tags customizadas

A engine também suporta extensões. Permitindo que novas tags sejam criadas. Para isto, a nova tag deve ser uma classe que herda de **TemplateTag** que está no pacote

org.futurepages.util.template.simpletemplate.template.builtin.tags.

Ao estender ***TemplateTag***, devemos sobrescrever estes 4 métodos:

- `public abstract Exp` **`evalExpression`**`(String expression) throws ExpectedOperator, ExpectedExpression, BadExpression, Unexpected;`
- `public abstract TemplateTag` **`getNewInstance`**`();`
- `public abstract boolean` **`hasOwnContext`**`();`
- `public abstract int` **`doBody`**`(AbstractTemplateBlock block, ContextTemplateTag context, TemplateWritter sb);`

evalExpression – Recebe uma como parâmetro a string que repesenta os parâmetros da tag. Retorna um objeto que implementa a interface **Exp**. Por exemplo, em `<!--if [a + b]-->`, a expressão `a + b` é convertida em um objeto que implementa a interface **Exp**. Esta interface encontra-se no pacote ***org.futurepages.util.template.simpletemplate.expressions.tree***. A própria classe **TemplateTag** possui um método estático auxiliar chamado **defaultEvalExpression** que pode ser usado para avaliar a string que representa a expressão e retornar um objeto do tipo **Exp**. Caso a tag precise de mais de um parâmetro, a solução é criar uma classe que implemente **Exp** ou **TagParams** em ***org.futurepages.util.template.simpletemplate.template.builtin.customtagparams*** e que armazene tais parâmetros numa instância desta classe. E então, esta instância retornada por **evalExpression**.

getNewInstance – Define como serão criadas novas instâncias da tag. Retorna uma instância de **TemplateTag** que é o objeto que representa a tag. Pode ser usado para retornar o operador de auto referência `this`, fazendo com que seja uma instância única para todos os templates. Sendo assim o mesmo objeto será usado em todos os templates gerados pelo processo de compilação.

hasOwnContext – Retorna um booleano que informa se a tag em questão terá o seu próprio contexto (escopo). Sendo assim novas variáveis que sejam criadas dentro do corpo da tag não estarão disponíveis fora dele. A tag `forEach`, por exemplo tem seu próprio contexto. Já a tag `if`, não.

doBody – Contém a lógica que indica se o corpo da tag será ou não avaliado. Existem as constantes `EVAL_BODY` e `SKIP_BODY` que devem ser usados como valor de retorno. Sendo que `EVAL_BODY` indica que o corpo da tag deve ser avaliado e `SKIP_BODY`, que o corpo da tag não deve. Qualquer outro valor será ignorado e o corpo da tag não será avaliado.

O construtor desta tag deve chamar o construtor da classe base **TemplateTag**, passando uma string que será o nome da tag que será usado no template.

Existe o método **evalBody** que o método responsável por avaliar o corpo da tag. Este método pode ser chamado de dentro do método **doBody**, fazendo com que a avaliação seja imediata, e não apenas quando **doBody** retornar `EVAL_BODY`.

A tag `forEach` faz o uso deste artifício, pois como ela precisa avaliar seu corpo repetidas vezes, uma a cada iteração, ela **evalBody** chama diretamente dentro de **doBody**. Mas ao final retorna `SKIP_BODY`, pois como as avaliações já foram feitas, não há mais necessidade de outra.

Ao terminarmos de implementar a nova tag, precisamos registrá-la, para que a engine saiba que esta nova funcionalidade existe. Isto se dá através do método **addTag** da classe **TemplateTagInitializer**,localizado no pacote

org.futurepages.util.template.simpletemplate.template.builtin.tags.

Para exemplificar a criação de uma nova tag, vamos criar uma que receba dois parâmetros separados por |, sendo que o primeiro é um array ou uma lista e o segundo, uma string que usaremos para juntar esta lista e formar uma única string (um join). Caso a lista ou array estejam vazios ou nulo, a tag deverá imprimir o corpo da tag. A sintaxe ficará assim:

```
<!--join list | joinner--><!--endjoin-->
```

Sendo assim, vamos começar definindo a classe que será a nossa nova tag.

```
public class JoinTemplateTag extends TemplateTag {

	// Definindo um joinner padrão, caso um não seja
	// passado como parâmetro para tag.
	private static final String DEFAULT_JOINNER = ",";

	public JoinTemplateTag() {
		// Chama a super classe, passando como
		// parâmetro uma string que diz como será
		// o nome da tag.
		super("join");
	}

	// Se esta classe tivesse um único parâmetro,
	// bastaria: return defaultEvalExpression(expression);
	@Override
	public Exp evalExpression(String expression)
	throws ExpectedOperator, ExpectedExpression, BadExpression, Unexpected {
		// Método auxiliar já pronto para separar parâmetros os parâmetros
		// que estão entre |.
		String [] attrs = splitParams(expression);

		if (attrs != null) {
			if (attrs.length >= 2) { // Foram passados 2 parâmetros
				// Chamadas ao método auxiliar para
				// avaliar os parâmetros.
				Exp p1 = defaultEvalExpression(attrs[0]);
				Exp p2 = defaultEvalExpression(attrs[1]);

				return new JoinParams(p1, p2);

			} if (attrs.length == 1) { // Foi passado apenas nenhum parâmetro
				// Chamada ao método auxiliar para avaliar o parâmetro.
				Exp p1 = defaultEvalExpression(attrs[0]);

				return new JoinParams(p1, null);
			}
		}

		// Nenhum parâmetro foi passado.
		return new JoinParams(null, null);
	}

	@Override
	public TemplateTag getNewInstance() {
		// Não preciso criar uma nova instância
		// para cada template. Por isso retorno
		// "eu mesmo".
		return this;
	}
```

```
@Override
public boolean hasOwnContext() {
	// Não precisamos que hava um contexto próprio.
	return false;
}

@Override
public int doBody(AbstractTemplateBlock block, ContextTemplateTag context,
TemplateWritter sb) {
	// TemplateBlock é o objeto que encapsula a
	// tag e seu parâmetro. Neste caso uma instancia
	// de JoinTemplateTag e uma instância de JoinParams
	// respectivamente.
	TemplateBlock actualBlock = (TemplateBlock) block;

	// Recebendo os parâmetros.
	JoinParams joinParams = (JoinParams)actualBlock.getParams();

	Object [] params = (Object [])joinParams.eval(context);

	// definindo joinner;
	String joinner = params[1] != null
		? params[1].toString()
		: DEFAULT_JOINNER;
	String connector = "";
	StringBuilder _sb;

	switch (isOk(params)) {
		case 1: // caso lista
			_sb = new StringBuilder();
			List l = (List)params[0];

			// Fazendo o join na lista
			for (Object o : l) {
				_sb.append(connector).append(o.toString());
				connector = joinner;
			}

			// Colocando o resultado do join no output
			sb.append(_sb.toString());

			// Tudo ok! Não é necessário avaliar o corpo da tag.
			return SKIP_BODY;

		case 2: // caso array
			_sb = new StringBuilder();
			Object [] arr = (Object[])params[0];

			// Fazendo o join no array
			for (Object o : arr) {
				_sb.append(connector).append(o.toString());
				connector = joinner;
			}

			// Colocando o resultado do join no output
			sb.append(_sb.toString());

			// Tudo ok! Não é necessário avaliar o corpo da tag.
			return SKIP_BODY;

		default: // Senão, avalia o corpo da tag
			return EVAL_BODY;
	}
}
```

```
	// Aqui vemos se os p[0] é diferente e é
	// uma instancia de List ou é um array.
	private int isOk(Object []p) {
		if (p[0] != null) {
			if (p[0] instanceof List) {
				return 1;
			} else if (p[0].getClass().isArray()) {
				return 2;
			} else {
				return 0;
			}
		}

		return 0;
	}
}
```

E agora a classe que guarda os parâmetros da tag:

```
public class JoinParams implements TagParams {

	private Exp list;
	private Exp joinner;

	public JoinParams(Exp list, Exp joinner) {
		this.list = list;
		this.joinner = joinner;
	}

	@Override
	public Object eval(ContextTemplateTag context) {
		Object l = list != null ? list.eval(context) : null;
		Object j = joinner != null ? joinner.eval(context) : null;

		return (new Object[] {l, j});
	}

	@Override
	public void toString(StringBuilder sb) {
		// podemos deixar vazio
	}
}
```

Gere um arquivo texto chamado **teste_nova_tag.txt** que contenha o seguinte conteúdo:

```
<!--join lista | '-'-->(VAZIO)<!--endjoin-->
```

E agora vamos criar uma classe com um método main:

```
public class TestesNovaTag {

	public static void main(String[] args) throws Exception {
		// Registrando a tag.
		TemplateTagInitializer.instance().addTag(new JoinTemplateTag());

		// Pegando o conteúdo do arquivo.
		String templateString = FileUtil.getStringContent(
		"path/para/o/arquivo/teste_nova_tag.txt"
		);
		TemplateBlockBase template = TemplateParser.compile(templateString);
```

```
            ArrayList<String> list = new ArrayList<String>();
            list.add("mercúrio");
            list.add("vênus");
            list.add("terra");
            list.add("marte");
            list.add("júpter");
            list.add("saturno");
            list.add("urano");
            list.add("netuno");

            //String [] arr = {"mercúrio", "vênus", "terra", "marte", "júpiter",
            "saturno", "urano", "netuno"};

            HashMap<String, Object> mapa = new HashMap<String, Object>();
            // Passando list como sendo a variável "lista"
            mapa.put("lista", list);

            String output = template.eval(mapa);

            System.out.println(output);
        }
}
```

Se você rodar este teste, verá que a saída é:

```
mercúrio-vênus-terra-marte-júpter-saturno-urano-netuno
```

Experimente trocar a lista por um array ou mesmo omitir o segundo parâmetro da tag e veja como fica a saída.

6 – Para o futuro

Novas funcionalidades devem ser incluídas aos poucos, mas já há planos de adicionar novas funcionalidades, que são:

- Todos as tags terem a possibilidade de ter uma tag <!--else--> ou uma tag <!--elif [condicao]-->. Permitindo que fosse possível a sintaxe:

```
<!--if [condicao]-->
      ...
<!--else-->
      ...
<!--endif-->

<!--forEach 1..10:pass|var-->
      ...
<!--else-->
      ...
<!--endforEach-->
```

- A avaliação de funções em expressões, como em ${empty(teste)}.

- Suporte aos operadores “[]” e “?:”, como em ${map['key'] == array[1]} ou em ${teste ? 'sim' : 'não'}.

- E é claro, sempre que possível, buscar formas de melhorar a performance e corrigir bugs.