ecmarchitect.com

Alfresco Developer: Advanced Workflows

November, 2007

Jeff Potts

Alfresco Developer: Advanced Workflows

November 2007

Jeff Potts

Table of Contents

Alfresco Developer: Advanced Workflows

November 2007

Jeff Potts

Introduction

This article is about the advanced workflow functionality available in Alfresco through its embedded JBoss jBPM workflow engine. First, because "workflow" can mean different things to different people, I'll talk about my definition of the term. Then, I'll dive into fundamental jBPM concepts that will help you understand how processes are defined and how the workflow engine actually works. (Those who use jBPM separate from Alfresco might even find this section helpful). Once that foundation is in place, I'll walk through an example that features many of the different concepts.

The example continues the "SomeCo" examples covered in earlier papers. In it, we'll implement a business process that helps SomeCo route whitepapers for review and approval by internal as well as external parties.

What is a workflow?

When Alfresco released version 1.4 of the product, they made a huge leap forward in enterprise readiness. That was the release when Alfresco embedded the JBoss jBPM engine into the product which meant that enterprises could route Alfresco repository content through complex business processes. A content repository without the ability to facilitate business processes that produce, consume, or transform the content within it is little more than a glorified file server, so this was a welcome addition.

But before we geek out on the wonders of graph based execution languages let's agree on what the term *workflow* means. Generically, a workflow is "a reliably repeatable pattern of activity enabled by a systematic organization of resources...that can be documented and learned"[1]. The term has been around since people started studying the nature of work in the early 20th century in an effort to streamline

1 http://en.wikipedia.org/wiki/Workflow

manufacturing processes.

In fact, in the world of ECM, it is sometimes helpful to think of an assembly line or manufacturing process when thinking about how content flows through an organization. Content is born of raw material (data), shaped and molded by one or more people (collaboration) or machines (systems), reviewed for quality, and delivered to consumers. Content may go through a single process or many sub-processes. Content may take different routes through a process based on certain aspects of that content. The output of an organization or department of knowledge workers is essentially the content that comes rolling off the assembly line (the collection of workflows that define that organization's business processes).

Although not always formalized or automated, almost everyone in modern society has been involved in a workflow in some way:

- When you submit an insurance claim, you are initiating a workflow.
- If you witness drunk and disorderly conduct on an airline flight and are asked to provide a statement to the airline, you are participating in a workflow. (Seriously, it happens more often than you'd think).
- When you check on the status of your loan application, you are asking for metadata about a running workflow.
- When someone brings you a capital request that requires your approval because it is over a certain dollar amount, a characteristic of that request (the dollar amount) has triggered a decision within the workflow that routes the capital request to you.
- When you give the final approval for a piece of web content to be published, it is likely you are completing a workflow.

As varied as these examples are, all of them have a couple of things in common that make them relevant to ECM: (1) They are examples of human-to-human and, in some cases, human-to-machine interaction and (2) They are content- or document-centric.

These are two very important characteristics that help clarify the kind of workflow we're talking about. There are standalone workflow engines (in fact, jBPM is one of them) that can be used to model and execute all sorts of “repeatable patterns of activity”, with or without content, but in the ECM space, the patterns we care most about are those that involve humans working with content.[1]

Workflow options

Some of you are saying, “You're right. Workflows are everywhere. I could really streamline my organization by moving processes currently implemented with email, phone calls, and cubical drive-bys into a more formalized workflow. What are my options?” Let's talk about three: Roll your own, Standalone workflow engines, and Embedded workflow engines.

Roll your own. People are often tempted to meet their workflow requirements with custom code. Very basic systems might be able to get by with a single flag on a record or an object that declares the status of the content like “Draft” or “In Review” or “Approved”. But flags only capture the “state” of a piece of content. If you want to automate *how* content moves from state to state, the coding and maintenance becomes more complex. Sure, you can write code as part of your application that knows that once Draft documents are submitted for review, they need to go to Purchasing first and then to Finance, if and only if the requested cash outlay is more than $10m but do you really want to?

People intent on rolling their own workflow often realize the maintenance problem this creates, so they create an abstraction used to describe the flow from state-to-state that keeps them from embedding that logic in compiled code. Once they've done that, though, they've essentially created their own proprietary workflow engine that no one else in the world knows how to run or maintain. And with all of the open source workflow engines available, why would you want to do that? So the “roll your own” option is really not recommended for any but the most basic workflow requirements.

Standalone engines. There are a number of standalone workflow engines—sometimes more broadly referred to as BPM (Business Process Management)—both open source and proprietary. These are often extremely robust and scalable solutions that can be used to model, simulate, and execute any process you can think of from high-volume loan processing to call center queue management. Often, these workflow engines are implemented in conjunction with a rules engine which lets business users have control over complicated if-then-else decision trees.

Standalone engines are most appropriate for extremely high volume or exceedingly complex solutions

1 Of course document-centric workflows may include fully automated steps and machine-to-machine interactions—the point is that document-centric workflows in which humans review, approve, or collaborate in some way are in the scope of the discussion while processes which run lights-out system-to-system orchestration or integration are not.

involving multiple systems. Another good use for standalone engines is when you are developing a custom application that has workflow requirements. Standalone engines can usually talk to any database or content management repository you might have implemented, but they won't be as tightly integrated into the content management system's user interface as the workflow engine built-in to the CMS. For this reason, for content-centric solutions that operate mostly within the scope of the CMS, it is usually less complicated (and less costly) to use the workflow engine embedded within the CMS, provided it has enough functionality to meet the business' workflow requirements.

Embedded workflow engines. Almost every CMS available today, whether open source or proprietary, has a workflow engine of some sort embedded within it. However, the capability of each of these vary widely. If you are in the process of selecting a CMS and you already know the kind of workflow requirements you'll face, it is important to understand the capabilities of the workflow engine embedded within the systems you are considering before making a final selection.

The major benefit of leveraging an embedded workflow engine is the tight level of integration for users as well as developers. Users can initiate and interact with workflows without leaving the CMS client. Typically, developers customizing or extending the CMS can work with workflows using the core CMS API.

Alfresco workflow

Alfresco has two options for implementing workflows within the product. For very simplistic workflows, non-technical end-users can leverage Alfresco's Basic Workflow functionality. For more complex needs, Alfresco leverages the embedded JBoss jBPM engine to provide Advanced Workflow capability.

Basic workflows are a nice end-user tool. You should know how they work and what the features and limitations are so you can make good decisions about when to use them. Basic workflows use folders and a “forward step/backward step” model to implement serial processes. When a piece of content is dropped in a folder, a rule is triggered that associates a “forward step” and a “backward step” (one or the other or both) with the content. These steps are tied to Alfresco actions such as “Set a property” or “Move the content to a specified folder”. End users can then click on the appropriate step for a given piece of content.

For example, suppose we have a simple submit-review-approve process in which content is submitted, then reviewed, then approved or rejected. One way to implement this with basic workflows is to use three folders—let's say they are called "Draft", "In Review", and "Approved"—each of which have a rule set that applies a basic workflow. The workflow for content in the Draft folder would have a single forward step labeled "Submit" and its action would move content to the "In Review" folder and send an email to the approver group. The "In Review" folder would have a workflow in which the forward step would be labeled "Approve" and it would copy the content to an "Approved" folder. The backward step would be labeled "Reject" and its action would move the content back to the "Drafts" folder.

You can see that basic workflows are useful, but limited with regard to the complexity of the business processes they can handle.

Although we haven't yet covered the detailed capabilities of Alfresco advanced workflows, I thought it would be a good idea to compare basic and advanced workflows at a high level now so we can leave the topic of basic workflows behind and spend the rest of the article on advanced workflows:

Alfresco basic workflows...	**Alfresco advanced workflows...**
● Are configurable by non-technical end-users via the Alfresco web client ● Leverage rules, folders, and actions ● Can only handle processes with single-step, forward and/or backward, serial flows ● Do not support decisions, splits, joins, or parallel flows ● Do not maintain state or metadata about the process itself	● Are defined by business analysts and developers via a graphical Eclipse plug-in or by writing XML ● Leverage the power of the embedded JBoss jBPM workflow engine ● Can model any business process including decisions, splits, joins, parallel flows, sub-processes, wait states, and timers ● Can include business logic written either in JavaScript or Java, either of which can access the Alfresco API ● Maintain state and process variables (metadata) about the process itself)

Now that you understand the definition of workflow in the context of ECM, some of the options for implementing workflow requirements, and the options within Alfresco specifically, it's time to dive in to the gory details of the jBPM engine.

jBPM concepts

This section is a bit like vegetables—somewhat ugly and unpleasant tasting but ultimately good for you (no offense to my vegetarian friends). While it is possible to modify the out-of-the-box workflows to suit your needs without understanding the details of how the jBPM engine works, I recommend you pinch your nose and take a bite. You'll thank me later when you are ready to do something complex and completely unlike any of the out-of-the-box examples.

What is the jBPM engine?

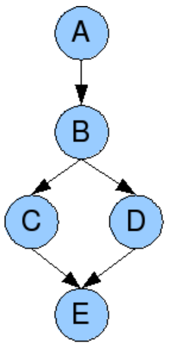
JBoss jBPM is an open source, standalone workflow engine[1]. It can run in any servlet container—it doesn't require JBoss Application Server. The jBPM engine is responsible for managing deployed processes, instantiating and executing processes, persisting process state and metadata to a relational database (via Hibernate), and tracking task assignment and task lists.

Process definitions

jBPM is built on the idea that any process can be described as a graph or a set of connected nodes. Workflows are described with "process definitions" using an XML-based language called Java Process Definition Language (jPDL). jPDL is one example of a graph based execution language. Others include BPEL for service orchestration and SEAM pageflow.

In jPDL, each node represents a step in a workflow. Connections between nodes signify the transition from one step to another. Consider the figure below. We see a relatively simple process with 5 nodes. By looking at the transitions we see that the path of execution will always be from node A to node B. Node B has two outgoing paths. One path is to node C and the other to node D. The paths converge on node E. Note that from the diagram it is impossible to tell which path will be taken. It's also possible that both could be followed simultaneously.

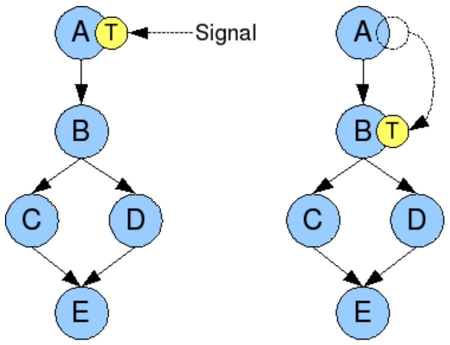
1 JBoss rightly calls jBPM a "platform for graph based execution languages". "Workflow" as it is defined here is one of several different domains that can be addressed with a graph based execution language. So jBPM is more than *just* a workflow engine, but for this article, that's all we care about.



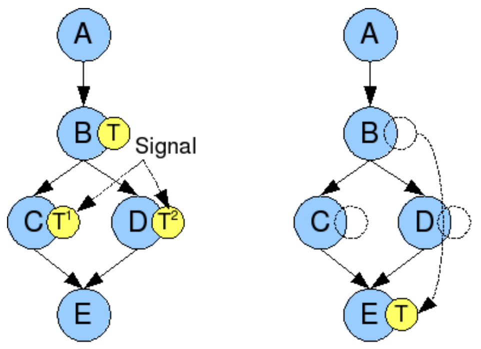
The node's type determines how it behaves. So you might have a node that splits execution into several paths (like node B) or a node that joins multiple paths of execution into a single path (like node E). You might have a node that calls some other system and waits to hear back from that system before proceeding. We'll explore the different node types available and briefly talk about creating your own node types shortly.

Tokens

A *token* is like the "You Are Here" flag for a process. The token moves from node-to-node as the process is executed. But it doesn't move on its own. Tokens only move when they are *signaled*. Let's look at an example. In the figure below, when we execute the process, the token is at Node A. If we signal the token, it will take the only path available to it which is to move to Node B where it will wait for a signal. From Node B there are multiple paths available. If we signal the token without specifying which path to take, it will take a default path. Or, we could tell it which path to take when we send it the signal.



Tokens can have children. For example, in the figure we just looked at, suppose we wanted to follow **both** paths. In that case, as shown in the figure below, the token would spawn two child tokens, one for the path from Node B to Node C and one from Node B to Node D. When the paths converge, the child tokens go away and the parent token resumes.



Why should we care about tokens? I'm sure there are lots of reasons but a few that come to mind are (1) when you write your own node types you are responsible for how tokens are signaled (2) process variables (more on those shortly) are scoped to a token, and (3) if a process ever gets stuck you may need to signal the token manually.

Node types

I mentioned earlier that the node type determines the node's behavior which begs the question: What node types are available?

Node Type	Description
Start-state	Only one allowed per process definition. Only outgoing transitions are allowed.
Fork	Spawns multiple concurrent paths of execution.
Join	Joins multiple paths into a single path. Becomes a wait state until all tokens have reached the join.
Decision	Choice between multiple paths of execution. We'll see examples of how to implement the logic for a decision later in the article.
Node	Plain old node. Good for containing an action which might execute business logic.
State	A wait state. Execution does not proceed until the node is explicitly signaled.
Process State	Executes a sub-process. Behaves as a wait state while the sub-process executes.

Node Type	Description
Task Node	A node that contains one or more tasks assigned to humans.
End State	Only one allowed per process definition. Only incoming transitions are allowed.

It is important to note that if your application has a requirement that isn't addressed by one of the out-of-the-box node types you can add your own. And, if the behavior of a node type isn't exactly what you need, you can extend it.

If we implemented the diagram shown previously in which node B is a fork and node E is a join, the jPDL would look like the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<process-definition xmlns="" name="simple-process">
   <start-state name="start">
      <transition name="" to="Node A"></transition>
   </start-state>
   <node name="Node A">
      <transition name="" to="Node B"></transition>
   </node>
   <fork name="Node B">
      <transition name="" to="Node C"></transition>
      <transition name="tr2" to="Node D"></transition>
   </fork>
   <node name="Node C">
      <transition name="" to="Node E"></transition>
   </node>
   <node name="Node D">
      <transition name="" to="Node E"></transition>
   </node>
   <join name="Node E">
      <transition name="" to="end1"></transition>
   </join>
   <end-state name="end1"></end-state>
</process-definition>
```

Actions

So far we've seen that a process can be modeled as a collection of nodes connected via paths or transitions. A common requirement is to be able to execute some code or business logic at certain points within the process. For example, maybe you want to send an email or perhaps you want to increment a counter that keeps track of how many times a node has been executed. *Actions* are the hooks that make this happen.

What triggers an action? As a token propagates through the graph it fires events. Examples include things like entering a node, leaving a node, or following a transition.

Actions can be a beanshell expression or a Java class. In the context of Alfresco, actions can also be written using JavaScript which can make use of the Alfresco JavaScript API.

In the example below we see a task-node named “review” with a task called “scwf:reviewTask”. The task has script associated with its “task-create” event. Don't worry about what the script is doing for now. This is just an example of associating business logic with an event.

```
    <task-node name="review">
        <task name="scwf:reviewTask" swimlane="reviewer">
            <event type="task-create">
                <script>
                   if (bpm_workflowDueDate != void) taskInstance.dueDate = 
bpm_workflowDueDate;
                   if (bpm_workflowPriority != void) taskInstance.priority = 
bpm_workflowPriority;
                </script>
            </event>
         ...
```

Here's an example that uses the “action” tag to associate Alfresco JavaScript with a transition. This particular JavaScript runs an action against every piece of content in the workflow package.

```
<task-node name="addAspect">
    <transition name="transToReview" to="review">
      <action class="org.alfresco.repo.workflow.jbpm.AlfrescoJavaScript">
        <script>
          var scAspectQName = "{http://www.someco.com/model/content/1.0}webable";
          var addAspect = actions.create("add-features");
          addAspect.parameters["aspect-name"] = scAspectQName;
          for (var i = 0; i &lt; bpm_package.children.length; i++) {
            if (!bpm_package.children[i].hasAspect(scAspectQName)) {
              addAspect.execute(bpm_package.children[i]);
            }
          }
        </script>
      </action>
    </transition>
</task-node>
```

Note that in the Alfresco JavaScript example, the jPDL points to a Java class. In this case, Alfresco has provided an action class that executes Alfresco JavaScript. But you can write your own action classes with Java as well as we shall soon see in the example.

Process Variables

Often, there is metadata about a process that needs to be tracked. A due date or a priority flag are two examples. A due date isn't really a property of the content being routed through the workflow—it's a property of the process itself. jBPM gives us the ability to store this kind of data as part of the running process through *process variables.*

Process variables are name-value pairs that will get persisted with the rest of the process state. Variables can be scoped to a specific token. By default, they are scoped to the root token so they are effectively global.

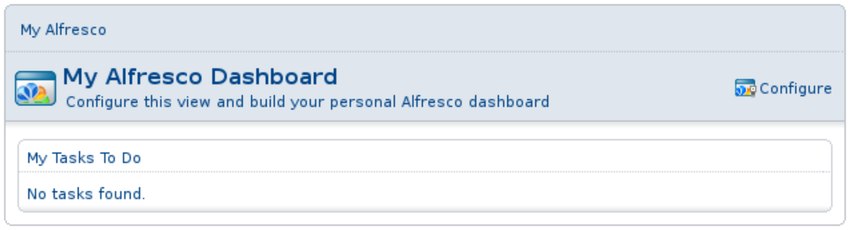
In the example below, we use the "script" element to set a variable called scwf_tempCnt equal to 0 when the token enters the node.

```
<event type="node-enter">
         <script>
                  <variable name="scwf_tempCnt" access="write"/>
                  <expression>
                           scwf_tempCnt = 0;
                  </expression>
         </script>
</event>
```

Elsewhere in the process we could read the value of the variable with an expression like: `#{scwf_tempCnt}`.

Tasks

A task is a step in a workflow that requires human interaction. jBPM maintains a list of tasks assigned to each participant. How users interact with the task list is up to each application. In Alfresco, a dashlet displays a to-do list for the currently logged in user. As users complete their tasks the tasks are removed from the to-do list. An empty to do list is shown below.


My Alfresco

My Alfresco Dashboard

Configure this view and build your personal Alfresco dashboard

My Tasks To Do

No tasks found.

Task Assignment (Swimlanes, "Initiator" swimlane, actors, pooled actors)

If tasks are steps a human performs, how do tasks get associated with the people who need to perform them (actors)? One of the child elements of the "task" element is "assignment". The assignment element points to a Java class that is an instance of AssignmentHandler. It handles assigning the task to an actor. Alfresco saves us some work here—they provide an AssignmentHandler out-of-the-box. We'll see an example of how it can be used momentarily.

Often, a process has the notion of a role in which multiple tasks during the process get assigned to the same actor playing that role. For example, suppose we are defining a process that contains multiple tasks performed by "marketing". Rather than assign the marketing group or individual repeatedly to each task, it would be nice if we could make the assignment once and then tell the other tasks to use the same assignment. In jBPM this is implemented through *swimlanes.* An actor can be assigned to a swimlane, and then all tasks that need to be performed by the same actor refer to the swimlane. The snippet below shows what this looks like in jPDL.

```
<swimlane name="marketing">
   <assignment class="org.alfresco.repo.workflow.jbpm.AlfrescoAssignment">
      <pooledactors>#{people.getGroup('GROUP_marketing')}</pooledactors>
   </assignment>
</swimlane>

<swimlane name="engineering">
   <assignment class="org.alfresco.repo.workflow.jbpm.AlfrescoAssignment">
      <pooledactors>#{people.getGroup('GROUP_engineering')}</pooledactors>
   </assignment>
</swimlane>

<task-node name="marketingReview">
   <task name="scwf:marketingReview" swimlane="marketing"></task>
   <transition name="transToLegal" to="engineeringReview"></transition>
   <transition name="transToRejected" to="rejected"></transition>
</task-node>
```

```
<task-node name="engineeringReview">
   <task name="scwf:engineeringReview" swimlane="engineering"></task>
   <transition name="transToMarketing" to="finalMarketingReview"></transition>
   <transition name="transToRejected" to="rejected"></transition>
</task-node>

<task-node name="finalMarketingReview">
   <task name="scwf:finalMarketingReview" swimlane="marketing"></task>
   <transition name="transToApproved" to="approved"></transition>
   <transition name="transToRejected" to="rejected"></transition>
</task-node>
```

In this example we're defining two swimlanes: “engineering” and “marketing”. The engineering swimlane is assigned to the “engineering” group. The marketing swimlane is assigned to the “marketing” group. Each of the three tasks is assigned to the appropriate swimlane using the “swimlane” attribute of the “task” element.

A special swimlane exists called “initiator”. This swimlane always has the actor that started the workflow. If you want to assign one or more tasks to the initiator, add the initiator swimlane to the process definition like this:

```
   <swimlane name="initiator" />
```

and then use the swimlane attribute to make the assignment.

If you have already used Alfresco Advanced Workflows you know that in the sample workflows the initiator assigns the review step to a user or group by selecting a value in the user interface. We'll see how that value makes its way into the task assignment a little later.

Pooled Actors

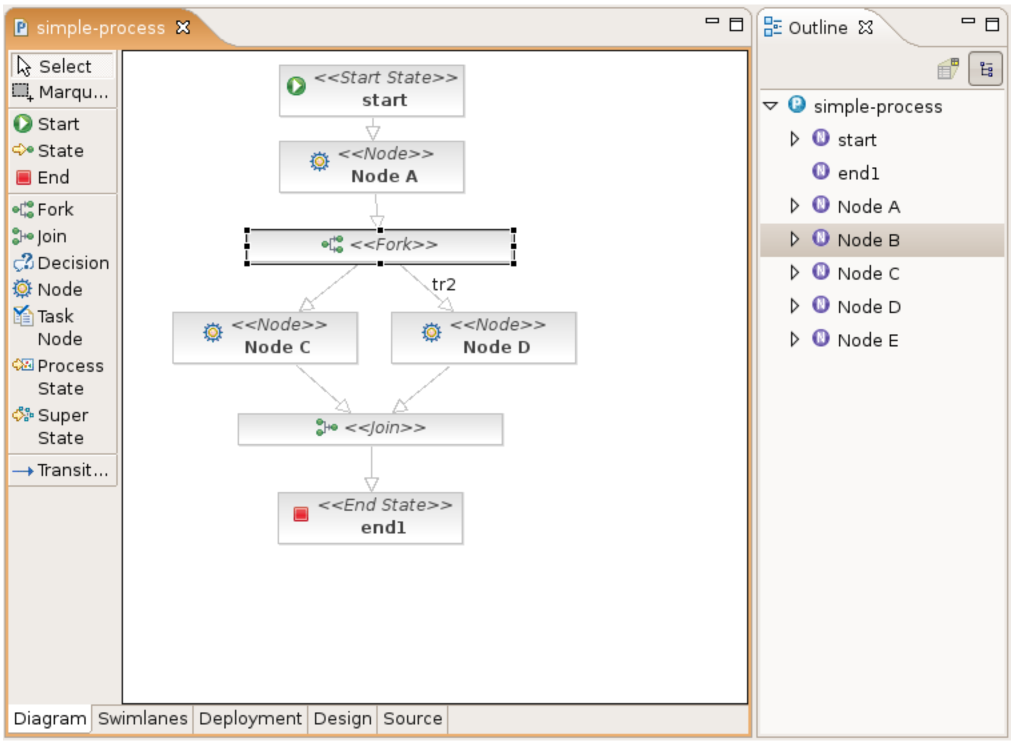
When defining a business process it is important to understand how the participants in the process will do the work. One specific area that needs to be considered is whether to use “pooled actors” for a given task. Suppose, for example, you assigned a task to a group of ten people. You could iterate through the group and assign a task to each and every member of the group and then not consider the task complete until all actors have taken action. An alternative is to use pooled actors. Using a pool, all members of a group are notified of the task, but as soon as one actor takes “ownership” of the task, it is removed from everyone else's to do list. The owner can then either complete the task or return it to the pool. If it is returned to the pool, all members of the group see the task in their to do list until another person takes ownership or completes the task. To use pooled actors, use the “pooledactors” child element of the “assignment” element instead of the “actor” element.

The decision to use pooled actors or not depends entirely on the business process—there is no preferred approach.

jBPM Process Designer

One of the nice things about jBPM is that a graphical tool is available as an Eclipse plug-in for creating and deploying process definitions called the Graphical Process Designer (GPD). Marketing guys for workflow tools love to say things like, “Using the graphical process designer, business analysts can create advanced workflows without writing code!”. While it is a handy tool, particularly when you are first laying out a process, in the real world, you'll find yourself switching to the source tab fairly quickly.

Here's what the “Diagram” tab looks like for the five node process we defined earlier.



The Diagram tab is used to lay out the process by selecting node types from the palette and clicking on the canvas. The Transition tool is used to connect the nodes. Node properties can be edited in the tree view of the process or nodes can be double-clicked to open a properties editor. At any time you can switch to the Source tab to see and edit the underlying XML.

Deploying processes

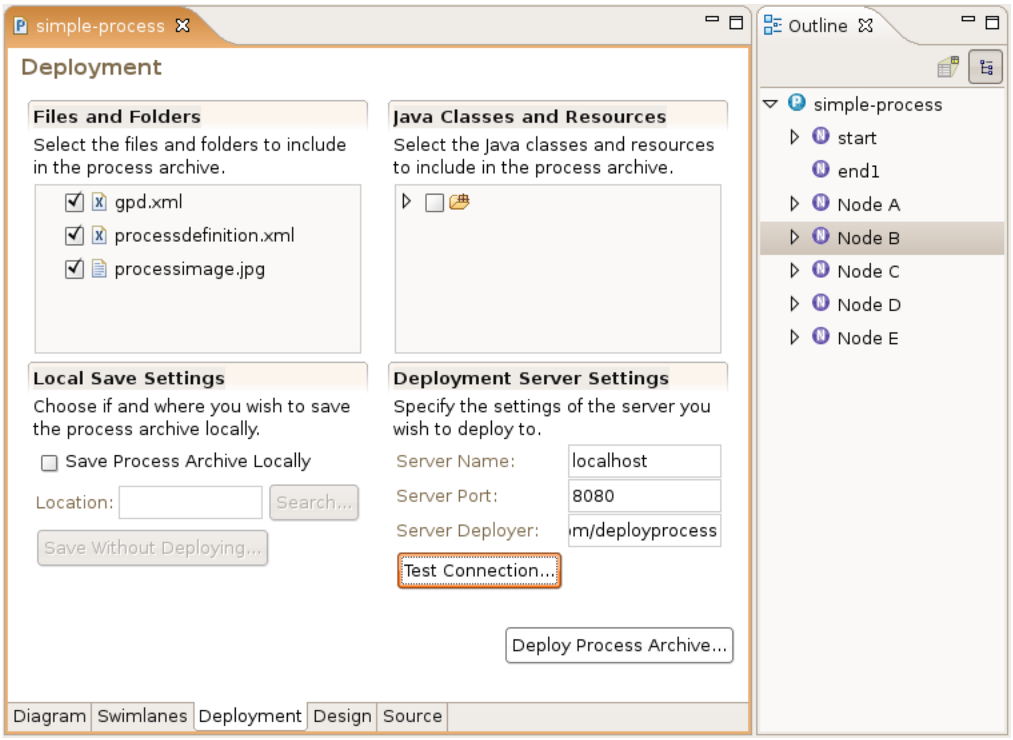
If you look at the out-of-the-box process definitions you'll notice that Alfresco chose to put all process definitions in the same directory. This is a bit annoying to me because it breaks the GPD deployment tool. So, when organizing your work, I recommend you not follow Alfresco's example and instead put each process definition in its own directory and name the jPDL file “processdefinition.xml”.

If you are writing complex processes that depend on resources beyond just the definition itself, you can optionally package those dependencies along with your process definition into a Process Archive (PAR) file (Like a JAR file, a PAR file is just a zip). The “Java Classes and Resources” section of the Deployment tab is used to associate dependencies with the process so the GPD knows to include them with the PAR. If you choose not to create a PAR, dependent classes can reside anywhere on the Alfresco classpath, preferably right along side your other classes in the extension directory.

Once you've defined your process and decided how to package it, the process definition has to be deployed to the jBPM runtime. Let's look at two options for deploying a jBPM process definition to Alfresco: (1) use the deployment tab in the GPD or (2) deploy the process definition alongside your other web client customizations in the Alfresco extension directory.

The deployment tab is the easiest option and is very handy when you are still developing and debugging the process because you don't need to restart the application server between deployments. The graphic below shows a screenshot of the deployment tab.

The Deployment Server Settings tell the plug-in how to communicate with the jBPM deployment process. The Test Connection button will verify the plug-in is able to communicate to the deployer process.


simple-process
Outline
Deployment
Files and Folders
Select the files and folders to include in the process archive.
gpd.xml
processdefinition.xml
processimage.jpg
Java Classes and Resources
Select the Java classes and resources to include in the process archive.
Local Save Settings
Choose if and where you wish to save the process archive locally.
Save Process Archive Locally
Location:
Search...
Save Without Deploying...
Deployment Server Settings
Specify the settings of the server you wish to deploy to.
Server Name:
localhost
Server Port:
8080
Server Deployer:
m/deployprocess
Test Connection...
Deploy Process Archive...
Diagram
Swimlanes
Deployment
Design
Source
simple-process
start
end1
Node A
Node B
Node C
Node D
Node E

When you click “Deploy Process Archive...” the process will be uploaded to the jBPM engine. If the process has already been deployed it will get versioned. If either the Test Connection button or the Deploy Process Archive button result in an error, check the application server log for clues.

Another deployment alternative is to use a Spring bean. The snippet below shows how to point to a single process definition.

```
<bean id="extension.workflows.workflowBootstrap" parent="workflowDeployer">
   <property name="workflowDefinitions">
      <list>
         <props>
            <prop key="engineId">jbpm</prop>
            <prop key="location">alfresco/extension/workflows/simple-
process/processdefinition.xml</prop>
            <prop key="mimetype">text/xml</prop>
            <prop key="redeploy">true</prop>
         </props>
      </list>
   </property>
</bean>
```

If you have multiple workflows to deploy using this approach use multiple “props” elements.

The **engineId** must be set to “jbpm”. Maybe at some point other engines will be supported but for now, jBPM is the only choice.

The **location** is any location on the classpath, but to be consistent with how customizations are deployed, I recommend standardizing on the Alfresco extension directory or, as in this example, a directory called “workflows” within that.

The **mimetype** setting is “text/xml” when deploying the processdefinition.xml file. If you choose to deploy the process as a PAR, set the location to the path of the PAR file and set the mimetype to “application/zip”.

The redeploy flag tells Alfresco whether or not it should automatically redeploy the process on startup. During development, if you are deploying your processes via Spring you probably want this to be set to true. Once you get to production, set it to false to avoid needlessly creating new versions of the process definition every time the server is restarted.

There are other deployment options: jBPM ships with an Ant task called “deploypar”, the jBPM console (more on that later) can be used, and processes can be deployed programmatically using the jBPM API. This article won't cover any of these alternatives, but it's nice to know they are there if you need them.

You may be wondering what happens to running workflow instances when a new version of the process definition is checked in. The answer is that jBPM handles that—it makes sure that running workflows continue to run with their original process definition. All new workflows will use the most current process definition.

Wiring a process to the Alfresco UI

So far we've talked about the definition of workflow, specifics around the jBPM engine, and process deployment but we haven't addressed how to expose a process to Alfresco web client users.

For those familiar with extending the content model the steps for integrating a custom workflow with the Alfresco web client UI will be very familiar. The process is nearly identical. At a high level it consists of the following:

- Define a content model for your workflow in which workflow tasks map to content types.

- Update web-client-config-custom.xml to tell Alfresco how to expose the process metadata to the web client user interface.
- Externalize the strings.

We don't need to re-hash the details of custom content models here—I'll assume you already know how to extend Alfresco's content model. If these are new concepts to you I recommend you take a look at some of the resources cited in the “More Information” section at the end of this article.

Define a workflow-specific content model

The workflow-specific content model defines the data structure for the process. Workflow models use the same fundamental building blocks—types, properties, aspects, and associations—as “normal” Alfresco content model definitions. In fact, if you already have a custom model, you can define your workflow-specific model in the same content model XML file, although to reduce confusion, I recommend you keep your content types separate from your workflow types by using at least two different model files.

What is the purpose of the workflow-specific model? Think of it like any other content model. Custom content models are used to define the metadata we want to capture about a piece of content. The metadata (properties) are grouped into types and aspects. By virtue of defining these properties as part of the content model, Alfresco takes care of persisting the data to the underlying database.

Workflow models function in the same way. Suppose you have a process in which three different departments are involved in an approval process. Maybe you'd like the workflow initiator to be able to define which of those departments are required approvers and which are optional or “FYI” reviewers. A workflow model would define how that information is going to be stored.

As in other content models, you don't have to start from scratch. Alfresco ships out-of-the-box with some workflow-specific types already defined. There are two model definition files related to this. One is called called bpmModel.xml. It resides in your Alfresco web application root under WEB-INF/classes/alfresco/model. The other is called workflowModel.xml and it resides under WEB-INF/classes/alfresco/workflow.

The **bpmModel** file contains the lowest-level workflow classes such as the base definition for all tasks and the default start task. It also contains important aspects such as a set of “assignee” aspects that define associations between tasks and users or groups.

The **workflowModel** file contains the content model for the out-of-the-box process definitions. This model file offers a lot of potential for reuse in your custom processes. For example, if your process starts by allowing the submitter to specify a list of several people to receive a task, you could use the submitParallelReviewTask. If you want to base an approval on the percentage of individuals who approve a task, you can use the submitConcurrentReviewTask. Of course just like any model you are free to use these as-is, extend them, or not use them at all.

When users interact with the workflow via the web client, Alfresco will use the workflow content model and the web-client-config-custom.xml file to figure out what metadata to expose to the UI and how to present it just as it does when viewing content properties. Alfresco uses the name of the workflow task to figure out the appropriate workflow content type. So, **all tasks in which there are Alfresco web client user interactions must be given a name that corresponds to the name of a workflow content type.**

Consider this process definition snippet:

```
    <start-state name="start">
        <task name="scwf:submitGroupReviewTask" swimlane="initiator" />
        <transition name="transToAddAspect" to="addAspect" />
    </start-state>
```

The snippet shows a start-state of a process with a task assigned to the initiator. The task name must match a corresponding workflow content type. Looking in the custom workflow content model file we find the matching type definition:

```
    <type name="scwf:submitGroupReviewTask">
        <parent>bpm:startTask</parent>
        <mandatory-aspects>
            <aspect>bpm:groupAssignee</aspect>
        </mandatory-aspects>
    </type>
```

The type is a child of bpm:startTask and declares a mandatory aspect. In this particular case, the type will keep track of the group the initiator picks when submitting the workflow. A swimlane within the process definition can then read the selected group and pass that to the assignment class.

```
    <swimlane name="reviewer">
        <assignment class="org.alfresco.repo.workflow.jbpm.AlfrescoAssignment">
            <pooledactors>#{bpm_groupAssignee}</pooledactors>
        </assignment>
    </swimlane>
```

Note the use of the underscore to separate the namespace from the property name.

Update web-client-config-custom.xml

The next step is to tell Alfresco how to display the process metadata. This works exactly like custom content types. Continuing the example we just looked at, we need to expose the group picker to the web client so the initiator can find and select a group. The snippet below shows how.

```
    <config evaluator="node-type" condition="scwf:submitGroupReviewTask"
replace="true">
        <property-sheet>
            <separator name="sep2" display-label-id="users_and_roles" component-
generator="HeaderSeparatorGenerator" />
            <show-association name="bpm:groupAssignee"/>
        </property-sheet>
    </config>
```

Externalize the strings

The final step is to externalize the strings used to display things like the workflow title and description that show up in the “Start Advanced Workflow” dialog, and titles and descriptions for individual tasks. The format of the identifiers for these strings follow a specific format. Look at the snippet below as an example:

```
# scWorkflowModel related strings
scwf_workflowmodel.type.scwf_submitGroupReviewTask.title=Start SC Web Review
scwf_workflowmodel.type.scwf_submitGroupReviewTask.description=Submit SC Web
documents for review & approval to a group of people
scwf_workflowmodel.type.scwf_reviewTask.title=SC Web Review
scwf_workflowmodel.type.scwf_reviewTask.description=Review documents for
publication to the SC Web
scwf_workflowmodel.type.scwf_approvedTask.title=Content Approved
scwf_workflowmodel.type.scwf_approvedTask.description=Content Approved for
publication to the SC Web
scwf_workflowmodel.type.scwf_rejectedTask.title=Content Rejected
scwf_workflowmodel.type.scwf_rejectedTask.description=Content must be revised
before being published to the SC Web
```

```
# process definition related strings
scwf_submittoscweb.workflow.title=Submit to SC Web
scwf_submittoscweb.workflow.description=Review and approve SC Web content

scwf_submittoscweb.node.start.transition.transToAddAspect.title=Set SC Web
properties
scwf_submittoscweb.node.start.transition.transToAddAspect.description=System sets
SC Web properties

scwf_submittoscweb.node.addAspect.transition.transToReview.title=Review
scwf_submittoscweb.node.addAspect.transition.transToReview.description=Review
prior to publish
```

I tend to think of these properties as belonging to two groups. One group is the set of model-related properties. These properties externalize the strings in the workflow content model. The other is the set of process-related properties. These properties externalize the strings users see when they are working with the process (the workflow title, the workflow history, etc.).

Implementation summary

We've covered a lot of ground so far. The following summarizes the advanced workflow implementation steps:

1. Model the process using the jBPM Process Designer. Just get the process right—don't worry about node names, events, or actions just yet.
2. Add logic using Beanshell expressions, Alfresco JavaScript, or Java classes.
3. Define a workflow content model. If you use a new content model file, remember to update the custom model-context.xml file to point to the new content model definition XML file.
4. Update web-client-config-custom.xml to expose workflow tasks to the Alfresco web client.
5. Create/update a workflow-specific properties file to externalize the strings in both the workflow model and the process definition.
6. Deploy the process definition using either the deployment tab or a spring bean config file.

At this point you know enough about advanced workflows to be dangerous. Let's work through an example to put some of this new knowledge to work.

SomeCo Whitepaper submission example

This example continues the SomeCo example that we've looked at during the past several “Alfresco Developer” articles. SomeCo is going to use Advanced Workflows to route Whitepapers for approval before being flagged for publication on the web site. The next section describes the process.

Business process description

Anyone that can log in to Alfresco can submit a whitepaper for publication on the web site. The only information the submitter needs to specify is the email address of an external third-party reviewer, if applicable. More on that shortly.

The whitepaper needs to be reviewed by the Engineering team as well as the Marketing team. It doesn't matter who on the team does the review—SomeCo wants to notify each team and then let one representative from each team “own” the review task. Either team can reject the whitepaper. If rejected, the person who submitted the whitepaper can make revisions and resubmit. If both teams approve, the whitepaper moves on to the next step.

Some whitepapers need to be reviewed by an external third-party. The third-party won't actually log in to Alfresco—they'll get an email and click a link to approve or reject the whitepaper. If the third-party doesn't do anything in a certain amount of time, the whitepaper should be automatically approved.

High-level steps

Alright. We're going to implement this process in four major steps. Here's a look at the major steps and the respective sub-steps:

1. Implement the basic flow and workflow user interface
    1. Lay out the process using the JBoss jBPM Graphical Process Designer.
    2. Configure swimlanes and add task-nodes with appropriate assignments.
    3. Add decision logic.
    4. Implement the workflow content model, the workflow client configuration, and the workflow properties.

    5. Deploy and test.

2. Implement web scripts and actions for external third-party integration and other business logic
    1. Execute an action to add the "sc:webable" aspect to the whitepaper and set the properties appropriately.
    2. Write a web script to handle approval/rejection via HTTP. The logic needs to grab the workflow ID and then signal the node with the appropriate transition.
    3. Write a jBPM action class that sends a notification to the third-party email address.
    4. Deploy and test.
3. Add a timer to the third-party task
    1. Add a timer to the Third Party Review task so that if the third party doesn't respond in a timely fashion the task will automatically approve.
    2. Deploy and test.
4. Configure the workflows for deployment upon startup via Spring.

Now that you know where we're headed at a high-level, let's get into the details.

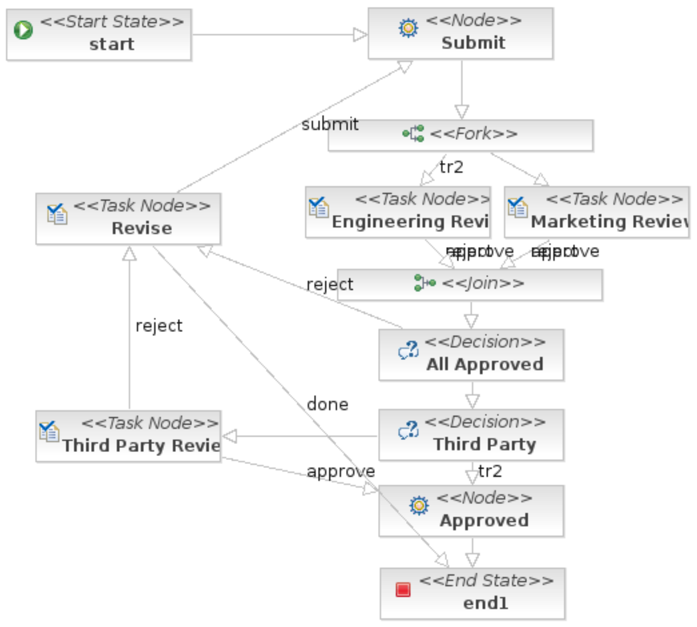
Step 1: Implement the basic flow and workflow user interface

Layout the process

I'm going to follow the "extension" convention and place my workflow in a file called processdefinition.xml in a folder called "alfresco/extension/workflows/publish-whitepaper". It definitely helps the GPD if each workflow resides in its own folder.

Speaking of which, I'm going to assume you are also using the GPD. If you don't want to, that's fine. Just create the folder structure I mentioned above and create the empty XML file. If you aren't using the GPD, skip the rest of the "Layout the process" step and move on to "Swimlanes".

Using the GPD Diagram tab and the description of the business process, lay out the process as follows:



Here are a couple of things to note about the diagram. First, the reason the transition labels between the Engineering Review and Marketing Review task nodes and the Join are hard to read is because there are two transitions for each and the GPD overlays them. So each Review task node has two leaving transitions—one for approve and one for reject.

Next, notice I've named most, but not all, of my transitions. Any transition leaving a task node should be labeled because a human is going to see that (or its externalized equivalent) in the UI. Everywhere else it is convenient if the transitions have names but it isn't required.

Finally, don't spend too much time making the diagram pretty. GPD has an annoying habit of scrambling the diagram from time-to-time.

Swimlanes

Now edit the source of processdefinition.xml. First, let's fix the process definition element. By default, the xmlns attribute is blank and the name will contain whatever you specified when you created the process (if you used "New JBoss jBPM Process Definition" in Eclipse). These need specific values to work with Alfresco. Update them as follows:

```
<process-definition xmlns="urn:jbpm.org:jpdl-3.1" name="scwf:publishwhitepaper">
```

The name has a namespace on it because we're going to externalize it and we don't want it to clash with other workflow names. I'm using the namespace of the workflow content model we will create a little later.

Now drop in the swimlanes. We need the built-in “initiator” swimlane we talked about earlier plus one for Marketing and one for Engineering. The out-of-the-box processes show you how to use a picker to let the initiator specify a user or a group. We're going to hardcode Alfresco groups because we know which groups need to be assigned to the swimlanes—there's no reason to make the initiator pick in this case.

Also, we're going to use pooled actors because the business process is that the entire team needs to get a task with one person taking ownership of and completing that task.

Add the swimlanes to the start of the process definition.

```
    <swimlane name="initiator" />

    <swimlane name="marketing">
       <assignment class="org.alfresco.repo.workflow.jbpm.AlfrescoAssignment">
           <pooledactors>#{people.getGroup('GROUP_marketing')}</pooledactors>
       </assignment>
    </swimlane>

    <swimlane name="engineering">
       <assignment class="org.alfresco.repo.workflow.jbpm.AlfrescoAssignment">
           <pooledactors>#{people.getGroup('GROUP_engineering')}</pooledactors>
       </assignment>
    </swimlane>
```

The swimlanes use the AlfrescoAssignment class to assign the actor. In this case we're using an expression that leverages the Alfresco people object because we need the actor to be a reference to the group object, not just a string containing the group name.

Add tasks to task-nodes

Our process has four task-nodes and a start-state node. Each of these need a task. It is the task that gets assigned to a swimlane. As mentioned earlier, task names will eventually match up to names of types in our workflow content model so they are prefixed with the namespace we're going to use for the workflow content model. The table below shows the task-nodes, tasks, and assignments.

Node	Task Name	Swimlane Assignment
start	scwf:submitReviewTask	initiator

Marketing Review	scwf:marketingReview	marketing
Engineering Review	scwf:engineeringReview	engineering
Third Party Review	scwf:thirdPartyReview	initiator
Revise	scwf:revise	initiator

The resulting jPDL for each of the nodes above looks like this:

```
<start-state name="start">
    <task name="scwf:submitReviewTask" swimlane="initiator" />
    <transition name="" to="Submit"></transition>
</start-state>

<task-node name="Marketing Review">
    <task name="scwf:marketingReview" swimlane="marketing" />
    <transition name="approve" to="join1"></transition>
    <transition name="reject" to="join1"></transition>
</task-node>

<task-node name="Engineering Review">
    <task name="scwf:engineeringReview" swimlane="engineering" />
    <transition name="approve" to="join1"></transition>
    <transition name="reject" to="join1"></transition>
</task-node>

<task-node name="Third Party Review">
    <task name="scwf:thirdPartyReview" swimlane="initiator" />
    <transition name="approve" to="Approved"></transition>
    <transition name="reject" to="Revise"></transition>
</task-node>

<task-node name="Revise">
    <task name="scwf:revise" swimlane="initiator"></task>
    <transition name="submit" to="Submit"></transition>
    <transition name="done" to="end1"></transition>
</task-node>
```

You may be wondering why the Third Party Review is a task-node. The Third Party Review is going to be assigned to someone external to the system who will approve or reject the task via email. So, technically, this should be a state node instead of a task-node. In fact, that's how I originally implemented it, but, unfortunately, Alfresco's implementation of jBPM timers only works for tasks in the current release[1]. The bright side is that if the initiator gets tired of waiting for the third-party before the timer runs out, they can approve the task on behalf of the third-party.

Decision logic

The process definition has two decisions. One decision figures out if all required approvals have been obtained. If so, the process continues. If not, the initiator gets a chance to make revisions. The other decision is used to determine if a third-party review is required based on whether or not the initiator provided an email address.

These are pretty easy decisions to make based on process variables. For the “All Approved” decision, we can increment a counter when the process follows an “approve” transition. If the counter is equal to 2, we know we received both approvals.

We need to be careful that we initialize the counter to 0 because it's possible that a whitepaper may go through several review cycles. The “Submit” node gives us a convenient place to do that:

```
<node name="Submit">
    <event type="node-enter">
      <script>
        <variable name="approveCount" access="read,write"/>
        <expression>
          approveCount = 0;
        </expression>
      </script>
    </event>
    <transition name="" to="fork1"></transition>
  </node>
```

Now we need to increment the counter when the approve transition is taken. Let's look at the Engineering Review as an example. We can modify the approve transition with a little script that increments the counter:

```
  <task-node name="Engineering Review">
    <task name="scwf:engineeringReview" swimlane="engineering" />
    <transition name="approve" to="join1">
      <script>
        <variable name="approveCount" access="read,write"/>
        <expression>
             approveCount = approveCount + 1;
```

1 http://issues.alfresco.com/browse/AR-1879

```
        </expression>
      </script>
    </transition>
    <transition name="reject" to="join1"></transition>
  </task-node>
```

If we add the same script to the approve transition for Marketing Review as well (not shown), our decision can conditionally transition based on the counter:

```
  <decision name="All Approved">
    <transition name="reject" to="Revise"></transition>
    <transition name="" to="Third Party">
      <condition>#{approveCount == 2}</condition>
    </transition>
  </decision>
```

Note that the first transition doesn't have a condition. It will be used as the default if the condition is not met.

For the “Third Party” decision we'll do something similar. In this case we're going to check a user-provided value. Because this property will be defined as part of the workflow content model, it includes the “scwf” namespace, just like the task names we set earlier.

```
  <decision name="Third Party">
    <transition name="tr2" to="Approved"></transition>
    <transition name="" to="Third Party Review">
      <condition>#{scwf_reviewerEmail!=""}</condition>
    </transition>
  </decision>
```

We didn't need it, but if our decision was more complex, we could have used a Java class to implement the decision logic. To do that, we would have used a “handler” tag that pointed to a Java class that implemented the DecisionHandler interface.

```
  <handler class="com.someco.bpm.SomeDecisionHandler" />
```

The handler class' decide() method would return the transition to take. For our needs, though, expressions were enough.

The fork, join, approved, and end nodes are all fine as they are for now. If you elected not to use the GPD, you can get them from the accompanying source if you are following along.

Workflow content model, UI, and props

Now let's integrate the process with Alfresco by defining the content model and updating the client configuration.

I recommend keeping your “normal” content models separate from your workflow models. So the first step is updating someco-model-context.xml with a pointer to the new workflow content model file:

```
  <bean id="extension.dictionaryBootstrap" parent="dictionaryModelBootstrap"
depends-on="dictionaryBootstrap">
     <property name="models">
        <list>
           <value>alfresco/extension/scModel.xml</value>
           <value>alfresco/extension/scWorkflowModel.xml</value>
        </list>
     </property>
  </bean>
```

While you're in that file, you might as well go ahead and add the config for the workflow properties file we're going to create shortly:

```
  <bean id="extension.workflowBootstrap" parent="workflowDeployer">
     <property name="labels">
        <list>
           <value>alfresco.extension.scWorkflow</value>
        </list>
     </property>
  </bean>
```

Next, create a new model file that matches the name in the context XML (scWorkflowModel.xml). I'm going to leave out the namespace declaration and imports and just show the types and aspects. Check the source for the full file.

```
<model name="scwf:workflowmodel"
  xmlns="http://www.alfresco.org/model/dictionary/1.0">

  <!-- namespaces and imports omitted from listing -->

  <types>
     <type name="scwf:submitReviewTask">
        <parent>bpm:startTask</parent>
        <mandatory-aspects>
           <aspect>scwf:thirdPartyReviewable</aspect>
        </mandatory-aspects>
     </type>
```

```
    <type name="scwf:marketingReview">
      <parent>bpm:workflowTask</parent>
      <overrides>
        <property name="bpm:packageItemActionGroup">
          <default>read_package_item_actions</default>
        </property>
      </overrides>
    </type>

    <type name="scwf:engineeringReview">
      <parent>bpm:workflowTask</parent>
      <overrides>
        <property name="bpm:packageItemActionGroup">
          <default>read_package_item_actions</default>
        </property>
      </overrides>
    </type>

    <type name="scwf:thirdPartyReview">
      <parent>bpm:workflowTask</parent>
      <overrides>
        <property name="bpm:packageItemActionGroup">
          <default>read_package_item_actions</default>
        </property>
      </overrides>
    </type>

    <type name="scwf:revise">
      <parent>bpm:workflowTask</parent>
      <overrides>
        <property name="bpm:packageItemActionGroup">
          <default>edit_package_item_actions</default>
        </property>
      </overrides>
    </type>

  </types>
```

There's one type for each task. The name of each type matches the name of the corresponding task.

You'll notice that each type inherits from a type defined in the BPM content model. If you look at the bpmModel.xml file you'll see that the bpm:startTask has helpful properties such as the workflow description, due date, and priority.

The bpm:workflowTask has an association called bpm:package. The bpm:package points to a bpm:workflowPackage which is the aspect applied to a container (like a folder) that holds the documents being routed through a workflow. When you write code that needs to access the content being routed in a workflow you can get to it through the bpm:package association.

The “bpm:packageItemActionGroup” defines what actions are available for working with the content in the workflow at that particular step in the process. In our case, we want the initiator to be able to change the contents of the workflow when the workflow is started and when making revisions, but we don't want the reviewers to be able to add or remove anything to or from the workflow.

The start task has a mandatory aspect called scwf:thirdPartyReviewable. We'll define that aspect to contain a property we can use to store the third-party reviewer's email address:

```
    <aspects>
      <aspect name="scwf:thirdPartyReviewable">
        <title>Someco Third Party Reviewable</title>
        <properties>
          <property name="scwf:reviewerEmail">
            <type>d:text</type>
            <mandatory>true</mandatory>
            <multiple>false</multiple>
          </property>
        </properties>
      </aspect>
    </aspects>
```

Now we need to tell Alfresco how to render the properties in the workflow model. Just like extending regular content models, we do that through the web-client-config-custom.xml file. I like to put all of my workflow-related config entries at the bottom of this file to keep them separate.

```
  <!--  workflow property sheets -->

  <config evaluator="node-type" condition="scwf:submitReviewTask" replace="true">
    <property-sheet>
      <separator name="sep1" display-label-id="general" component-
generator="HeaderSeparatorGenerator" />
      <show-property name="bpm:workflowDescription" component-
generator="TextAreaGenerator" />
    </property-sheet>
  </config>

  <config evaluator="node-type" condition="scwf:marketingReview" replace="true">
```

```
    <property-sheet>
      <separator name="sep1" display-label-id="general" component-
generator="HeaderSeparatorGenerator" />
      <show-property name="bpm:description" component-
generator="TextAreaGenerator" read-only="true"/>
      <show-property name="bpm:comment" component-generator="TextAreaGenerator"
/>
    </property-sheet>
  </config>

  <config evaluator="node-type" condition="scwf:engineeringReview" replace="true">
    <property-sheet>
      <separator name="sep1" display-label-id="general" component-
generator="HeaderSeparatorGenerator" />
      <show-property name="bpm:description" component-
generator="TextAreaGenerator" read-only="true"/>
      <show-property name="bpm:comment" component-generator="TextAreaGenerator"
/>
    </property-sheet>
  </config>

  <config evaluator="node-type" condition="scwf:revise" replace="true">
    <property-sheet>
      <separator name="sep1" display-label-id="general" component-
generator="HeaderSeparatorGenerator" />
      <show-property name="bpm:description" component-
generator="TextAreaGenerator" read-only="false"/>
      <show-property name="bpm:comment" component-generator="TextAreaGenerator"
/>
    </property-sheet>
  </config>

  <!--  add third-party reviewable related aspect properties to property sheet -->
  <config evaluator="aspect-name" condition="scwf:thirdPartyReviewable">
    <property-sheet>
      <show-property name="scwf:reviewerEmail" display-label-id="email" />
    </property-sheet>
  </config>
```

You can see that although the BPM model defines several properties, I'm only choosing to expose the workflow description and comment properties for this particular process. The description is editable only when submitting the workflow or doing a revision and is read-only everywhere else.

The last step is to externalize the strings in our model and our process. Remember that earlier we registered a resource bundle called scWorkflow. Create a file called scWorkflow.properties and add the strings. The first set of strings to add have to do with the model:

```
# scWorkflowModel related strings
scwf_workflowmodel.type.scwf_submitReviewTask.title=Start SC Web Review
scwf_workflowmodel.type.scwf_submitReviewTask.description=Submit SC Web documents
for review & approval to a group of people
scwf_workflowmodel.type.scwf_marketingReview.title=Marketing Review
scwf_workflowmodel.type.scwf_marketingReview.description=Review documents for
impact on SomeCo marketing message
scwf_workflowmodel.type.scwf_engineeringReview.title=Engineering Review
scwf_workflowmodel.type.scwf_engineeringReview.description=Review documents for
technical accuracy and best practices
scwf_workflowmodel.type.scwf_thirdPartyReview.title=Third-Party Review
scwf_workflowmodel.type.scwf_thirdPartyReview.description=Third-party reviews
documents as necessary
scwf_workflowmodel.property.scwf_reviewerEmail.title=Reviewer email
scwf_workflowmodel.property.scwf_reviewerEmail.description=Third-party reviewer
email address
```

The first part of each key matches the name of the workflow content model. There's a title and a description entry for each type and property in our workflow model.

Next, to the same file, we add the process-related strings:

```
# processdefinition related strings
scwf_publishwhitepaper.workflow.title=Publish Whitepaper to SC Web
scwf_publishwhitepaper.workflow.description=Review and approve SC Whitepaper
content

scwf_publishwhitepaper.node.Marketing\ Review.transition.approve.title=Approve
scwf_publishwhitepaper.node.Marketing\
Review.transition.approve.description=Approve this change

scwf_publishwhitepaper.node.Marketing\ Review.transition.reject.title=Reject
scwf_publishwhitepaper.node.Marketing\ Review.transition.reject.description=Reject
this change
```

This file is fairly redundant so it's truncated here. Note that the first part of the property key matches the name we gave our process definition. The values for the workflow.title and workflow.description keys will be what the user sees when she clicks “Start Advanced Workflow” in the Alfresco web client. The remaining titles and descriptions are the strings shown when someone manages a task.

Deploy and test

While we're still tweaking the workflow, it's handy to use the GPD deployment tool because we can change the process definition without a restart. The exception is when the model changes. So for this first test, we have to restart Alfresco to pick up the new model, then we can deploy using GPD.

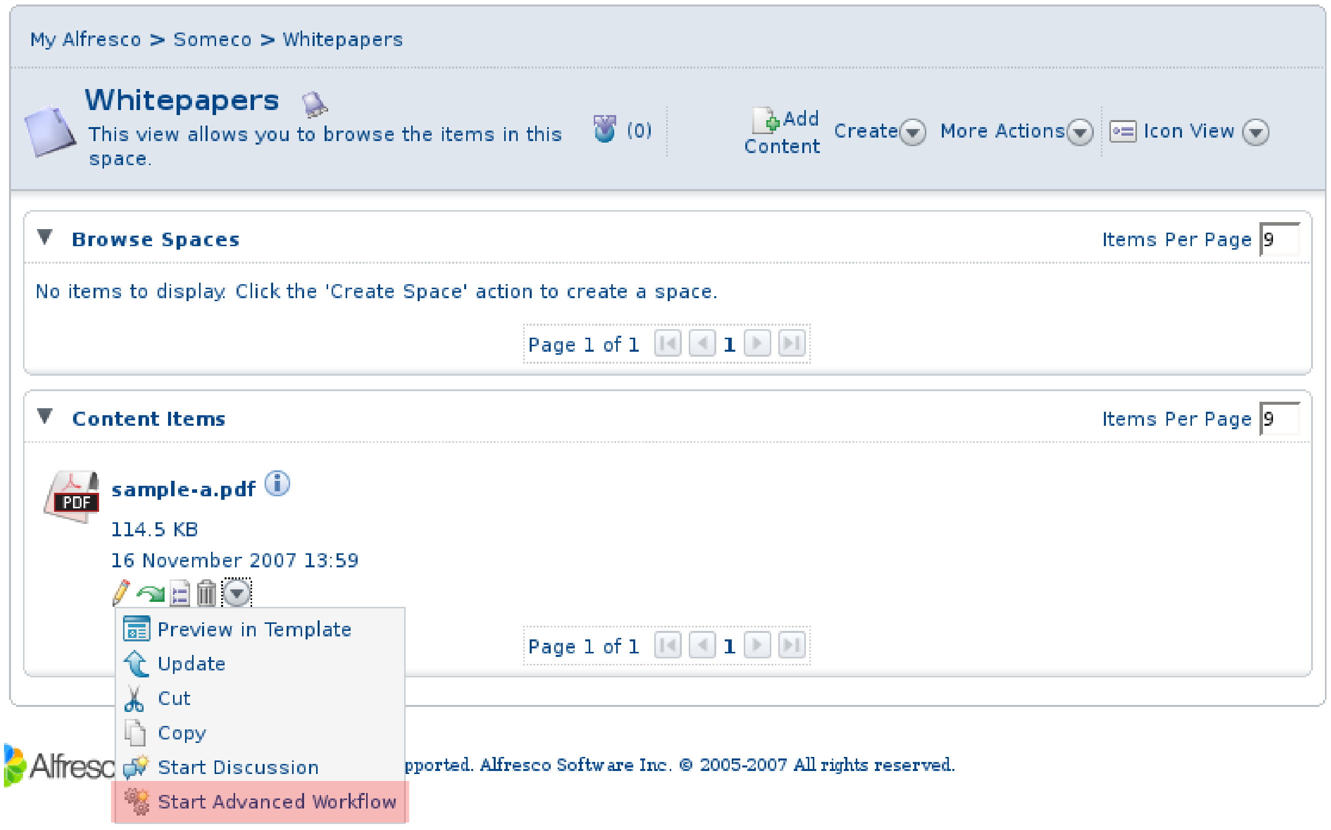
If you haven't done so already, create a group called “engineering” and one called “marketing”. Create a couple of test users for each group.

Deploy the customizations and restart Alfresco (See “How to deploy” near the end of this article if you need help). If everything starts cleanly, use the Deployment tab on the GPD to deploy the process definition. You don't need to do anything other than make sure the Deployment Server Settings are correct (Server Deployer should be “/alfresco/jbpm/deployprocess”).

Tip: Change the default jBPM Server Deployer URL in the Eclipse preferences to avoid having to re-set the Server Deployer URL value every time you want to deploy the process.

Create a piece of test content. Make sure you've configured permissions such that the marketing and engineering groups have Editor access or higher[1]. Then, click Start Advanced Workflow.

1 We haven't set it up yet, but the last step in the process is going to be to run an action that adds an aspect and sets a property. There appears to be a bug (Jira TBD) in which Alfresco JavaScript executes using the credentials of the last assigned task node. If the workflow path does not go to the Third-Party Review and instead goes directly to Approved from the join, the credentials of either the Engineering or Marketing user will be used to add the aspect. Thus, those groups need Editor access or higher.


My Alfresco > Someco > Whitepapers

Whitepapers

This view allows you to browse the items in this space.

Browse Spaces — Items Per Page 9

No items to display. Click the 'Create Space' action to create a space.

Page 1 of 1

Content Items — Items Per Page 9

sample-a.pdf

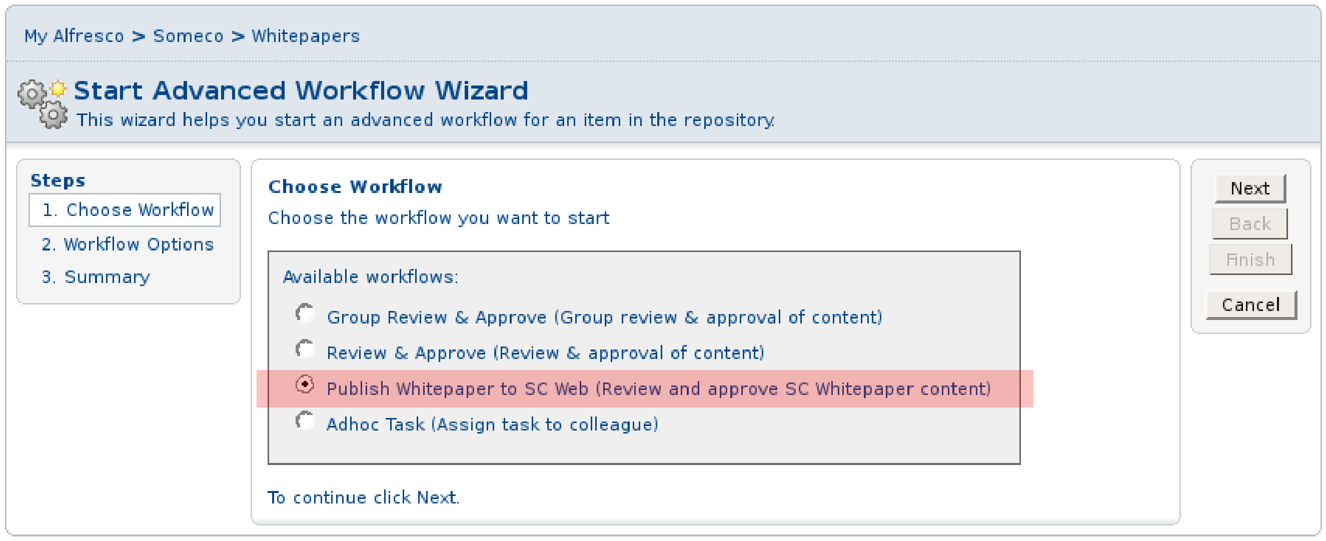
114.5 KB

16 November 2007 13:59

Page 1 of 1

Alfresc... pported. Alfresco Software Inc. © 2005-2007 All rights reserved.

You should see the newly-deployed workflow in the list of available workflows.


My Alfresco > Someco > Whitepapers

Start Advanced Workflow Wizard

This wizard helps you start an advanced workflow for an item in the repository.

Choose Workflow

Choose the workflow you want to start

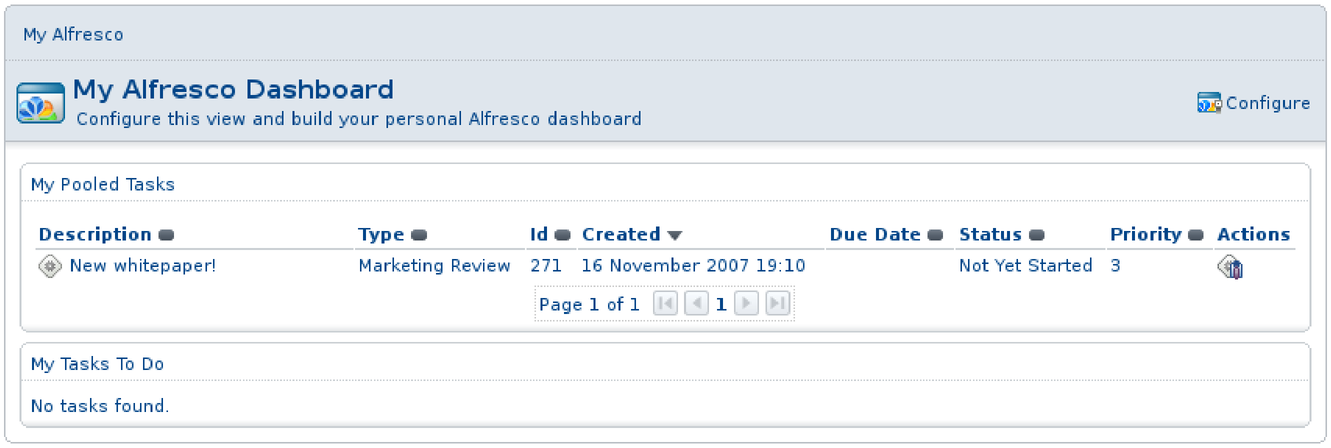
Available workflows:

- ○ Group Review & Approve (Group review & approval of content)
- ○ Review & Approve (Review & approval of content)
- [✓ Publish Whitepaper to SC Web (Review and approve SC Whitepaper content)]
- ○ Adhoc Task (Assign task to colleague)

To continue click Next.

Provide a description. To test the third-party review path, specify an email address. Because we haven't set up the notification yet you don't have to worry about it trying to send an email. Click Finish to launch the workflow.

Log in as one of your test users. Because we're using pooled tasks, you'll need to add a dashlet to your “My Alfresco” dashboard. After configuring the dashboard to include the “My Pooled Tasks” dashlet, you should then see the task.


My Alfresco
My Alfresco Dashboard
Configure this view and build your personal Alfresco dashboard
Configure
My Pooled Tasks

Description	Type	Id	Created	Due Date	Status	Priority	Actions
New whitepaper!	Marketing Review	271	16 November 2007 19:10		Not Yet Started	3	

No tasks found.

Either take ownership of the task so that you can see how the task moves to your To Do list and gets removed out of the other users' inboxes (if the users are members of the same group) or simply Approve the task.

In the real world, you'd run several tests—you have to make sure you test every possible path through the workflow.

Using the workflow console

If something goes wrong or you just want to get up close and personal with the execution of the process, you'll need to use the workflow console. Unfortunately, there's not a link in the UI for it just yet, but the URL is http://localhost:8080/alfresco/faces/jsp/admin/workflow-console.jsp. The table below shows some common commands and what they do.

Command	What it does
`show workflows all`	Shows all running workflows.
`use workflow <workflow id>`	Makes all subsequent commands happen in the

where `<workflow id>` is something like jbpm$71	context of the specified workflow.
`show transitions`	Shows all leaving transitions.
`signal <path id> <transition>` where `<path id>` is something like jbpm$71-@ and transition is the name of the leaving transition you want to take. Leave off the transition to take the default.	Signals the token. Good when your workflow is stuck on a node or when you want to take a transition without fooling with the task management UI.
`desc path <path id>` where `<path id>` is something like jbpm$71-@	Dumps the current context. Great for debugging process variables.
`end workflow <workflow id>`	Cancels the specified workflow.
`show definitions all`	Shows the current deployed workflow
`undeploy definition <workflow id>` or `undeploy definition name <workflow name>`	Undeploys the specified workflow and stops any workflows running with that definition. The <workflow id> variant undeploys a specific version of a workflow.

These are a subset of the commands available. Type “help” and click Submit to see the full list of commands.

Other debug aids include using “logger.log” statements in Alfresco JavaScript actions (with log4j.logger.org.alfresco.repo.jscript set to DEBUG) and using System.out statements in non-Alfresco script blocks. When you start writing Java action classes and decision handlers you may find it handy to hook up the Eclipse remote debugger as well.

If all goes well, your workflow should complete successfully. After each test run you should see no active workflows when you do a “show workflows all” on the workflow console.

Step 2: Implement web scripts and actions

Now that the base process is running and it's hooked in to the Alfresco UI, it's time to pimp it out with some business logic.

Call the set-web-action in the Approved node

Let's work on the Approved node first because it's easy. When a whitepaper is approved we want to add the “sc:webable” aspect to it and set the “isActive” and “published” properties. You may recall from the web script article that these get used by the front-end to determine if the whitepaper should be shown on the web site. I happen to have a rule action lying around that adds the aspect and sets the properties[1], so all we have to do is tell our process to execute it via Alfresco JavaScript. Update the Approved node as follows:

```
<node name="Approved">
  <transition name="" to="end1">
    <action class="org.alfresco.repo.workflow.jbpm.AlfrescoJavaScript">
      <script>
        <variable name="bpm_package" access="read" />
        <expression>
          var setWebFlagAction = actions.create("set-web-flag");
          setWebFlagAction.parameters["active"] = true;
          for (var i = 0; i &lt; bpm_package.children.length; i++) {
            setWebFlagAction.execute(bpm_package.children[i]);
          }
        </expression>
      </script>
    </action>
  </transition>
</node>
```

This is a straightforward piece of Alfresco JavaScript that executes the custom action called “set-web-flag” for every piece of content in the workflow package. The action adds the aspect and sets the properties appropriately.

Notice I've placed the action on the transition instead of as a child of the Approved node itself. It's syntactically correct to move the action outside of the transition (it then behaves like a node-enter event), but I found that the action was getting executed and then not signaling the node. With the action on the transition, the node immediately takes the default transition and performs the action as part of

1 I've covered actions in another paper. Unfortunately, it was before the “SomeCo” example was born so it doesn't flow well with the rest of the Alfresco Developer articles. I'll fix that someday. If you want to see how the action is defined, take a look in the source at someco-actions-context.xml, somecoactions.properties, and com.someco.action.executer.SetWebFlag.java

that step which is exactly what I needed it to do.

Potential Gotcha! Notice that I'm declaring the bpm_package variable prior to accessing it in my expression. This isn't always required, but I haven't been able to reliably predict when it is and when it isn't. This exact same code worked without a variable declaration in prior versions of Alfresco, but as of 2.1, the variable tag declaring bpm_package seems to be required. With 2.1, Alfresco upgraded to version 3.2 of jBPM so that may be the cause of the change in behavior. Regardless, it's probably always best to make the declaration.

That's all we need to do for the Approved node. Now let's work on the Third Party Review.

Implement the external third-party review

Someday there will be an out-of-the-box mechanism for exposing business processes to external parties. Until then, we can roll our own using the out-of-the-box mail action and web scripts. There are two pieces required to make this work. The first part is that when the token arrives in the Third Party Review node, we want to send an email to the third party with “approve” and “reject” links. The recipient will open their email and click on either the approve link or the reject link. The second part is that those links will invoke an Alfresco web script that signals the appropriate node.

This may seem backwards, but let's build the web script that handles the links first, then build the assignment class that sends the email notification.

We've covered the Web Script framework in a previous article (See “More Information”) so let's just look at the Java class that acts as the web script controller. Here's the executeImpl() method:

```
  protected Map<String, Object> executeImpl(WebScriptRequest req, WebScriptStatus
status) {

    String id = req.getParameter("id");
    String action = req.getParameter("action");

    if (id == null || action == null) {
       status.jsSet_code(400);
       status.jsSet_message("Required data has not been provided");
       status.jsSet_redirect(true);
    }

    workflowService.signal(id, action);

    Map<String, Object> model = new HashMap<String, Object>();
    model.put("response", "Success");

    return model;
  }
```

The class grabs the workflow ID and the action to take (which is really just a transition name) and then uses the workflow service to signal the node. So, for example, if someone were to post this URL:

http://localhost:8080/alfresco/service/someco/bpm/review?id=jbpm$89-@&action=approve

the Java class would signal the node identified by jbpm$89-@ with the "approve" transition[1].

The second piece to the Third Party Review is sending the email to the third-party. There is an out-of-the-box mail action, and you've already seen how to call an action from a workflow using Alfresco JavaScript, but we've got a few things to take care of other than simply sending an email, so let's write a custom action class.

Create a new class called com.someco.bpm.ExternalReviewNotification that extends JBPMSpringActionHandler. We need to build an email that has two links—one for approve and one for reject—that represent the two possible transitions out of the Third Party Review node.

The URL for the BPM web script we just implemented contains the path ID that needs to get signaled. The path ID is an Alfresco concept that I equate to jBPM's "token". It is a string created by concatenating the workflow engine identifier ("jbpm") with the jBPM process instance ID which we can get from the jBPM API.

With that information, we can implement the execute() method of the ExternalReviewNotification action class as follows:

```
public void execute(ExecutionContext executionContext) throws Exception {

    String recipient = (String) executionContext.getVariable(
                            ExternalReviewNotification.RECIP_PROCESS_VARIABLE);

    StringBuffer sb = new StringBuffer();
    sb.append("You have been assigned to a task named ");
    sb.append(executionContext.getToken().getNode().getName());
    sb.append(". Take the appropriate action by clicking one of the links
below:\r\n\r\n");
```

1 The web script configuration that maps the URL to the controller and view resides in alfresco/extension/templates/webscripts/com/someco/bpm. The controller class is com.someco.scripts.GetReview.

```
    List transitionList = executionContext.getNode().getLeavingTransitions();

    for (Iterator it = transitionList.iterator(); it.hasNext(); ) {
      Transition transition = (Transition)it.next();
      sb.append(transition.getName());
      sb.append("\r\n");
      sb.append("http://localhost:8080/alfresco/service/someco/bpm/review?id=jbpm
$");
      sb.append(executionContext.getProcessInstance().getId());
      sb.append("-@");
      sb.append("&action=");
      sb.append(transition.getName());
      sb.append("\r\n\r\n");
    }

    Action mailAction = this.actionService.createAction(MailActionExecuter.NAME);
    mailAction.setParameterValue(
          MailActionExecuter.PARAM_SUBJECT,
          ExternalReviewNotification.SUBJECT);
    mailAction.setParameterValue(MailActionExecuter.PARAM_TO, recipient);
    mailAction.setParameterValue(
          MailActionExecuter.PARAM_FROM,
          ExternalReviewNotification.FROM_ADDRESS);
    mailAction.setParameterValue(MailActionExecuter.PARAM_TEXT, sb.toString());

    this.actionService.executeAction(mailAction, null);

    return;
  }
```

The first thing the method does is grab the recipient from a process variable. I used a different process variable than the one in our “scwf” namespace because I didn't want to couple this action class with a specific workflow model.

Next, I start building the message body with a string buffer. The `executionContext.getToken().getNode().getName()` call grabs the node name (“Third Party Review”).

Rather than hardcode “approve” and “reject”, I iterate over the leaving transitions to spit out their names followed by the appropriate web script URL. That way, if the process ever changes, there's a chance the action class won't need to be touched.

The last major block of code uses the action service to execute the Alfresco mail action. Sure, you could use the Java mail API to do it yourself, but why not leverage the mail action? That way you can leverage the same SMTP configuration settings as Alfresco.

The last thing we have to do is call the new action class from the process. We'll use the “node-enter” event to trigger the action. Update the Third Party Review node as follows:

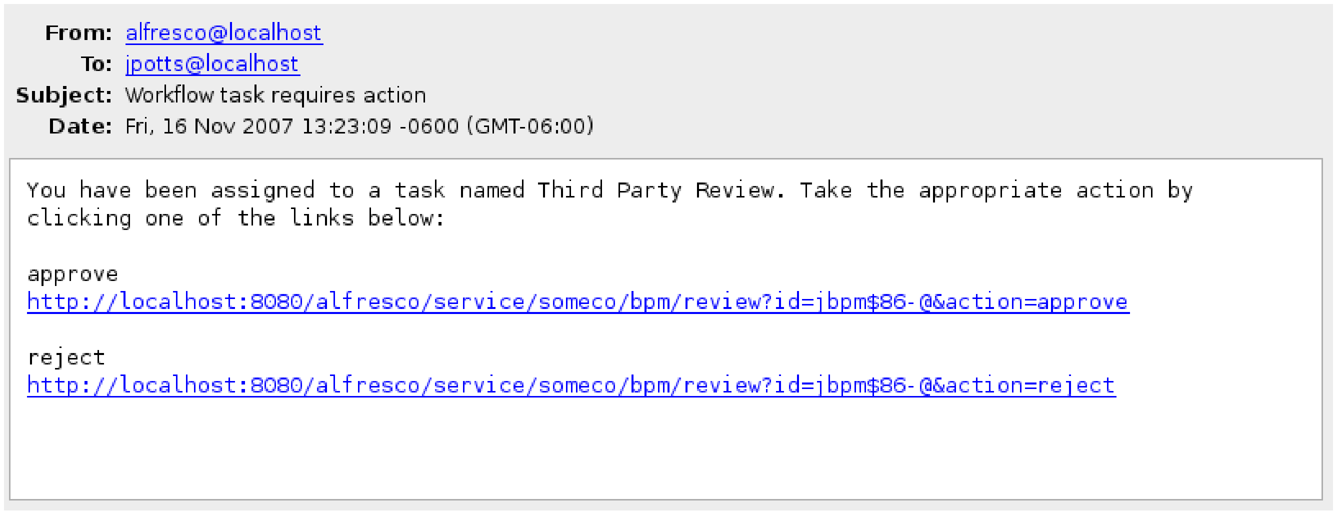
```
<task-node name="Third Party Review">
  <event type="node-enter">
    <script>
      <variable name="notificationRecipient" access="read,write" />
      <variable name="scwf_reviewerEmail" access="read" />
      <expression>
        notificationRecipient = scwf_reviewerEmail;
      </expression>
    </script>
    <action class="com.someco.bpm.ExternalReviewNotification"/>
  </event>
  ...
```

In the first part of the event, I'm reading the reviewer's email address from the property populated by the workflow initiator into a variable named as the action class expects. Then, I call the action.

Deploy and test

We've added a new rule action, a web script, and a jBPM action class. It's time to deploy and test. If you want to try out the notification piece, you'll need access to an SMTP server. For developing and testing locally, Apache James works great. If the SMTP server you use is running somewhere other than localhost, you'll have to tell Alfresco about it via the mail.host setting in custom-repository.properties.

If all goes well, you should get an email that looks like this:



```
From: alfresco@localhost
To: jpotts@localhost
Subject: Workflow task requires action
Date: Fri, 16 Nov 2007 13:23:09 -0600 (GMT-06:00)

You have been assigned to a task named Third Party Review. Take the appropriate action by
clicking one of the links below:

approve
http://localhost:8080/alfresco/service/someco/bpm/review?id=jbpm$86-@&action=approve

reject
http://localhost:8080/alfresco/service/someco/bpm/review?id=jbpm$86-@&action=reject
```

If you click on a link, it should signal the task-node. You should see the workflow continue on the appropriate path.

Warning: The Third Party example is *not* production-ready

I included the third-party example to show one type of wait-state/asynchronous behavior in a process. It's got a long way to go before it can be used in production. A short list list of obvious issues includes:

- The email recipient doesn't get a copy of the documents being reviewed. One way to address this would be to have the notification action send a zip of the documents in the workflow package. Another way would be to write additional web scripts or send them a download link. I simply didn't have time to implement this and figured it was a bit off-topic anyway.
- The person clicking the email link doesn't need to authenticate because it uses the session already established by the web browser when you logged in as one of your test users to complete the review step. (A horrible hack, I know. See previous "Time" excuse.) In real life, the Java controller would have to create its own session as admin or some other user in order to signal the node. There's an example of how to do this in the "Intro to Web Scripts" article.
- It'd be really easy for an unauthorized person to signal any node in the system because my controller class doesn't do any validation whatsoever and the path ID's are sequential. In real life, you'd want to check that (1) the person making the request is the person assigned to the task, (2) that the task is still active, and (3) possibly use an additional security mechanism like a shared secret of some kind.

- The email body should probably come from a Freemarker template. That way you could reuse the notification class in any number of processes and it simplifies email body maintenance.

So, long story short, feel free to use this idea, but realize that I've cut corners for brevity's sake.

Step 3: Add a timer to the third-party task

What if you sent an email to the third party and they never took action? One way to handle that problem is with a timer. Although jBPM has been a part of Alfresco since release 1.4, timers didn't start working until 2.1—they had to be there for the pre-publish functionality in WCM.

Timers get set on a node either using the "create-timer" tag or via the "timer" tag. When a timer expires, the process can take a transition, execute a script, or call an action. In our case, we just want to take the "approve" transition when the timer expires and write a message to the log. Update the scwf:thirdPartyReview task in the Third Party Review node as follows:

```
    <task name="scwf:thirdPartyReview" swimlane="initiator" >
      <timer name="thirdPartyTimer" duedate="10 minutes" transition="approve">
        <action class="org.alfresco.repo.workflow.jbpm.AlfrescoJavaScript">
          <script>
            logger.log("Third-party timer expired...approving");
          </script>
        </action>
      </timer>
    </task>
```

I've got the timer set for 10 minutes but you could obviously set it for as long or as short as you wish. You can also add a "business" modifier if you want to use business days instead of calendar days, for example. Setting an absolute date works as well. The due date could also be the result of an expression. Take a look at the out-of-the-box WCM workflow for an example.

Deploy and test

Adding a timer doesn't require a restart. Just redeploy the process using the GPD then start a new workflow. When the timer expires, the process should continue as if someone signaled the "approve" transition.

Step 4: Configure the workflows for deployment

The final step in the example is to configure the workflow for automatic deployment on Alfresco startup. Open the someco-model-context file. The bean that we used to register the workflow labels can be used to deploy workflows. Update it with the following:

```
  <property name="workflowDefinitions">
    <list>
      <props>
        <prop key="engineId">jbpm</prop>
        <prop key="location">alfresco/extension/workflows/publish-
whitepaper/processdefinition.xml</prop>
        <prop key="mimetype">text/xml</prop>
        <prop key="redeploy">true</prop>
      </props>
    </list>
  </property>
```

The details on the prop elements have already been discussed. The key is that the “location” matches the path to the process definition file.

Conclusion

You should now know the ins and outs of implementing advanced workflows using the jBPM engine embedded within Alfresco. You know when Alfresco's basic workflows will suffice and when advanced workflows are more appropriate. We talked about process definitions being a collection of nodes connected by transitions. Although you didn't see an example of this, you know that you could create a custom node type if you had to. And, you now know how to add business logic to workflows using expressions, Alfresco JavaScript and Java.

We also looked the overall process around implementing advanced workflows. Then we dived into the details by walking through an example that used many of the different node types and business logic options. We even spiced things up a bit by exposing the business process to a third-party via SMTP and HTTP with the help of the web script framework.

Hopefully, this has sparked some ideas about how you can leverage Alfresco and JBoss jBPM in your own projects and has given you some concrete examples you can leverage in your own projects going forward.

Deploying and testing

To deploy the sample code included with this article, all you have to do is:

1. Import the advanced-worklfow-article-project.zip file into Eclipse. (Or just expand it on your file system).
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 all other assets to the appropriate directories.

In case you are curious, my environment is:

- Ubuntu Gutsy Gibbon
- MySQL 5 (with version 5.1.5 of the JDBC driver)
- Java 1.5.0_13
- Tomcat 5.5.17
- Alfresco 2.1.1 Enterprise, WAR-only distribution
- JBoss jBPM Graphical Process Designer Plug-in 3.0.12
- Apache James 2.3.1 (for testing third-party notification via SMTP)

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

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:
  - "Intro to the Web Script Framework", October, 2007.

  - “Implementing custom behaviors”, September, 2007.
  - “Working with Custom Content Types”, June, 2007.
  - “Developing custom actions”, January, 2007.
- Alfresco wiki pages related to this topic:
  - Workflow Administration wiki page
  - Alfresco JavaScript API 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.
- JBoss jBPM
  - The JBoss jBPM Starter Kit has a standalone jBPM instance, examples, and documentation. If you're doing anything serious with advanced workflows, you should check it out.
  - The JBoss Graphical Process Designer is an Eclipse plug-in that can be used to create process definitions graphically and to deploy processes to the jBPM instance in Alfresco.
  - “The State of Workflow” is a technical article by Tom Baeyens, the founder and lead of the JBoss jBPM project, about BPM, workflow, and technical considerations that went into the creation of jBPM.

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.