ecmarchitect.com

Alfresco Developer: Implementing custom behaviors

September, 2007

Jeff Potts

Alfresco Developer: Implementing custom behaviors
September 2007
Jeff Potts

Introduction

This article discusses how to write your own custom behavior code in Java or JavaScript and then bind that code to node events or "policies".

In previous articles I've discussed how to create custom content models and how to write custom actions. In both cases, we've seen how to write code that works with custom content types, properties, aspects, and associations, but the code wasn't tightly coupled to the objects on which it operated. With an action, the business logic is triggered by something—an item in the user interface, a schedule, or a workflow—rather than being *bound* to the content type or aspect.

Actions are very useful when the business logic the action carries out is generic enough to be applied to many types of objects. The out-of-the-box "move" or "add aspect" actions are obvious examples.

There are times, though, when you want code to be tightly-coupled to a content type because you need to be sure it gets executed rather than leaving it up to a rule on a space that triggers an action or a workflow that does the same. Fortunately, Alfresco provides just such a mechanism—it's called "behaviors".

Behaviors are used throughout Alfresco. Auditing and versioning are examples where behaviors are involved. In our work with custom Alfresco implementations behaviors have come in handy several times. In one case we needed to default some metadata values using logic that couldn't be expressed using the Alfresco content model so we wrote a custom behavior that set the metadata appropriately. In another, we needed a way to synchronize metadata between folders and the items in those folders so we wrote a custom behavior to handle the sync.

In this article we'll see a simple example also based on a real-world implementation: using a custom behavior to compute the average user rating for a piece of content.

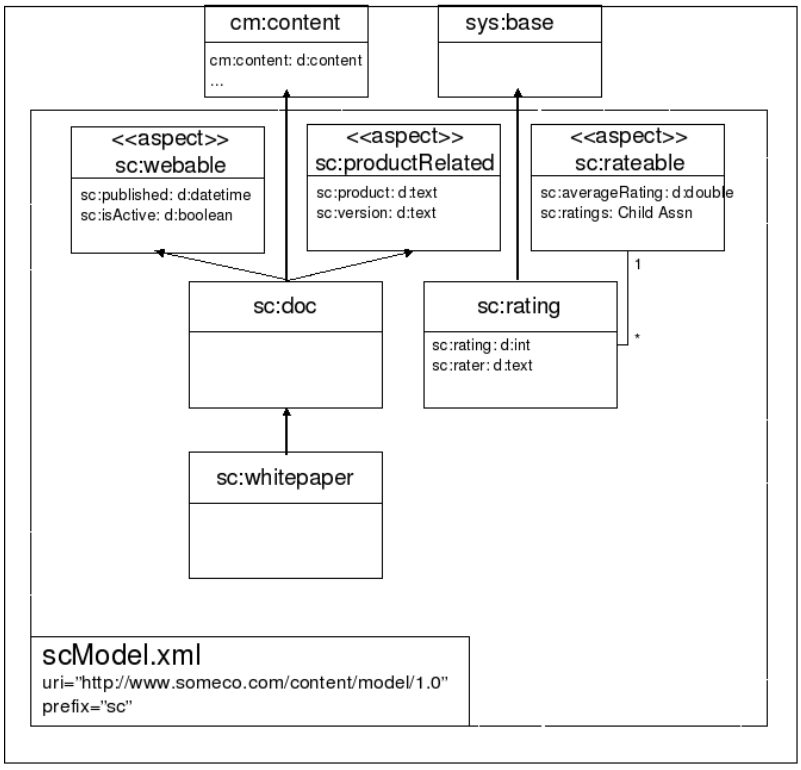
You should already be familiar with general document management and Alfresco web client concepts. If you want to follow along, you should also know how to write basic Java code. See "Where to find more information" at the end of this document for a link to the code samples that accompany this article. Specifically, you may want to read "Working with Custom Content Types" on ecmarchitect.com if you aren't already familiar with how to extend Alfresco's content model.

Example: User ratings

Recall from the Custom Content Types article that we created a custom type called a Whitepaper for a fictitious company called SomeCo. We used an aspect called “webable” that was attached to content objects we wanted to show on the web. So, for example, SomeCo might use Alfresco to manage all of its whitepapers, but show only a subset on the web. Whitepapers to be shown on the web get the webable aspect attached and the isActive flag set to true. The front-end can then query for whitepapers based on that flag.

Now let's extend that example further. Suppose that the Marketing folks at SomeCo have caught the Web 2.0 bug and they've decided to add user ratings to their web site. They would like users to be able to assign a rating to a whitepaper and to display the average of all ratings received for a specific whitepaper.

Assuming we want to store the ratings in Alfresco instead of directly in a relational database, one way to do this would be to create a custom “rating” type that would be related to a whitepaper through a child association. We could create a “rateable” aspect that would define the association as well as a property to store the average rating for that whitepaper.



Drawing 1: SomeCo's content model modified to support ratings

Drawing 1 shows the original custom content model with these enhancements.

That takes care of the data model, but where should we put the code that computes the average? One way to handle it would be to write an action that gets called by a rule. Any time a rating is added to a folder, the rule would trigger the action to update the average. But this isn't the best option because every time you wanted to use user ratings you'd have to make sure to set up a rule on the folder. A scheduled action might not be bad—it could be written to find all objects with the rateable aspect and then compute the average. But if you want the average rating computed in real-time (and let's assume we do) a scheduled action isn't a a great option.

As you've already guessed, the best option in our example is to use a behavior. We'll write the logic we need to compute the average and then bind it to the appropriate policies on the rating content type. Any time a rating gets created (or deleted), it will know how to find its parent (the whitepaper) and recalculate the average.

What can trigger a behavior?

So the rating content type will be bound to business logic that knows how to compute the overall average rating for a whitepaper. But what will trigger that logic? The answer is that there are a bunch of *policies* to which your behavior can be bound. To find out what's available, you need only look as far as the source code (or Javadocs). If you grep the "repository" project in the Alfresco source code for classes that end in "*Policies.java" you'll find four interfaces. Each of those interfaces contains inner interfaces that represent the policies you can hook into. Check the Javadocs or source code for specifics —I'm listing the methods in Table 1 so you can see the breadth of what's available.

Note: To make it easier to read, I'm omitting the inner interface which follows the pattern of <method-name>Policy. For example, the onContentUpdate method is a method of the inner interface OnContentUpdatePolicy.

Interface	Method
org.alfresco.repo.content.ContentServicePolicies	onContentUpdate
	onContentRead
org.alfresco.repo.copy.CopyServicePolicies	onCopyNode
	onCopyComplete
org.alfresco.repo.node.NodeServicePolicies	beforeCreateStore

Interface	Method
	onCreateStore
	beforeCreateNode
	onCreateNode
	onMoveNode
	beforeUpdateNode
	onUpdateNode
	onUpdateProperties
	beforeDeleteNode
	onDeleteNode
	beforeAddAspect
	onAddAspect
	beforeRemoveAspect
	onRemoveAspect
	beforeCreateNodeAssociation
	onCreateNodeAssociation
	beforeCreateChildAssociation
	onCreateChildAssociation
	beforeDeleteChildAssociation
	onDeleteChildAssociation
	onCreateAssociation
	onDeleteAssociation

Interface	Method
org.alfresco.repo.version.VersionServicePolicies	beforeCreateVersion
	afterCreateVersion
	onCreateVersion
	calculateVersionLabel

Table 1: Policies available for behavior binding

Which policy shall we use? We need to recalculate a whitepaper's rating either when a new rating is created or when a rating is deleted. One possibility would be to bind our behavior to the NodeService policy's onCreateChildAssociation and onDeleteChildAssociation policy for the whitepaper node. But that would mean we'd constantly be inspecting the association type to see if it we wanted to take any action because there could be other child associations besides ratings. Instead, we'll bind to the rating node's onCreateNode and onDeleteNode policies.

Java or JavaScript?

There are two options for writing the code for the behavior: Java or JavaScript. The decision as to which one to use depends on the standards you've settled on for the solution you are building. In this article, we'll implement the ratings example using Java but I'll also show you how to use JavaScript as an alternative.

Implementing and deploying the custom behavior

Before we start, a couple of notes about my setup:

- Ubuntu Dapper Drake (I know, I know. Past due for an upgrade).
- MySQL 4.1
- Tomcat 5.5.x
- Alfresco 2.1.0 Enterprise, WAR-only distribution

Obviously, other operating systems, databases, and application servers will work as well. Some of the functionality used in this example is available only with 2.1 so your mileage my vary if you use an older

release.

I'll also be building on the custom model defined in the Custom Types article. That code was originally written for Alfresco 2.0.1 but will run fine on Alfresco 2.1.0 with no changes needed. You don't need to download the code from that article for the code in this article to work. The code supplied with this article is a superset.

Java Example

Let's do the Java example first. Here are the steps we are going to follow:

1. Extend the SomeCo model defined in the previous article with our new rateable aspect and rating type (and all the UI config steps that go with it).
2. Write server-side JavaScript that creates test ratings.
3. Write the custom behavior bean and bind it to the appropriate policies.
4. Configure a Spring bean to initialize our behavior class and pass in any dependencies.
5. Build, deploy, restart and test.

Let's get started.

Extend the model

Open the scModel.xml file we created in the previous article. We need to add a new type and a new aspect. Insert the “sc:rating” type definition after the existing “sc:whitepaper” definiton:

```
<type name="sc:rating">
         <title>Someco Rating</title>
         <parent>sys:base</parent>
         <properties>
                  <property name="sc:rating">
                           <type>d:int</type>
                           <mandatory>true</mandatory>
                  </property>
                  <property name="sc:rater">
                           <type>d:text</type>
                           <mandatory>true</mandatory>
                  </property>
         </properties>
</type>
```

Listing 1: The rating type in the scModel.xml file.

Note that sc:rating inherits from sys:base. That's because we don't intend to store any content in a rating object, only properties.

Now add the “sc:rateable” aspect after the existing “sc:productRelated” aspect. The rateable aspect will have one property to store the average rating and will also define the child association for a piece of content's related ratings. Using an aspect gives us the ability to add ratings functionality to any piece of content in the repository. (In fact, you don't need any of the SomeCo model from the previous article to use the ratings functionality. That's the beauty of aspects).

```
<aspect name="sc:rateable">
         <title>Someco Rateable</title>
         <properties>
                  <property name="sc:averageRating">
                           <type>d:double</type>
                           <mandatory>false</mandatory>
                  </property>
         </properties>
         <associations>
                  <child-association name="sc:ratings">
                           <title>Rating</title>
                           <source>
                                    <mandatory>false</mandatory>
                                    <many>true</many>
                           </source>
                           <target>
                                    <class>sc:rating</class>
                                    <mandatory>false</mandatory>
                                    <many>true</many>
                           </target>
                  </child-association>
         </associations>
</aspect>
```

Listing 2: The rateable aspect in the scModel.xml file.

The rateable properties and associations need to show up on property sheets for objects with the aspect so add the following to web-client-config-custom.xml:

```
<!--  add rateable aspect properties to property sheet -->
<config evaluator="aspect-name" condition="sc:rateable">
         <property-sheet>
                  <show-property name="sc:averageRating" display-label-
id="average" read-only="true" />
                  <show-child-association name="sc:ratings" display-label-
id="ratings" read-only="false" />
         </property-sheet>
</config>
```

Listing 3: Updated web-client-config-custom.xml file to show rateable aspect properties.

The rateable aspect needs to show up on the “add aspect” list so add the rateable aspect to the existing list of SomeCo custom aspects in web-client-config-custom.xml. Note that this snippet only shows the “<aspects>” element of the “Action Wizards” config. The rest is unchanged from the previous article:

```
<config evaluator="string-compare" condition="Action Wizards">
         <!-- The list of aspects to show in the add/remove features action -->
         <!-- and the has-aspect condition -->
         <aspects>
                  <aspect name="sc:webable"/>
                  <aspect name="sc:productRelated"/>
                  <aspect name="sc:rateable"/>
         </aspects>

         <!-- Remaining config is unchanged -->
</config>
```

Listing 4: Updated web-client-config-custom.xml file to add rateable to aspect list.

The label IDs need values in the webclient.properties file:

```
#sc:rateable
average=Avg Rating
ratings=Ratings
```

Listing 5: Updated webclient.properties file for externalized strings.

Recall that in the previous article we decided it was a good idea to use a class for our model to store constants such as the names of types, properties, and aspects. Modify com.someco.model.SomeCoModel.java to include new constants for the type, aspect, and properties we just added to the model.

```
    public static final String TYPE_SC_RATING = "rating";
    public static final String ASPECT_SC_RATEABLE = "rateable";
    public static final String PROP_RATING = "rating";
    public static final String PROP_AVERAGE_RATING= "averageRating";
```

Listing 6: Updated SomeCoModel with new constants

That's all we need to do for the model. After restarting Alfresco you should see the aspect in the "Add Aspect" action configuration wizard.

Write a server-side JavaScript file that creates test data

In the last article we used Java to test out the model by writing code against the Alfresco Web Services API. This time, let's use JavaScript to create a server-side script that we can run any time we need to create some test rating nodes for a piece of content.

First, log in to the Alfresco web client and navigate to Company Home/Data Dictionary/Scripts.

Next, create a new piece of content named addTestRating.js with the following content:

```
// add the aspect to this document if it needs it
if (document.hasAspect("sc:rateable")) {
  logger.log("Document already as aspect");
} else {
  logger.log("Adding rateable aspect");
  document.addAspect("sc:rateable");
```

```
}

// randomly pick a num b/w 1 and 5 inclusive
var ratingValue = Math.floor(Math.random()*5) + 1;

var props = new Array(2);
props["sc:rating"] = ratingValue;
props["sc:rater"] = person.properties.userName;

// create a new ratings node and set its properties
var ratingsNode = document.createNode("rating" + new Date().getTime(),
"sc:rating", props, "sc:ratings");
ratingsNode.save();
logger.log("Ratings node saved.");
```

Listing 7: The addTestRating.js script can be executed to create test rating nodes.

Now create a piece of content in your repository and then use "Run Action" on that piece of content to execute the addTestRating.js script. Every time you run it, a new rating (with a random value) will be created as a child node of that content.

At this point, you should see the ratings on the content's property sheet as child associations but the average won't be set until the behavior code is in place.

Note, if you want the log messages to show up you have to set log4j.logger.org.alfresco.repo.jscript to DEBUG in log4j.properties.

Write the custom behavior

The custom behavior is implemented as a Java class that implements the interfaces that correspond to the policies to which we want to bind. In this example, the two policy interfaces are: NodeServicePolicies.OnDeleteNodePolicy and NodeServicePolicies.OnCreateNodePolicy so the class declaration is:

```
public class Rating
         implements NodeServicePolicies.OnDeleteNodePolicy,
                                 NodeServicePolicies.OnCreateNodePolicy {
```

Listing 8: The class declaration for the Rating bean.

The class has two dependencies that Spring will handle for us. One is the NodeService which will be used in the average calculation logic and the other is the PolicyComponent which is used to bind the behavior to the policies.

```
         // Dependencies
         private NodeService nodeService;
         private PolicyComponent policyComponent;

         // Behaviours
         private Behaviour onCreateNode;
```

```
        private Behaviour onDeleteNode;
```

Listing 9: Dependency and behavior declaration in the Rating bean.

At some point Alfresco has to know that the behavior needs to be bound to a policy. In the JavaScript example we'll see how to do that in a Spring bean config file. In this example, we'll create a method called “init” to handle the binding. The init method will get called when Spring loads the bean.

```
public void init() {

        // Create behaviours
        this.onCreateNode = new JavaBehaviour(
                            this,
                            "onCreateNode",
                            NotificationFrequency.TRANSACTION_COMMIT);

        this.onDeleteNode = new JavaBehaviour(
                            this,
                            "onDeleteNode",
                            NotificationFrequency.TRANSACTION_COMMIT);

        // Bind behaviours to node policies
        this.policyComponent.bindClassBehaviour(
                    Qname.createQName(NamespaceService.ALFRESCO_URI,
"onCreateNode"),
                    Qname.createQName(
                              SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
                              SomeCoModel.TYPE_SC_RATING),
                    this.onCreateNode);

        this.policyComponent.bindClassBehaviour(
                    QName.createQName(NamespaceService.ALFRESCO_URI,
"onDeleteNode"),
                    Qname.createQName(
                              SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
                              SomeCoModel.TYPE_SC_RATING),
                    this.onDeleteNode);
}
```

Listing 10: The Rating bean's init method binds the bean's methods to specific policies.

The first thing to notice here is that you can decide when the behavior should be invoked by specifying the appropriate NotificationFrequency. Besides TRANSACTION_COMMIT, other choices include FIRST_EVENT and EVERY_EVENT.

Also note that there are a few different overloaded methods for bindClassBehaviour (note the UK spelling). In this case we're binding the Qname of a behavior to the Qname of our type (“Rating”) and telling Alfresco to call the onCreateNode and onDeleteNode behaviors of our bean.

There are also additional bind methods for associations (bindAssociationBehaviour) and properties (bindPropertyBehaviour) that you should use depending on the type of policy you are binding to.

Next we need to write the methods required by the two policy interfaces. Regardless of whether a ratings node is created or deleted, we need to recalculate the average. So our onCreateNode and onDeleteNode methods simply call computeAverage and pass in the rating node reference.

```
public void onCreateNode(ChildAssociationRef childAssocRef) {
        computeAverage(childAssocRef);
}

public void onDeleteNode(ChildAssociationRef childAssocRef, boolean
isNodeArchived) {
        computeAverage(childAssocRef);
}
```

Listing 11: The onCreateNode and onDeleteNode methods implement the behavior.

The computeAverage method asks the child (the rating) for its parent node reference (the whitepaper, for example) and asks the parent for a list of its children. It iterates over the children, computes an average, and sets the average property on the content.

```
public void computeAverage(ChildAssociationRef childAssocRef) {

        // get the parent node
        NodeRef parentRef = childAssocRef.getParentRef();

        // check the parent to make sure it has the right aspect
        if (nodeService.hasAspect(
                parentRef,
                Qname.createQName(SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
                                SomeCoModel.ASPECT_SC_RATEABLE))) {
                // continue, this is what we want
        } else {
                return;
        }

        // get the parent node's children
        List<ChildAssociationRef> children =
nodeService.getChildAssocs(parentRef);

        // iterate through the children to compute the total
        Double average = 0d;
        int total = 0;
        for (ChildAssociationRef child : children) {
                int rating = (Integer)nodeService.getProperty(
                        child.getChildRef(),

        Qname.createQName(SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
                        SomeCoModel.PROP_RATING));
                total += rating;
        }

        // compute the average
```

```
        average = total / (children.size() / 1.0d);

        // store the average on the parent node
        nodeService.setProperty(
                parentRef,
                QName.createQName(SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
                        SomeCoModel.PROP_AVERAGE_RATING),
                average);

        return;
}
```

Listing 12: The computeAverage method calculates the average rating for a piece of content.

The only items remaining, then, are the getters and setters for the NodeService and PolicyComponents, but I'll leave those out here.

Configure a Spring bean

The last step before we test is to configure the behavior class as a Spring bean. The bean config can go in any context file. If you had several behaviors it might make sense to put them in their own. For this example, we'll use the someco-model-context.xml file. Add the following before the closing "</beans>" tag:

```
<bean id="ratingBehavior" class="com.someco.behavior.Rating" init-method="init">
        <property name="nodeService">
                <ref bean="nodeService" />
        </property>
        <property name="policyComponent">
                <ref bean="policyComponent" />
        </property>
</bean>
```

Listing 13: The updated someco-model-context.xml file includes the Spring bean config for the Rating behavior bean.

Build, deploy, restart and test

Modify the build.properties file to match your environment, then use Ant to run the Deploy target. The code will be compiled, JAR'd, and unzipped on top of the existing Alfresco installation.

Hopefully, Alfresco will start up error free. If so, log in and execute the addTestRating.js script against a piece of content. The average should now get computed.

Finally, delete one of the test ratings by editing the properties for a piece of rated content and clicking the trash can icon. When you click "ok" to save your changes, the average rating should get recalculated. If you delete all of the ratings, the average should get set to 0.

Note: If you are running the downloaded sample code instead of creating it by following along, set com.someco to DEBUG in log4j.properties to display the logger messages. I've omitted them in the

code samples for brevity.

JavaScript Example

We've seen how to implement the average rating calculation behavior in Java, but what if you wanted to implement the behavior using JavaScript instead? Behaviors can be implemented in JavaScript and bound to policies through Spring. Let's re-implement the Rating bean using JavaScript.

The high-level steps we're going to follow are:

1. Write the custom behavior as one or more server-side scripts.
2. Configure a Spring bean to bind the scripts to the appropriate policies.
3. Deploy, restart and test.

Write the custom behavior as a server-side JavaScript

For this example I'm going to shamelessly steal a JavaScript file that is part of the Alfresco source and then tweak it. The original script is used by Alfresco to test Policy functionality. (As a side note, the test code that is buried in the Alfresco source tree is a great resource for example code).

We're actually going to write three scripts. The onCreateRating.js and onDeleteRating.js scripts will be bound to the onCreateNode and onDeleteNode policies respectively. The rating.js script will contain our average rating calculation logic and will be imported by the other two scripts using the new "<import>" tag that became available with the 2.1 release.

In this example the scripts are going to reside as part of the web application rather than being uploaded to the repository. Nothing requires that this be the case—we've written one script already that resides in the repository so I thought we'd change it up a little. This approach does have the downside that modifications require a re-deploy. Of course, depending on whether or not you want end users monkeying around with your script, this could be a benefit in a production environment.

In your Eclipse project, create a "scripts" directory under the src/alfresco/extension folder and add a file called onCreateRating.js with the following content:

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

// Have a look at the behaviour object that should have been passed
if (behaviour == null) {
    logger.log("The behaviour object has not been set.");
    scriptFailed = true;
}

// Check the name of the behaviour
```

```
if (behaviour.name == null && behaviour.name != "onCreateNode") {
        logger.log("The behaviour name has not been set correctly.");
        scriptFailed = true;
} else {
        logger.log("Behaviour name: " + behaviour.name);
}

// Check the arguments
if (behaviour.args == null) {
        logger.log("The args have not been set.");
        scriptFailed = true;
} else {
        if (behaviour.args.length == 1) {
                var childAssoc = behaviour.args[0];
                logger.log("Calling compute average");
                computeAverage(childAssoc);
        } else {
                logger.log("The number of arguments is incorrect.");
                scriptFailed = true;
        }
}
```

Listing 14: The onCreateRating.js is invoked when a new rating node is created.

The code for onDeleteRating.js is identical with the exception of the behavior name and the number of arguments expected (2 instead of 1) so I won't duplicate the listing here.

As in our Java example, both call the same computeAverage function. Create a third JavaScript file called rating.js and add the content as follows:

```
//calculate rating
function computeAverage(childAssocRef) {
   var parentRef = childAssocRef.parent;

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

   // get the parent node's children
   var children = parentRef.children;

   // iterate through the children to compute the total
   var average = 0.0;
   var total = 0;

   if (children != null && children.length > 0) {
      for (i in children) {
```

```
         var child = children[i];
         var rating =
child.properties["{http://www.someco.com/model/content/1.0}rating"];
         total += rating;
      }

      // compute the average
      average = total / children.length;
   }

   // store the average on the parent node
parentRef.properties["{http://www.someco.com/model/content/1.0}averageRating"] =
average;
   parentRef.save();

   return;
}
```

Listing 15: The rating.js file contains the shared computeAverage function.

As you can see, this is the same logic we used in the Java example modified to follow the Alfresco JavaScript API syntax.

Configure a Spring bean to bind the script to the appropriate policies

In the Java example, we used an init method on the Rating bean to make calls to the binding method of the PolicyComponent. The JavaScript example doesn't do that. Instead, it uses Spring to associate the JavaScript files with the onCreateNode and onDeleteNode policies.

Edit the someco-model-context.xml file. Comment out the bean config we used for the Java example and add two new bean configs below it for the JavaScript behavior code—one for the create and one for the delete:

```
<bean id="onCreateRatingNode"
      class="org.alfresco.repo.policy.registration.ClassPolicyRegistration"
      parent="policyRegistration">
      <property name="policyName">
            <value>{http://www.alfresco.org}onCreateNode</value>
      </property>
      <property name="className">
            <value>{http://www.someco.com/model/content/1.0}rating</value>
      </property>
      <property name="behaviour">
            <bean class="org.alfresco.repo.jscript.ScriptBehaviour"
                  parent="scriptBehaviour">
                  <property name="location">
                        <bean
class="org.alfresco.repo.jscript.ClasspathScriptLocation">
                              <constructor-arg>
                                    <value>alfresco/extension/scripts/onCreateRating.js</value>
                              </constructor-arg>
```

```
                    </bean>
                  </property>
            </bean>
          </property>
</bean>
```

Listing 16: The Spring bean config used to bind the onCreateNode policy to the onCreateRating.js script. The config for the onDeleteNode policy is not shown.

Deploy, restart and test

If you haven't already, make sure you've set log4j.logger.org.alfresco.repo.jscript to DEBUG in log4j.properties or you won't see any of the logger output.

Use Ant to run the Deploy target.

Assuming Alfresco starts up error-free, you should be able to use addTestRating.js to create Rating nodes which should trigger the onCreateRating JavaScript. Deleting ratings will trigger the onDeleteRating JavaScript. In either case, the average rating should get calculated as it did with the Java example.

Conclusion

This article has shown how to bind custom behavior to Alfresco policies. Specifically, we implemented a custom “rateable” aspect and a custom “rating” type that can be used to persist user ratings of content stored in the repository. The custom behavior is responsible for calculating the average rating for a piece of content any time a rating is created or deleted. The article showed how to implement the average rating calculation behavior in Java as well as JavaScript.

Where to find more information

- The complete source code that accompanies this article is available here from ecmarchitect.com.
- The previous article that discusses custom content models is called “Working with Custom Content Types” and is available at ecmarchitect.com.
- The Alfresco SDK comes with a custom behavior example that's a little more complex than the one presented here. The “SDK Custom Aspect” project implements a “hit counter” that increments every time a piece of content is read.
- 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.

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.