How To Build Template Driven Java Websites with FreeMarker and RESTEasy

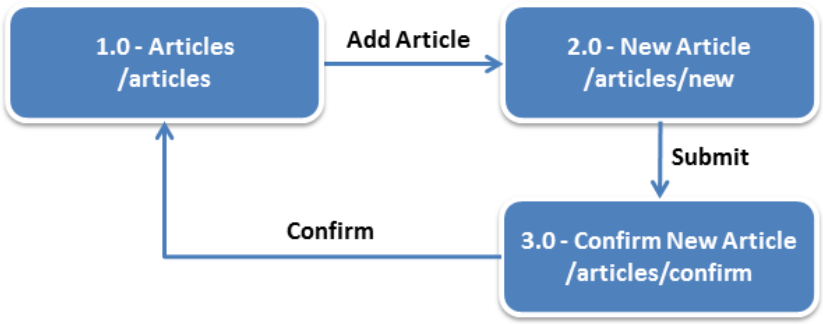
By Dele Taylor

Last week I wrote about why you should switch to a templating engine from Java Server Pages. This week I'll take it a step further and show you how to use FreeMarker, along with Bootstrap and RESTEasy, to create truly template driven websites. By adding a thin layer on top of FreeMarker, you'll be able to theme your Java web apps like anything built on top of a CMS like WordPress or Drupal.

The Final Product

The example I'll use here is a simple article submission website. Something like what you'd see on Javalobby or TheServerSide to add news and announcements. In this case, the app does only three things:

1. Displays submitted articles
2. Accepts new article submissions
3. Accepts confirmation for new article submissions



Bootstrap Theme

This web app uses Twitter's Bootstrap front-end framework for it's look and feel (theme). Since the app is template driven, you can swap out the four template files to completely re-theme it with another style.

Templates Files

All of the the app's HTML output — forms, tables, links — are stored in four FreeMarker template files. These files can be located anywhere FreeMarker can access (see FreeMarker Template Loaders), but "WEB-INF/content" is a safe default to start with. When being used, the files will be referenced without the "WEB-INF/content" prefix, as "site.html" and "articles/form-article.html". Here's an overview of the files, you'll see how they are used in the REST resources later.

#	Template	Description
1	WEB-INF/content/site.html	Overall website template. Includes references to JS and CSS files.
2	WEB-INF/content/articles/grid-articles.html	Article list table.
3	WEB-INF/content/articles/form-article.html	New article entry form.
4	WEB-INF/content/articles/form-article-confirmation.html	New article confirmation form.

site.html

WEB-INF/content/site.html XHTML

```
33    <div class="container">
34
35      <div class="starter-template">
36        ${body}
37      </div>
38
39    </div><!-- /.container -->
```

This file houses the bulk of the HTML/FreeMarker code. It contains a body region — ${body} — where each page's content (like grid-articles.html) will be injected. (Click the *Expand Code* icon in the toolbar to see the full source code.)

grid-articles.html

WEB-INF/content/articles/grid-articles.html XHTML

```
18      <#list articles as article>
19      <tr>
20        <td>${article_index + 1}</td>
21        <td>${article.title}</td>
22        <td>${article.name}</td>
23        <td>${article.email}</td>
24        <td>${article.url}</td>
25      </tr>
26      </#list>
```

The grid template draws the article table using a combination of HTML and FreeMarker code. It expects a collection named "article" and uses the FreeMarker list directive to create a table row for each element it finds.

The actual "articles" collection can be Java beans, JSON objects, or any other kind of sequence or collection FreeMarker understands.

form-article.html

WEB-INF/content/articles/form-article.html XHTML

```
1  <h2>Add New Article</h2>
2  <form method="post" action="${addNewArticleFormAction!}">
3    <div class="form-group">
4      <input type="url" class="form-control" id="url" name="url" placeholder="Enter URL
5    </div>
6    <div class="form-group">
7      <input type="text" class="form-control" id="title" name="title" placeholder="Ente
8    </div>
9    <div class="form-group">
10     <input type="email" class="form-control" id="email" name="email" placeholder="Ent
11   </div>
12   <div class="form-group">
13     <input type="text" class="form-control" id="name" name="name" placeholder="Enter
14   </div>
15   <div class="buttons">
16     <button type="submit" class="btn btn-default btn-primary">Submit</button>
17     <a type="submit" class="btn" href="..">Cancel</a>
18   </div>
19 </form>
```

The new article template renders the input form for new entries. On submit it does a POST back to it's GET URL if no *addNewArticleFormAction* value is specified. It also looks for, and tries to use, the article's properties (like article.url) if provided to the template.

form-article-confirmation.html

WEB-INF/content/articles/form-article-confirmation.html XHTML

```
<form method="post" action=""  class="form-horizontal">
  <h2>Confirm Article Submission?</h2>
  <div class="form-group">
    <label class="col-sm-2 control-label">URL</label>
    <div class="col-sm-10">
      <p class="form-control-static">${article.url}</p>
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">Title</label>
    <div class="col-sm-10">
      <p class="form-control-static">${article.title}</p>
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">Email</label>
    <div class="col-sm-10">
      <p class="form-control-static">${article.email}</p>
    </div>
  </div>
  <div class="form-group">
    <label class="col-sm-2 control-label">Name</label>
    <div class="col-sm-10">
      <p class="form-control-static">${article.name}</p>
    </div>
  </div>
  <div class="buttons">
    <button type="submit" class="btn btn-default btn-primary">Confirm</button>
    <a type="submit" class="btn" href="../..">Cancel</a>
  </div>
</form>
```

The confirmation template uses the supplied article to display it's values, then POSTs back to it's GET location on submit.

RESTful Resources

All page requests and form posts are handled by the RootResource class. RootResource can be though of as the controller in the model-view-controller pattern. It's implemented as a JAX-RS resource running on JBoss' RESTEasy framework.

POST requests always redirect the browser to a GET method on completion, while GET requests typically return content — complete HTML pages in this case. The interesting methods here are the GET handlers. They use the content API to interact with FreeMarker and link the model to the various template views.

RootResource.java Java

```
    @GET
    @Path("/")
    public Response getHome() throws Throwable {
        return Response.seeOther(new URI("/articles/")).build();
    }

    @GET
    @Path("/articles")
    public Content getArticles() throws Throwable {
        Content content = new Content("site.html");
        content.add("body", new Content("articles/grid-articles.html"));
        content.add("articles", articles);
```

```
65          return content;
66      }
67
68      @GET
69      @Path("/articles/new")
70      public Content getNewArticle() throws Throwable {
71          Content content = new Content("site.html");
72          content.add("body", new Content("articles/form-article.html"));
73          return content;
74      }
75
76      @POST
77      @Path("/articles/new")
78      public Response postNewArticle(@Form Article article) throws Throwable {
79          String articleId = UUID.randomUUID().toString();
80          pendingArticles.put(articleId, article);
81
82          return Response.seeOther(new URI("/articles/confirm/" + articleId + "/")).bu
83      }
84
85      @GET
86      @Path("/articles/confirm/{articleId}")
87      public Content getConfirmNewArticle(@PathParam("articleId") String articleId) th
88          Article article = pendingArticles.get(articleId);
89
90          Content content = new Content("site.html");
91
92          content.add("title", "Confirm Article Submission");
93          content.add("body", new Content("articles/form-article-confirmation.html"));
94          content.add("body", new Content("articles/form-article.html"));
95          content.add("addNewArticleFormAction", "../../new/");
96          content.add("article", article);
97
98          return content;
99      }
100
101     @POST
102     @Path("/articles/confirm/{articleId}")
103     public Response postConfirmNewArticle(@PathParam("articleId") String articleId)
104         Article article = pendingArticles.remove(articleId);
105         articles.add(article);
106         return Response.seeOther(new URI("/articles/")).build();
107     }
108
```

Content API

The Content class is the central piece in that thin layer on top of FreeMarker I mentioned earlier. It allows multiple templates and/or values to be added to the same named slot (like ${body}) and it allows nested templates to find models added to their parent. It also handles all the boilerplate FreeMarker code. Its possibly the only class you'll need to use templates in your own servlets or RESTful resources.

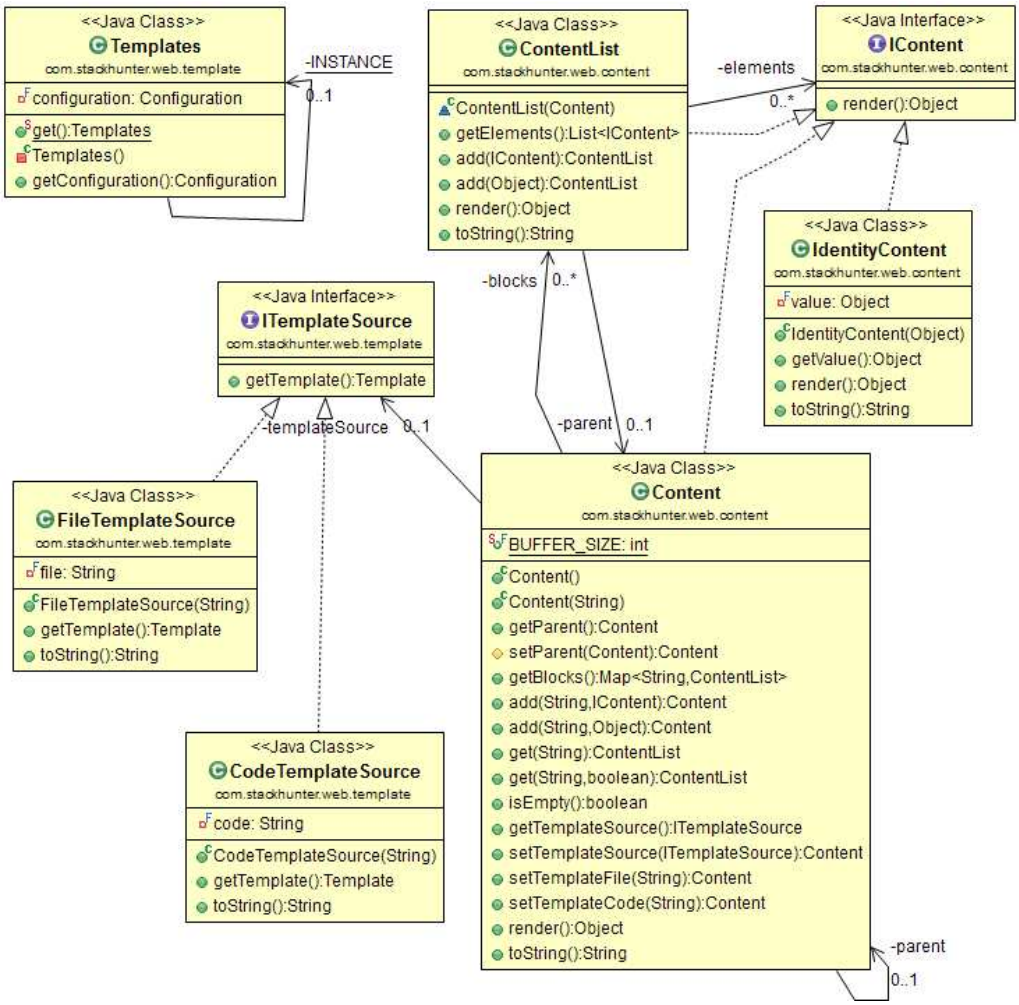
RootResource.getConfirmNewArticle(String) Java

```
1       @GET
2       @Path("/articles/confirm/{articleId}")
3       public Content getConfirmNewArticle(@PathParam("articleId") String articleId) thr
4           Article article = pendingArticles.get(articleId);
5
6           Content content = new Content("site.html");
7
8           content.add("title", "Confirm Article Submission");
9           content.add("body", new Content("articles/form-article-confirmation.html"));
10          content.add("body", new Content("articles/form-article.html"));
```

```
11          content.add("addNewArticleFormAction", "../../new/");
12          content.add("article", article);
13
14          return content;
15      }
```

Let's breakdown the *getConfirmNewArticle* method to get a better idea of how things work.

1. The outer website template is created (line 6). This template will hold data and other templates. It will evaluate itself and any nested templates in the right order sometime after line 14. You can force a manual evaluation by calling it's *toString* method.
2. Two templates are nested into the outer template inside it's *body* region (lines 9 & 10).
3. The pending article is added to the outer template as a variable named *article (line 12)*.
4. The outer template is returned, evaluated, and it's string contents sent to the browser (automatically, after line 14).



There are several other classes in the API. They are used internally and exposed to keep the API flexible for special cases. The *ITemplateSource*, for example, is a factory for creating FreeMarker Templates. In most cases calling the Content constructor with the file path argument will suffice.

Template Driven Websites

Template driven websites plus RESTful resources is the sweet spot for Java web development. Not only do you gain the ability to theme and re-theme your web application, you also get:

1. Rapid development — leverage modern front-end frameworks.
2. View-controller separation — view templates and REST controllers are completely

separated.
3. Role separation — developers handle the REST code and designers handle the content.
4. On-the-fly UI changes — just refresh the browser — no lost sessions, server restarts, or class loader issues.

Download the Source Code

The full source to the Content API and example application are available to email subscribers. Join my list and receive the Apache Licensed code along with StackHunter.com updates and articles like this one directly in your inbox.

Your email address Sign up

About Dele Taylor

Dele Taylor is the founder of StackHunter: http://StackHunter.com. You can follow him on Twitter @DeleTaylor and Google +DeleTaylor.