ecmarchitect.com

Alfresco Developer: Intro to the Web Script Framework

October, 2007

Jeff Potts

Alfresco Developer: Intro to the Web Script Framework
October 2007
Jeff Potts

Introduction

This article is an introduction to the Alfresco Web Script Framework that became available with release 2.1 of the product.

We'll continue to extend the "SomeCo Whitepapers" example started in previous articles. As a quick refresher, in those articles, we extended the out-of-the-box content model so that SomeCo could store custom metadata about one of their document types, whitepapers. We created a custom aspect called rateable that could be attached to any object that was user-rateable. Then, in the custom behavior article we wrote business logic associated with the rateable aspect that knew how to calculate the average user rating for a given piece of content. The calculation was triggered every time a rating was created or deleted. We used server-side JavaScript to create rating objects to test out our behavior but there wasn't an interface available that end users could use to rate whitepapers.

SomeCo is now ready to move to the next step: Exposing the rating functionality to the front-end. In their infinite wisdom, the team at SomeCo realizes that Alfresco's Web Scripts provide a nice way to expose a lightweight, RESTful API for working with whitepapers and ratings. So in this article, we'll roll our own REST API for retrieving a list of whitepapers, retrieving the average rating for a given whitepaper, retrieving a specific rating, posting a new rating for a whitepaper, and deleting all ratings for a given whitepaper. We'll use JavaScript for most of our controller logic but we'll see how to use Java as well. The view will be implemented using FreeMarker templates that return HTML and JSON.

The complete source code that accompanies this article is available at ecmarchitect.com. See the "More Information" section at the end of this article for the link. In addition to the code for this article, the zip includes the code created in the first two "SomeCo" articles so if you don't have to dig around for the code we build upon in this article.

Sound decent? Okay, let's get started.

What is the Web Script Framework?

Content-centric applications, whether they are inside or outside the firewall, are becoming more and more componentized. I think of this as turning traditional content management approaches inside-out. Rather than having a single, monolithic system responsible for all aspects of a content-centric web

application, loosely-coupled sub-systems are being integrated to create more agile solutions.

This approach requires that your content management system have a flexible and lightweight interface. You don't want to be locked in to a presentation approach based on the content repository you are working with. In fact, in some cases, you might have very little control over the tools that will be used to talk to your CMS.

Consider the exploding rate of Next Generation Internet (NGI) solutions, the growing adoption of wikis and blogs within an Enterprise ("Enterprise 2.0"), and the increasing popularity of mash-ups both inside and outside the Enterprise. These trends are driving implementations where the CMS is seen as a black-box component with the front-end (or perhaps many different front-ends) interacting with the CMS and other components via REST.

Among open source content management systems, Alfresco is one of the few that really lends itself to this approach because it has many options for interacting with the repository. These options have evolved over time. The following summarizes ways in which your front-end could work with the Alfresco repository prior to release 2.1:

- **Embed the repository**. Alfresco's repository can be embedded in a custom application. Using this approach you have the full power of the Alfresco foundation API. Of course, the downside is that you've just tightly coupled the repository with your application. Didn't we just talk about how important an open, loosely-coupled architecture is? Moving on...
- **Web Services**. Alfresco has had a SOAP-based Web Services API available for quite some time, but SOAP-based Web Services have heavier client-side requirements than their RESTful cousins. Some clients found that the out-of-the-box services were too chatty and had too much processing overhead to scale well so they ended up writing their own services and exposing them through the embedded Apache Axis server. So Web Services may not be the right fit in all cases.
- **JCR**. The JCR API is a standard way of working with content in a repository and can be leveraged remotely through RMI. This has the benefit of using a standards-based approach for interacting with the repository which theoretically reduces switching costs and makes the application easier to support. One challenge with this approach is that the JCR API may not do everything you need to do so you end up using the JCR in combination with one of the above approaches which reduces the "switching costs" benefit. Another potential issue is that it is Java-only.
- **URL Addressability**. Objects in the Alfresco repository are URL-addressable. And, FreeMarker templates and server-side JavaScript can be applied to any node in the repository. So, for example, you can write a FreeMarker template that returns XML or JSON. A front-end app can then post an XMLHttpRequest to Alfresco that specifies a node reference and a

reference to the FreeMarker template. Alfresco will process the FreeMarker template in the context of the node specified and return the results. This is the closest you can get to the functionality of the Web Script Framework prior to 2.1.

These options are all still available and may make sense depending on exactly what you are trying to do. But with 2.1 there's a potentially better way for interacting with the repository—the Web Script Framework.

The Web Script Framework essentially improves on the basic idea that started with URL addressability. Think of a web script as a chunk of code that is mapped to a human-readable (and search-engine readable) URL. So, for example, a URL that returns expense reports pending approval might look like:

```
/alfresco/service/expenses/pending
```

while a URL that returns expenses pending approval for a specific user might look like:

```
/alfresco/service/expenses/pending/jpotts
```

In the URL above, you could read the “jpotts” component of the URL as an implied argument. A more explicit way to provide an argument would be like:

```
/alfresco/service/expenses/pending?user=jpotts
```

Or maybe “pending” is an argument as well which tells the web script what status of expense reports to return. The point is that the structure of the URL and how (and if) your URL includes arguments is completely up to you.

The response the URL returns is also up to you. Your response might return HTML, XML, JSON, or even a JSR-168 Portlet.

The Web Script Framework makes it easy to follow the Model-View-Controller (MVC) pattern, although it isn't required. The Controller is server-side JavaScript, a Java Bean, or both. The Controller handles the request, performs any business logic that is needed, populates the Model with data, and then forwards the request to the View. The View is a FreeMarker template responsible for constructing a response in the appropriate format. The Model is essentially a data structure passed between the Controller and the View.

The mapping of URL to controller is done through an XML descriptor which is responsible for declaring the URL pattern, whether the script requires a transaction or not, and the authentication requirements for the script. The descriptor optionally describes arguments that can be passed to the script as well as the response formats that are available.

The response formats are mapped to FreeMarker templates through naming convention. So, for example, the FreeMarker template that returns expenses as HTML would be named with an extension of “html” while the one that returns XML would be named with an extension of “xml”.

The descriptor, the JavaScript file, and the FreeMarker templates can reside either in the repository or on the file system. If a Web Script uses a Java Bean, the class must reside somewhere on the classpath.

With these building blocks in mind you may already be thinking of different ways you could leverage Web Scripts. If not, let me help. You can use Web Scripts to expose the Alfresco content repository through a RESTful API to:

- Enable a front-end web application written in any language that can talk HTTP to retrieve repository data in XML, JSON, or any other format or to persist data to the repository;
- Populate JSR-168 portlets;
- Capture user-contributed content/data;
- Interact with a business process (e.g., a JBPM workflow) through non-web client interfaces such as email;
- Create ATOM or RSS feeds for repository content or business process data; and
- Decompose the existing web client into smaller components which could potentially lend itself to being re-born in new and exciting ways!

Okay, *you* probably shouldn't tackle that last one but rest assured that Alfresco is already working on it.

The last thing to mention is that Web Scripts are executed in a "Web Script Runtime". In 2.1, there are three runtimes available out-of-the-box. The servlet runtime executes all web scripts requested via HTTP. The JSF runtime that allows JSF components to execute scripts. A JSR-168 runtime allows portlets to invoke web scripts directly.

You can write your own runtime if these don't meet your needs. Alfresco may add more in the future. At some point, you could see web script execution separated entirely from the Alfresco web application into its own process which would lend itself to load-balancing, scalability, etc.

In this article, we'll focus on the servlet runtime for HTTP.

Web Scripts Directory

The web client comes with a tool for listing and reloading web script definitions. To get to the tool, go to http://localhost:8080/alfresco/service/index. You'll see links that let you browse the list of deployed web scripts and a button labeled "Refresh list of Web Scripts". Although making changes to web scripts that reside in the repository does not require a restart, you may have to refresh the index with this button after a change.

Click through the links to see what is available out of the box. You'll notice that the tool itself is built using web scripts.

Examples

Let's walk through some examples. We're going to start with a very simple Hello World web script. After that, we'll get progressively more complex until, at the end, we have a REST-based interface for creating, reading, and deleting SomeCo whitepaper ratings.

Hello World Example

Let's implement the most basic web script possible: A Hello World script that echoes back an argument. We'll need one descriptor and one FreeMarker template. Do the following:

1. Log in to Alfresco.
2. Navigate to /Company Home/Data Dictionary/Web Scripts Extensions.
3. Create a file called helloworld.get.desc.xml with the following content:

```
<webscript>
  <shortname>Hello World</shortname>
  <description>Hello world web script</description>
  <url>/helloworld?name={nameArgument}</url>
</webscript>
```

4. Create a file called helloworld.get.html.ftl with the following content:

```
<html>
    <body>
        <p>Hello, ${args.name}!</p>
    </body>
</html>
```

5. Go to http://localhost:8080/alfresco/service/index and press the Refresh button. If you then click the "List Web Scripts" link you should be able to find the web script you just defined.
6. Now go to http://localhost:8080/alfresco/service/helloworld?name=Jeff. You should see:

```
Hello, Jeff!
```

A few things to note. First, notice the file names include "get". That's the HTTP method used to call the URL. In later examples we'll see how to use POST and DELETE. By differentiating on the HTTP method, you can have multiple controllers for the same "service" depending on how the service is called (GET vs. POST, etc.). Second, in this case we only had one argument but we could add as many as we need. Watch out, though! Descriptors must be valid XML which means ampersands must be escaped. So the proper way to define a URL with multiple arguments is:

```
  <url>/helloworld?name={nameArgument}&amp;secondArg={anotherArg}</url>
```

Third, notice we didn't include a JavaScript file in this example but the script still ran because controllers are optional.

Most scripts are going to use a controller, though, so let's go ahead and add one.

1. Create a file called helloworld.get.js with the following content:

```
model.foo = "bar";
```

2. Update your helloworld.get.html.ftl file with the following content:

```
<html>
    <body>
        <p>Hello, ${args.name}!</p>
        <p>Foo: ${foo}</p>
    </body>
</html>
```

3. Go to http://localhost:8080/alfresco/service/index and press the Refresh button. This is required because you added a controller that the web script run-time didn't know about.
4. Now go to your web browser and enter the same URL from the first example which was http://localhost:8080/alfresco/service/helloworld?name=Jeff. You should see:

```
Hello, Jeff!
Foo: bar
```

What's going on here is that the controller is getting executed before the FreeMarker template. In the controller we can do anything the Alfresco JavaScript API can do. In this case, we didn't leverage the JavaScript API at all—we just put some data into the “model” object which was then read by the FreeMarker template. In subsequent examples the controller will have more work to do and in one case, we'll use Java instead of JavaScript for the controller.

SomeCo Whitepaper User-contributed Ratings Examples

We want to create a REST API that front-end developers can use to find whitepapers and ratings as well as post new ratings. Before we dive in, it probably makes sense to rough out the API.

URL	Method	Description	Response formats
/someco/whitepapers	GET	Returns a list of whitepapers.	HTML, JSON
/someco/rating?id={id}	GET	Gets the average rating for a given whitepaper by passing in the whitepaper's noderef.	HTML, JSON
/someco/rating?id={id}&rating={rating}&user={user}	POST	Creates a new rating for the specified whitepaper by passing in a rating value and the user who posted the rating.	HTML, JSON
/someco/rating?id={id}	DELETE	Deletes all ratings for a specified whitepaper.	HTML

Table 1: Planned ratings API

When this API is in place, front-end developers can incorporate whitepapers and user-contributed ratings into the SomeCo web site. The following screenshots show pages that use the API we're going to build to query for whitepaper and ratings data. It looks like the folks at SomeCo have shamelessly ripped off the Optaros publications section. They didn't even bother to change the logo.

Illustration 1: The index of whitepapers uses an AJAX call to retrieve whitepaper metadata and ratings.

You can't tell from the screenshots, but the ratings widget is clickable. When clicked it sends an asynchronous post to the /someco/rating URL described in the table above. When the “Get the White Paper” link is clicked, the page in Illustration 2 is displayed. I reused the description from the index page for the Executive Summary. In the real world this would probably be a more lengthy description separate from the introduction on the index page. The “Download this white paper” link uses the standard “Download URL” to give the user direct access to the content.

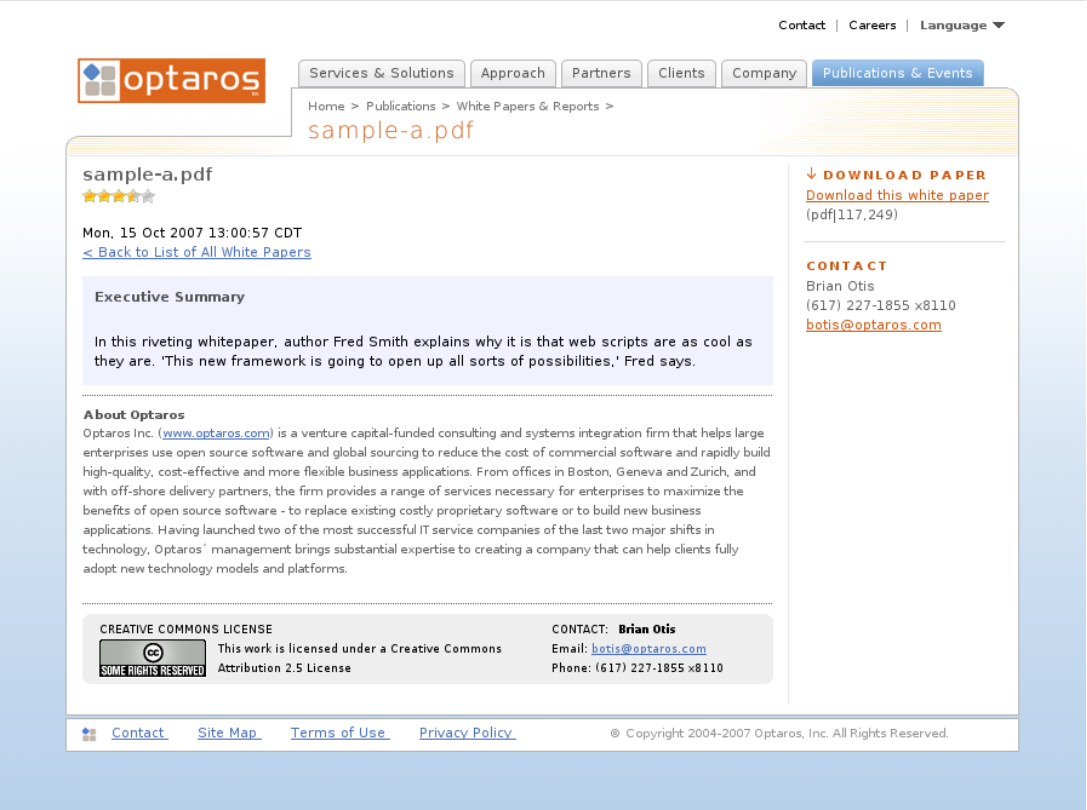

Illustration 2: The detail page also uses an AJAX call and includes the same ratings widget as well as a download link

Listing all Whitepapers

As a quick review, recall that SomeCo writes whitepapers and manages those papers with Alfresco. Some whitepapers are published to the web site. A custom aspect called “webable” has a flag called isActive. Whitepapers with the isActive flag set to true should be shown on the web site. For this article, we're going to ignore the webable aspect and the isActive flag. We'll just assume any sub-type of sc:whitepaper found in the /Someco/Whitepapers folder is fair game. (If this bugs you, see the sidebar).

Let's write a web script that returns all whitepapers. We want the list in two formats HTML and JSON. HTML will allow us to easily test the service and JSON will make it easy for code on the front-end to process the list. This will require four files: one descriptor, one JavaScript controller, and two FreeMarker templates—one for each format.

Before we start coding, let's talk a bit about organization. First, packages. The Web Script Framework allows us to organize script assets into a hierarchical folder or package structure. Just as it is with Java it is probably a good idea to do this for all web scripts. Following a reverse domain name pattern is probably a good convention. So we'll be using "com/someco" for our package which means our files will reside under /Company Home/Data Dictionary/Web Scripts Extensions/com/someco.[1]

Next, URLs can follow any pattern we want, but they will always start with <alfresco webapp>/service where "<alfresco webapp>" is the name of the Alfresco web application context (usually "alfresco"). Because the URL pattern must be unique, it is probably a good idea to incorporate the package name in the URL. In this article we'll prefix all URLs with "someco".

Alfresco reserves certain package names and URLs for their own use (see the Alfresco wiki) but by following the conventions proposed here, you'll steer clear of those.

Sidebar: Enabling a subset of whitepapers for web display

The real-world solution this example is based on uses UI actions to enable and disable the "sc:isActive" flag. An Evaluator class hides or shows the "Enable Web" or "Disable Web" UI action link based on the value of the flag and the group membership of the user. If you want to do something similar on your own, the sc:webable aspect is in the model included with the source code for this article. And, I've included two scripts (enableWeb.js and disableWeb.js) that you can use to attach the webable aspect and set the isActive flag appropriately. If you want the whitepaper service to filter the list based on the isActive flag, the Lucene query in the whitepaper.get.js file needs to be appended with "@sc\\:isActive:true" to show only active whitepapers. I left this functionality out of the example because it isn't core to the topic.

Finally, you've seen that web script assets can reside in the repository, but they can also reside in the file system. The only requirement is that they be on the classpath, but I suggest that they reside in the "alfresco/extension" directory just like your other extensions. Following the package structure suggested earlier, if we were to store our scripts on the file system, rather than the repository, we'd put them in alfresco/extension/templates/webscripts/com/someco.

The advantage of using the file system is that the web scripts that make up your solution can be deployed alongside your other extensions without requiring anyone to upload them to the repository. The disadvantage is that changes require a restart.

In the source code I've provided with this article, the web scripts are set up to deploy to the file system with the other extensions. If, instead of deploying the sample code, you want to follow along and you want to avoid restarts, move my scripts out of the alfresco extension directory before you deploy, then upload your scripts to the repository like we did for the Hello World example.

1 It isn't required that you use the "Web Scripts Extensions" folder in the repository. I did it because it seemed consistent with how web client customizations are deployed (using the alfresco/extension folder). Web scripts placed in the "Web Scripts Extensions" folder that have the same file names as those in the "Web Scripts" folder will override the scripts stored in "Web Scripts". For this reason, if you want others to be able to override your scripts, use the "Web Scripts" folder rather than "Web Scripts Extensions". See the Alfresco wiki for more on web script folder search order.

If scripts are defined in the repository as well as the classpath, the files in the repository take precedence over the files on the classpath. The Alfresco wiki documents the search order for web scripts (See “Where to find more information” at the end of this article for a list of references).

Step 1: The descriptor

With that, we should be good to go. The first step is to create the descriptor file. It should be named whitepapers.get.desc.xml and should look like this:

```
<webscript>
  <shortname>Get all whitepapers</shortname>
  <description>Returns a list of active whitepapers</description>
  <url>/someco/whitepapers</url>
  <url>/someco/whitepapers.json</url>
  <url>/someco/whitepapers.html</url>
  <format default="json">extension</format>
  <authentication>guest</authentication>
  <transaction>none</transaction>
</webscript>
```

There are a few elements in this descriptor we didn't see in the Hello World example. First, notice that there are multiple URL elements. There is one URL for each format plus a URL without a format. This shows how to request a different output format from the same base URL. Because the URLs differ only in format, it isn't strictly required that they be listed in the descriptor, but it is a good practice.

In this case, we're using the “extension” syntax—the extension on the URL specifies the format. An alternative syntax is to use the “argument” syntax like this:

```
  <url>/someco/whitepapers?format=json</url>
  <url>/someco/whitepapers?format=html</url>
```

My current thinking is that the extension syntax is preferred because it is friendlier to search engines but there may be reasons to use the argument syntax.

The format element declares the type of syntax we're using and defines a default output format. If you want to accept either syntax, you can use “any” as the format. In our case, using this descriptor if someone uses the argument syntax, they'll get an Error 500. If someone uses the URL without specifying a format, they'll get JSON.

The authentication element declares the lowest level of authentication required for this script. If your script touches the repository you will want this to be Guest or higher. Other options are “none”, “user”, and “admin”.

The transaction element specifies the level of transaction required by the script. Listing whitepapers doesn't need a transaction so we've got it set to “none”. Other possible values are “required” and “requiresnew”.

Next, we need a controller. Create a file called whitepapers.get.js with the following content:

```
<import resource="classpath:alfresco/extension/scripts/rating.js">

var whitepapers =
search.luceneSearch("PATH:\"/{http://www.alfresco.org/model/application/1.0}compan
y_home/{http://www.alfresco.org/model/content/1.0}Someco\"
+TYPE:\"{http://www.someco.com/model/content/1.0}whitepaper\"");

if (whitepapers == null || whitepapers.length == 0) {
      logger.log("No whitepapers found");
      status.code = 404;
      status.message = "No whitepapers found";
      status.redirect = true;
} else {
      var whitepaperInfo = new Array();
      for (i = 0; i < whitepapers.length; i++) {
          var whitepaper = new whitepaperEntry(whitepapers[i],
getRating(whitepapers[i]));
          whitepaperInfo[i] = whitepaper;
      }
      model.whitepapers = whitepaperInfo;
}

function whitepaperEntry(whitepaper, rating) {
      this.whitepaper = whitepaper;
      this.rating = rating;
}
```

The first thing to notice about the script is that we're importing another script. The rating.js script was created as part of the last article to contain logic used to calculate the average rating. The idea here is that retrieving a rating is also business logic related to a rating, so it should reside in the rating.js file as well. This makes it easy for us to reuse that logic in other scripts. We'll see the updated version of the rating.js script momentarily. (The ability to import a script from another script was added with release 2.1. The import tag is not native to the Rhino JavaScript implementation).

The next thing to notice is that the script queries the repository using Lucene to get a list of whitepapers. Look at what happens if there are no whitepapers found. The response code gets set to 404 which is the standard HTTP response code for “File not found”. Alfresco has a standard response template for error codes but you can override it with your own by creating FreeMarker templates that follow a specific naming convention. For example, we could have a custom 404 response template for whitepapers by creating a file called whitepapers.get.html.404.ftl. See the Alfresco wiki for more information.

The last thing that happens is that we build a new Array for our results. I could just set model.whitepapers equal to the whitepapers variable that contains the query results but I want to add some data to the result set, so I'm building a new Array and setting that to the model. (I know the average rating is a property of a whitepaper, so it may not yet be obvious why I have a separate function

for retrieving the rating or why I would store the rating in the model separate from the whitepaper. Trust that it will make sense later).

Remember that our controller imports a script called rating.js. This script resides in our classpath but the import tag also supports including scripts that reside in the repository. If you still have rating.js around from the custom behaviors article, the difference between it and this one is a new function called getRating as shown below:

```
function getRating(curNode) {
   var rating = {};
   rating.average =
curNode.properties["{http://www.someco.com/model/content/1.0}averageRating"];
   rating.count =
curNode.properties["{http://www.someco.com/model/content/1.0}ratingCount"];
   return rating;
}
```

The function simply retrieves the averageRating and ratingCount properties from the specified node and returns them in a rating object. The full source for rating.js is in the accompanying source code.

Assuming there are items in the search results, we'll need FreeMarker templates to process them. Let's create the HTML response template first. Create a new file called whitepapers.get.html.ftl with the following content:

```
<#assign datetimeformat="EEE, dd MMM yyyy HH:mm:ss zzz">
<html>
  <body>
     <h3>Whitepapers</h3>
     <table>
        <#list whitepapers as child>
           <tr>
              <td><b>Name</b></td><td>${child.whitepaper.properties.name}</td>
           </tr>
           <tr>

  <td><b>Title</b></td><td>${child.whitepaper.properties["cm:title"]}</td>
           </tr>
           <tr>
              <td><b>Link</b></td><td><a
href="${url.context}${child.whitepaper.url}?guest=true">${url.context}${child.whit
epaper.url}</a></td>
           </tr>
           <tr>
              <td><b>Type</b></td><td>${child.whitepaper.mimetype}</td>
           </tr>
           <tr>
              <td><b>Size</b></td><td>${child.whitepaper.size}</td>
           </tr>
```

```
            <tr>
               <td><b>Id</b></td><td>${child.whitepaper.id}</td>
            </tr>
            <tr>
               <td><b>Description</b></td>
               <td><p><#if child.whitepaper.properties["cm:description"]?exists &&
child.whitepaper.properties["cm:description"] !=
"">${child.whitepaper.properties["cm:description"]}</#if></p>
               </td>
            </tr>
            <tr>
               <td><b>Pub
Date</b></td><td>${child.whitepaper.properties["cm:modified"]?string(datetimeforma
t)}</td>
            </tr>
            <tr>
               <td><b><a
href="${url.serviceContext}/rating.html?id=${child.whitepaper.id}&guest=true">Rati
ng</a></b></td>
               <td>
               <table>
                  <tr>
                     <td><b>Average</b></td><td>${child.rating.average}</td>
                  </tr>
                  <tr>
                     <td><b>Count</b></td><td>${child.rating.count}</td>
                  </tr>
               </table>
               </td>
            </tr>
            <#if !(child.whitepaper == whitepapers?last.whitepaper)>
            <tr><td colspan="2" bgcolor="999999"> </td></tr>
            </#if>
         </#list>
      </table>
   </body>
</html>
```

This template iterates through the query results passed in by the controller, and builds an HTML table with properties of each whitepaper. (Yes, the table is ugly. Yes, you could use CSS to spruce it up tremendously or even remove the table entirely. But for SomeCo, this response template is really for debugging purposes only and I didn't want to fool with the CSS so a table it is).

The last thing we have to do before we test the web script is create the JSON response template. Create a file called whitepapers.get.json.ftl with the following content:

```
<#assign datetimeformat="EEE, dd MMM yyyy HH:mm:ss zzz">
```

```
{"whitepapers" : [
<#list whitepapers as child>
   {
   "name" : "${child.whitepaper.properties.name}",
   "title" : "${child.whitepaper.properties["cm:title"]}",
   "link" : "${url.context}${child.whitepaper.url}",
   "type" : "${child.whitepaper.mimetype}",
   "size" : "${child.whitepaper.size}",
   "id" : "${child.whitepaper.id}",
   "description" : "<#if child.whitepaper.properties["cm:description"]?exists &&
child.whitepaper.properties["cm:description"] !=
"">${child.whitepaper.properties["cm:description"]}</#if>",
   "pubDate" :
"${child.whitepaper.properties["cm:modified"]?string(datetimeformat)}",
   "rating" : {
      "average" : "${child.rating.average}",
      "count" : "${child.rating.count}",
      }
   }
   <#if !(child.whitepaper == whitepapers?last.whitepaper)>,</#if>
</#list>
]
}
```

Again, just like the HTML response template, the script iterates through the result set but this one outputs JSON. The JSON structure is completely arbitrary.

Assuming you have some test data in your repository (Someco Whitepaper objects in your Someco/Whitepapers folder) you should be able to refresh the web script list and run the web script. Because we told Alfresco that this script requires Guest access or higher, you'll need to either log in to Alfresco before running the script, authenticate with a valid user and password when the basic authentication dialog is presented, or append "&guest=true" to the URL like this: http://localhost:8080/alfresco/service/someco/whitepapers.html&guest=true. If you forget the ".html" you'll get a JSON response because we set that to the default.

If all goes well you should see something similar to the figure below:

Whitepapers

Name	sample-d.pdf
Title	sample-d.pdf
Link	/alfresco/d/d/workspace/SpacesStore/f6a570ec-6bf1-11dc-b587-f368be3aeacb/sample-d.pdf
Type	application/pdf
Size	117,248
Id	f6a570ec-6bf1-11dc-b587-f368be3aeacb
Description	Here's another humdinger from analyst Jane Doe. Doe has spent most of her career focusing on content management. In this paper, Jane lays out a strategy for a successful ECM implementation.
Pub Date	Mon, 15 Oct 2007 13:01:21 CDT
Rating	Average 1.923 Count 13

Name	sample-a.pdf
Title	sample-a.pdf
Link	/alfresco/d/d/workspace/SpacesStore/0e41db47-6bbd-11dc-8966-8d9225693aae/sample-a.pdf
Type	application/pdf
Size	117,249
Id	0e41db47-6bbd-11dc-8966-8d9225693aae
Description	In this riveting whitepaper, author Fred Smith explains why it is that web scripts are as cool as they are. 'This new framework is going to open up all sorts of possibilities,' Fred says.
Pub Date	Mon, 15 Oct 2007 15:58:46 CDT
Rating	Average 3.167 Count 12

Debugging

Did it work? If not, it's time to debug. The first thing you're going to want to do is to go into log4j.properties and set log4j.logger.org.alfresco.repo.jscript to DEBUG. This will cause any logger.log statements in your controller to write to catalina.out.

Another tool you'll want to leverage is the web script list. You can use it to see (1) if Alfresco knows about your script and (2) the version of the scripts the run-time knows about. For example, you can go to http://localhost:8080/alfresco/service/script/com/someco/whitepapers/whitepapers.get and Alfresco will dump the descriptor and all of the response templates.

The Node Browser can be helpful to debug problems as well. In this case, for example, we're running a Lucene query in our JavaScript. If the controller isn't finding any whitepapers even though you've created test data, try executing the query in the Node Browser. If it doesn't return results, there's something wrong with your test data.

Retrieving the Rating for a Whitepaper

Getting a specific rating is roughly the same as getting a whitepaper but it is a bit easier because of our

existing getRating function in rating.js. All the controller has to do is grab the ID argument, locate the node, then call getRating as shown below:

```
<import resource="classpath:alfresco/extension/scripts/rating.js">

if (args.id == null || args.id.length == 0) {
   logger.log("ID arg not set");
   status.code = 400;
   status.message = "Node ID has not been provided";
   status.redirect = true;
} else {
   logger.log("Getting current node");
   var curNode = search.findNode("workspace://SpacesStore/" + args.id);
   if (curNode == null) {
      logger.log("Node not found");
      status.code = 404;
      status.message = "No node found for id:" + args.id;
      status.redirect = true;
   } else {
      model.rating = getRating(curNode, args.user);
   }
}
```

The descriptor and response templates are very similar to the whitepaper example so I won't include them here. After we get the post in place, we'll revisit the HTML response template by making some updates that help us test.

For now, here's what a successful JSON call to the rating service returns:

```
{"rating" :
   {
      "average" : "1.923",
      "count" : "13",
   }
}
```

Posting a Rating with a Java-backed Web Script

Before we talk about the POST web script we should talk about authentication. All of our /someco scripts require Guest access or higher. That means we either have to have an active session already established, we have to append “&guest=true” to the URL, or we have to login when the browser presents us with a basic authentication dialog. (Another option is to get a ticket from a web service call, but that's not in the scope of this article).

SomeCo doesn't want to open up write access to the /Someco/Whitepapers folder to Guest users who might try to access the repository via the web client so that means we need a real user account in order to write new rating objects. We could use "user" authentication for the POST but SomeCo doesn't want to set up user accounts for every user that might rate content.

The solution is to let Guest call the POST URL but leverage the Alfresco Java API to "run as" a different user. (In our case we'll use admin but a user account dedicated to the purpose of creating ratings is probably a better idea). The post logic will reside in a Java Bean rather than a server-side JavaScript file.

The descriptor and the response templates look like the examples we've seen so far so I won't repeat them here. Take a look at the accompanying source code if you are curious.

The piece that is new is the use of Java as the controller so let's spend some time on that. We'll implement this Java-backed web script in three steps. The first step is to write the business logic for creating the rating. Just like when we put the business logic in the rating.js file to promote reuse, we're going to use the Rating bean we created in the previous article for the new create() method. The second step is to write the Java bean that functions as our controller. The third step is to configure the controller bean via Spring so that Alfresco knows to invoke it when the web script is called.

Step One: Business logic

Add the following method to the com.someco.behavior.Rating class we created in the previous article.

```
public void create(NodeRef nodeRef, int rating, String user) {
     boolean switchUser = false;

     String currentUser = AuthenticationUtil.getCurrentUserName();
     if (!currentUser.equals("admin")) {
        logger.debug("Current user is not admin so switching to admin");
        AuthenticationUtil.setCurrentUser("admin");
        switchUser = true;
     }

     UserTransaction txn = transactionService.getUserTransaction();

     try {
        txn.begin();

        // add the aspect to this document if it needs it
        if (nodeService.hasAspect(nodeRef,
              Qname.createQName(SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
                                SomeCoModel.ASPECT_SC_RATEABLE))) {
           logger.debug("Document already has aspect");
        } else {
           logger.debug("Adding rateable aspect");
```

```
            nodeService.addAspect(nodeRef,
               Qname.createQName(SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
                                 SomeCoModel.ASPECT_SC_RATEABLE), null);
         }

         Map<QName, Serializable> props = new HashMap<QName, Serializable>();
         props.put(QName.createQName(SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
"rating"), rating);
         props.put(QName.createQName(SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
"rater"), user);

         nodeService.createNode(nodeRef,
               Qname.createQName(SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
                                 SomeCoModel.ASSN_SC_RATINGS),
               Qname.createQName(SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
                                 "rating" + new Date().getTime()),
               Qname.createQName(SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
                                 SomeCoModel.TYPE_SC_RATING), props);

         txn.commit();

      } catch(Throwable e) {
         try {
            if (txn.getStatus() == Status.STATUS_ACTIVE) txn.rollback();
         } catch (Throwable ee) {
            e.printStackTrace();
         }
      }

      if (switchUser) AuthenticationUtil.setCurrentUser(currentUser);

   }
```

This is a bit painful to look at but basically what's going on is:

- If the current user is not admin, the current user is set to admin
- A new transaction is started
- If the node doesn't yet have the rateable aspect, it is added
- The rating and rater properties are set
- The transaction is committed
- If the current user was switched to admin, the current user is switched back to whomever it was before the switch to admin

Step Two: Web script controller bean

With the create() method in place, all we have to do is write a Java class that grabs the id, rating, and rater arguments and calls the method. To do that, create a new class called com.someco.scripts.PostRating. The class name isn't significant but it seems like following some sort of descriptive convention could be helpful here if there are a large number of Java-backed scripts. The class needs to extend org.alfresco.webscripts.DeclarativeWebScript. Our logic goes in executeImpl as shown below.

```
public class PostRating extends org.alfresco.web.scripts.DeclarativeWebScript {

   Logger logger = Logger.getLogger(PostRating.class);

   private Rating ratingBean;

   @Override
   protected Map<String, Object> executeImpl(WebScriptRequest req, WebScriptStatus
status) {
      String id = req.getParameter("id");
      String rating = req.getParameter("rating");
      String user = req.getParameter("user");

      if (id == null || rating == null || rating.equals("0") || user == null) {
         logger.debug("ID, rating, or user not set");
         status.jsSet_code(400);
         status.jsSet_message("Required data has not been provided");
         status.jsSet_redirect(true);
      } else {
         NodeRef curNode = new NodeRef("workspace://SpacesStore/" + id);
         if (curNode == null) {
            logger.debug("Node not found");
            status.jsSet_code(404);
            status.jsSet_message("No node found for id:" + id);
            status.jsSet_redirect(true);
         } else {
            ratingBean.create(curNode, Integer.parseInt(rating), user);
         }

      }

      Map<String, Object> model = new HashMap<String, Object>();
      model.put("node", id);
      model.put("rating", rating);
      model.put("user", user);

      return model;
   }

   public Rating getRatingBean() {
```

```
      return ratingBean;
   }

   public void setRatingBean(Rating ratingBean) {
      this.ratingBean = ratingBean;
   }

}
```

This code should look strikingly similar to a JavaScript controller and in fact it does the same thing. It checks the arguments, sets an error code if the arguments are missing, and then writes some data to the model.

The controller gets the Rating class through Spring dependency injection. We'll configure that in our Spring config, which is the next step.

Step Three: Spring config for the Web script controller bean

The following shows the contents of someco-scripts-context.xml. The name of the file isn't important, but it must end with *context.xml.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE beans PUBLIC '-//SPRING//DTD BEAN//EN'
'http://www.springframework.org/dtd/spring-beans.dtd'>

<beans>
   <bean id="webscript.com.someco.ratings.rating.post"
class="com.someco.scripts.PostRating" parent="webscript">
      <property name="ratingBean">
         <ref bean="ratingBehavior" />
      </property>
   </bean>
</beans>
```

This should look like any other Spring bean config file you've seen. The web script magic is in the id and parent attributes. The id follows a naming convention. The convention is:

```
webscript.package.service-id.method
```

Pay close attention to the use of the singular “webscript” here versus the plural “webscripts” in the Data Dictionary folders. That's a potential multi-hour debugging session ending in a forehead slap with a “Doh!” if you aren't careful.

It is probably worth mentioning that a decision to use a Java-backed web script doesn't exclude the use of JavaScript for that web script. If you have both a Java class and a JavaScript file, the Java class gets executed first followed by the JavaScript. The script has access to everything the Java class put in the model and can update the model before passing it along to the response template.

Revisiting the rating.get.html.ftl template

Now we have everything we need in place but we don't have a great way to test the rating POST. So, what we'll do is add a little rating widget[1] to the rating.get.html.ftl template. A simple link would do but I wanted to test out the widget before incorporating it into a real page.

First, let's see what it looks like when we call /someco/rating.html/id=someid. The figure below shows a call when the whitepaper node has 13 ratings and an average of 1.923.

The purpose of the rating widget is two-fold. First, it graphically displays the average rating for a whitepaper. Second, each star in the widget is hot. So when you click one of the rating stars, an asynchronous post is made to /someco/rating which causes a new rating object to get created. The rating posted depends on the star clicked. The person submitting the rating would normally be passed in based on some sort of credential, maybe from a portal session or a cookie. In our little test, the rater gets pulled from the field.

Let's look at HTML first, then some of the JavaScript:

```
<p><a href="${url.serviceContext}/whitepapers.html?guest=true">Back to the
list</a> of whitepapers</p>
<p>Node: ${args.id}</p>
<p>Average: ${rating.average}</p>
<p># of Ratings: ${rating.count}</p>
<form name="login">
   Rater:<input name="userId"></input>
</form>
Rating: <div class="rating" id="rating_${args.id}"
style="display:inline">${rating.average}</div>
```

This is all basic HTML/FreeMarker stuff you've seen before. The last line sets up a div for the ratings

1 I used the code at http://www.progressive-coding.com/tutorial.php?id=6 as the starting point for the rating widget. Most of it is unchanged with the exception of changing the ratings from being 0-indexed to being 1-indexed and following the author's instructions to hook the widget into the page using Prototype which I already happened to have lying around.

widget. The id of the div uses the node ref of the whitepaper. This allows multiple ratings widgets to be on the same page and makes it easy for the JavaScript to pass the node ref on to the /someco/rating URL.
The second piece is the JavaScript. I'm going to omit some of the less interesting functions and just show the functions related to posting ratings. The accompanying source code has the full source.

```
function submitRating(evt) {
   var tmp = Event.element(evt).getAttribute('id').substr(5);
   var widgetId = tmp.substr(0, tmp.indexOf('_'));
   var starNbr = tmp.substr(tmp.indexOf('_')+1);
   alert("Post to URL:" + widgetId + "," + starNbr);
   if (document.login.userId.value != undefined && document.login.userId.value !=
"") {
      curUser = document.login.userId.value;
   } else {
      curUser = "jpotts";
   }
   postRating(widgetId, starNbr, curUser);
}

function postRating(id, rating, user) {
   if (receiveReq.readyState == 4 || receiveReq.readyState == 0) {
      receiveReq.open("POST", "/alfresco/service/someco/rating?id=" + id +
"&rating=" + rating + "&guest=true&user=" + user, true);
      receiveReq.onreadystatechange = handleRatingPosted;
      receiveReq.send(null);
   }
}

function handleRatingPosted() {
   if (receiveReq.readyState == 4) {
      alert("Post successful");
   }
}
```

Those of you familiar with AJAX techniques may be wondering why I didn't use Prototype to make the post since I was already using it with the rating widget. I had trouble getting Prototype to play nicely with the Web Script Framework. For some reason the arguments weren't getting recognized. So I punted and used the lower-level XMLHttpRequest. You'll also notice that I don't dynamically update the rating or re-init the widget after the successful post. My only excuse for that one is laziness.

Deleting ratings

Setting up a web script for delete is similar to the GET for ratings. The descriptor is named rating.delete.desc.xml. I set mine to require “admin” authentication. It seems rare that you would want

to delete all ratings for a given node but highly likely that if you are going to expose it, it should be for admins only.

As in previous examples, the controller JavaScript reads and checks the arguments then calls a function. In this case it is the deleteRatings function that has been added to rating.js. The body of the function is:

```
function deleteRatings(curNode) {

     // check the parent to make sure it has the right aspect
     if (curNode.hasAspect("{http://www.someco.com/model/content/1.0}rateable"))
{
        // continue, this is what we want
     } else {
        logger.log("Node did not have rateable aspect.");
        return;
     }

     // get the node's children
     var children = curNode.children;

     if (children != null && children.length > 0) {
        logger.log("Found children...iterating");
        for (i in children) {
           var child = children[i];
           logger.log("Removing child: " + child.id);
           child.remove();
        }
     }
}
```

The script bails if the node doesn't have the rateable aspects (because there wouldn't be any ratings). Otherwise, it grabs the children and deletes them. Note the important assumption that the only children that exist are ratings. If there's a possibility of other child associations, you'd obviously want to be more discriminating.

Example summary

We've implemented two GET scripts (one for whitepapers and one for rating), a POST script for creating new ratings, and a DELETE for clearing out ratings. At this point SomeCo has everything they need for building a front-end that talks to the Alfresco repository via REST. One piece of functionality I didn't show, but I've included in the source, is the ability for an optional “user” argument to be passed in to the two GET scripts. When present, the script will return the last rating for the specified user. I'll leave it to you to follow the source to figure out how that works.

Dealing with the cross-domain scripting limitation

You may have noticed that in all of my URL examples, I'm using localhost. In fact, the static HTML pages (whitepaper index and whitepaper detail) that make AJAX calls are also on localhost. I did this to simplify the example but in real life, it is highly likely that the code making an AJAX call to your web script will reside on a different host than the one where Alfresco lives. This creates a problem called the "cross-domain scripting limitation". The issue is that for security reasons browsers don't let you open an XMLHttpRequest to a different host than the one serving the page. There are a few ways you can handle this depending on your situation.

- **Use script tags**. One way to work around the problem is to use a script tag in which the src attribute points to a location on a different host. The browser thinks it is loading a JavaScript file but what it is really doing is calling your web script which returns JSON. The script tags can be output dynamically through document.write.
- **Use a proxy**. Servers aren't subject to the browser's security constraints. You can easily write your own Java servlet that acts as a reverse proxy. AJAX calls go against the proxy and pass in the "real" URL as an argument. The servlet then invokes the URL and returns the results.
- **Use a callback mechanism**. Alfresco claims to have a callback mechanism built in to the web script run-time. The way it is supposed to work is that you pass in a function name as an argument to the web script like "&alf_callback=someFunction". The function is supposed to get called when the page is loaded. I couldn't get it working and ended up filing a Jira ticket.
- **Deploy everything to the same server**. This is the least likely scenario to work in a production implementation but it's the one I chose for this article so we wouldn't have to spend a lot of time on the issue. The scripts and images the ratings widget depends on reside in someco/javascript and someco/images, respectively, under the Alfresco web root. The whitepaper index and whitepaper details pages I used for the screenshots at the beginning of the article are enhanced copies of the files used for the Optaros web site deployed to the ROOT web application folder.

Deploying and Testing

To run the sample as-is, all you have to do is:

1. Import the web-script-article-project.zip file into Eclipse.
2. Change build.properties to match your environment.
3. Run the default Ant task.

The default Ant task will compile all necessary code, JAR it up, zip up the JAR and the extensions into

the appropriate folder structure, and then unzip on top of the Alfresco web root which deploys the custom model, Spring config files, web client customizations, scripts, web scripts, and the images and JavaScript for the rating.get.html page to the appropriate directories.

After an error-free start up, create Someco/Whitepapers in your Company Home and upload a couple of test whitepapers. Upload and execute the addTestRating.js script in the context of each test whitepaper to create test rating objects.

You should then be able to run any of the web scripts identified in this article without any problems.

In case you are curious, my environment is:

- Ubuntu Dapper Drake
- MySQL 4.1 (with version 5.0.3 of the JDBC driver)
- Java 1.5.0_12
- Tomcat 5.5.x
- Alfresco 2.1.0 Enterprise, WAR-only distribution

Obviously, other operating systems, databases, and application servers will work as well. Web Scripts, however, only work starting with Alfresco 2.1.

Conclusion

This article has given you an introduction to the Alfresco Web Script Framework. We began with a very simple Hello World script and then gradually moved to more complex examples which culminated in a REST API for retrieving whitepapers, getting the average rating for a specific whitepaper, posting new ratings for a given whitepaper, and deleting all ratings for a specific whitepaper. We used both JavaScript and Java to implement controller logic. We used FreeMarker to output HTML as well as JSON. We saw some options for working around the cross-domain scripting limitation.

There are still topics left to explore. One example is using Web Scripts to integrate a portal like Liferay or JBoss Portal with Alfresco. Another is Microsoft Office-Alfresco integration which is based on Web Scripts. And what about using Web Scripts to customize the Web Client user interface? Hopefully, you've been inspired enough to take a look at those topics on your own. Maybe you'll even blog about your experience. If so, or if you have any other feedback, please let me know. I'd love to hear from you.

Where to find more information

- The complete source code that accompanies this article is available here from ecmarchitect.com.
- You may also enjoy previous articles in the Alfresco Developer series at ecmarchitect.com:

    - "Implementing custom behaviors", September, 2007.
    - "Working with Custom Content Types", June, 2007.
    - "Developing custom actions", January, 2007.
- Alfresco wiki pages related to this topic:
    - Alfresco Web Scripts wiki page
    - Alfresco Web Script Runtimes wiki page
    - Alfresco JavaScript API wiki page
    - Alfresco Template Guide (FreeMarker info) wiki page
    - For deployment help, see the Client Configuration Guide and Packaging and Deploying Extensions in the Alfresco wiki.
    - For general development help, see the Developer Guide.
    - For help customizing the data dictionary, see the Data Dictionary wiki page.
- Luis Sala's presentation on Web Scripts at the West Coast Alfresco+Liferay Meetup along with a podcast of the audio portion of the presentation is available at Luis' Fresh Talk blog.
- Learn more about JSON at json.org and FreeMarker at freemarker.sourceforge.net.
- The jMaki Project is a framework for building Ajax-enabled, Java web applications. Included as part of it is a proxy you can use to work around the cross-domain scripting limitation if you don't want to write your own.
- The JSR-168 Portlet Specification is available on the Java Community Process site.

About the Author

Jeff Potts is the Enterprise Content Management Practice Lead at Optaros, a leading Open Source and Next Generation Internet consultancy. Jeff has fifteen years of experience implementing content management, collaboration, and other knowledge management technologies for a variety of Fortune 500 companies. Jeff lives in Dallas, Texas with his wife and two kids. Read more at ecmarchitect.com.