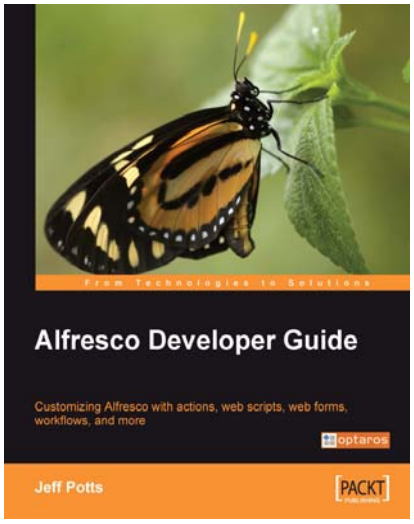
# Alfresco Developer Guide

**Jeff Potts**




## Chapter No. 3
## "Working with Content Models"

## In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Working with Content Models"

A synopsis of the book’s content

Information on where to buy this book

# About the Author

**Jeff Potts** leads the industry's largest group of certified Alfresco consultants as the Director of the Enterprise Content Management Practice at Optaros, a global consulting firm focused on assembling Next Generation Internet solutions featuring open source components. Jeff has over 10 years of ECM practice leadership and over 16 years of IT and technology implementation experience in IT departments and professional services organizations.

Jeff began working with and blogging about Alfresco in November of 2005. In 2006 and 2007, Jeff published a series of Alfresco tutorials and published them on his blog, ecmarchitect.com. That work, together with other Community activity in Alfresco's forum, Wiki site, and Jira earned him Alfresco's 2007 Community Contributor of the Year Award. The same year, Optaros earned Alfresco's Global Partner of the Year and Implementation of the Year awards.

Jeff's areas of business expertise include document management, content management, workflow, collaboration, portals, and search. Throughout his consulting career he has worked on a number of projects for Fortune 500 clients across the Media and Entertainment, Airline, Consumer Packaged Goods, and Retail sectors using technology such as Alfresco, Documentum, Java, XSLT, IBM WebSphere, and Lotus Domino.

Prior to Optaros, Jeff was a Vice President at Hitachi Consulting (formerly Navigator Systems, Inc.) where he founded and grew the ECM practice around legacy knowledge management, document management, Web Content Management (WCM), and collaboration solutions, in addition to custom development.

Jeff is a frequent speaker at Alfresco and Content Management industry events and has written articles for technical journals. This is his first book.

# Alfresco Developer Guide

Alfresco is the leading open source platform for Enterprise Content Management. The progress the Alfresco Engineering team has made since that first production release in June of 2005 has simply been amazing. The platform is well on its way to fulfilling its vision of becoming a viable alternative to those from legacy vendors who simply cannot keep up with the pace of innovation inherent in a solution assembled from open source components.

This book takes you through the process of customizing and extending the Alfresco platform. It uses a fictitious professional services company called "SomeCo" as an example. SomeCo has decided to roll out Alfresco across the enterprise. Your job is to take advantage of Alfresco's extension mechanism, workflow engine, and various APIs to meet the requirements from SomeCo's various departments.

Although many customizations can be made by editing XML and properties files, this book is focused on developers. That might mean writing Java code against the foundation API to implement an action or a behavior, maybe creating some server-side JavaScript to use as the controller of a RESTful web script, or perhaps implementing custom business logic in an advanced workflow. The point is that all but the most basic implementations of any ECM platform require code to be written. The goal of this book is to help you identify patterns, techniques, and specific steps that you can use to become productive on the platform more quickly.

By the end of this book, you will have stepped through every aspect of the Alfresco platform. You will have performed the same types of customizations and extensions found in typical Alfresco implementations. Most importantly, when someone comes to you and asks, "How would you do this in Alfresco?", you'll have at least one answer and maybe even some source code to go with it.

## What This Book Covers

*Chapter 1* is for people new to the Alfresco platform. It walks you through the capabilities of Alfresco and gives some examples of the types of solutions that can be built on the platform. You'll also learn what tools and skills are required to implement Alfresco-based solutions.

*Chapter 2* is about getting your development environment set up. Like preparing for a home improvement project, this is the trip to the hardware store to get the tools and supplies you'll need to get the job done. Throughout the book, you will be building and deploying changes. So just as in any software development project, it pays to get that process working up front. You'll also learn about the debugging tools that are available to you. The chapter includes a short and simple customization example to test out your setup.

*Chapter 3* starts where all Alfresco projects should begin: defining the content model. You'll learn how to define the content model as well as how to expose the model to the Alfresco web client. Once you've got it in place, you'll write some Java code that utilizes the Web Services API to test out the model. This will also be your first taste of the JavaScript API. The exercises set up the initial content model for SomeCo.

*Chapter 4* begins to show you the power of the repository by exposing you to some of the mechanisms or hooks that can be used to perform "hands off" operations on content. You'll learn about actions, behaviors, transformers, and metadata extractors. The exercises include implementing a rule action for SomeCo's Human Resources department to help manage HR policies, writing a custom behavior to calculate user ratings, and writing a custom metadata extractor to make Microsoft Project files indexable by the Lucene search engine.

*Chapter 5* takes you through web client customizations. First, it establishes whether or not you should be customizing the web client at all. Once that's out of the way, you learn how to add new menu items, how to create your own custom component renderers, and how to define new dialogs and wizards. Examples in this chapter include writing a new "Execute Script" UI Action to make it easier to run server-side JavaScript, creating a "Stoplight" component to graphically show project status, and creating a multi-step wizard SomeCo's HR department can use to set up job interviews.

*Chapter 6* focuses on the web script framework. Web scripts are an important part of the platform because they allow you to expose the repository through a RESTful API. They are also core to the Surf framework that is in the 3.0 release. The exercises in this chapter are about creating a set of URLs that can be called from the frontend web site to retrieve and persist user ratings of objects in the repository.

*Chapter 7* is about advanced workflows. You'll learn how the embedded JBoss jBPM workflow engine works and how to define your own workflows, including how to implement your own business logic. The chapter includes a comparison between the capabilities of Alfresco's simple workflow and advanced workflow so that you can decide which one is appropriate for your needs. By the end of the chapter, you will have built a workflow that SomeCo will use to review and approve Whitepapers for external publication. The process includes an asynchronous step, which leverages the web script knowledge you gained in the previous chapter.

*Chapter 8* takes you through the key developer-related aspects of Alfresco's Web Content Management functionality. The chapter is not an exhaustive WCM how-to. Rather, the chapter starts with a simple web form and then quickly moves to using the API to work with WCM assets. You'll also leverage advanced workflow and web script techniques you learned in previous chapters to work with WCM sites and assets. You'll create a "no approval" workflow that SomeCo will use for Job Postings and web scripts developers can use to deploy web sites to test servers and to commit changes to staging.

*Chapter 9* covers a variety of security-related topics. You'll learn how to define your own custom roles, and how to create users and groups with the API. Although not strictly developer-centric, you'll also learn how to configure Alfresco to authenticate and synchronize with an LDAP directory and how to implement Single Sign-On (SSO) between Alfresco and other web resources.

A set of Appendices is included at the end of the book. There you'll find reference information such as the JavaScript API, a set of diagrams showing the out of the box content model, and a list of the out of the box public spring beans. Also included is a section on packaging and deploying AMPs and an overview of the new Surf framework.

# 3
# Working with Content Models

From setting up the initial content model to programmatically creating, searching for, and deleting content, how you work with the content in a content management system is a foundational concept upon which the rest of the solution is built. In this chapter, you'll learn:

- What a repository is and how it is structured
- How to make the underlying content model relevant to your business problem by extending Alfresco's out of the box model with your own content types
- What practices are the best for creating your own content models
- How to configure the web client to expose your custom content model via the user interface
- How to interact with the repository via the Web Services and JavaScript APIs

## Defining SomeCo's Content Model

Recall from Chapter 1 that SomeCo is rolling out Alfresco across the organization. Each department has its own type of content to work with and different requirements for how it works with that content. SomeCo could just start uploading content into Alfresco. That would be better than nothing, but it relegates Alfresco to lowly file server status, doesn't take advantage of the full power of the platform, and makes for a really boring (and short) book. SomeCo would be better off formalizing the different types of content it works with by extending the Alfresco content model to make it SomeCo-specific.

# Step-by-Step: Starting the Custom Content Model with Custom Types

Let's start small and build the model up over successive examples. First, let's create a custom content type for Whitepapers, which are a type of marketing document. This is going to involve creating a content model file, which is expressed as XML, and then telling Alfresco where to find it using a Spring bean configuration file.

To start the content model, follow these steps:

1. Create an extension directory. In the Eclipse client-extensions project, under `|config|alfresco`, create a new folder called `extension` if you do not already have one. As discussed in Chapter 2, the extension directory keeps your customizations separate from Alfresco's code.
2. Create a custom model context file. A custom model context file is a Spring bean configuration file. Spring bean configuration files were also discussed in Chapter 2. Create the file in the extension directory and call it `someco-model-context.xml`. Add the following:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE beans PUBLIC '-//SPRING//DTD BEAN//EN'                    '
http://www.springframework.org/dtd/spring-beans.dtd'>
   <beans>
    <!-- Registration of new models -->
    <bean id="someco.dictionaryBootstrap"
    parent="dictionaryModelBootstrap" depends-on="dictionaryBoots
    trap">
        <property name="models">
            <list>
                <value>alfresco/extension/model/scModel.xml
                </value>
            </list>
        </property>
    </bean>
</beans>
```

3. Create a model file that implements the custom content model. The extension directory is going to fill up over time, so create a new directory under extension called `model`. Create a new XML file in the model directory called `scModel.xml` (this name matches the value specified in the `someco-model-context.xml` file).
4. Add the following XML that is used to describe the model, import other models that this model extends, and declare the model's namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Definition of new Model -->
<model name="sc:somecomodel" xmlns="http://www.alfresco.org/model/
dictionary/1.0">
 <!-- Optional meta-data about the model -->
 <description>Someco Model</description>
 <author>Optaros</author>
 <version>1.0</version>
 <!-- Imports are required to allow references to definitions in
   other models -->
 <imports>
  <!-- Import Alfresco Dictionary Definitions -->
  <import uri="http://www.alfresco.org/model/dictionary/1.0"
   prefix="d" />
  <!-- Import Alfresco Content Domain Model Definitions -->
  <import uri="http://www.alfresco.org/model/content/1.0"
    prefix="cm" />
 </imports>
 <!-- Introduction of new namespaces defined by this model -->
 <namespaces>
  <namespace uri="http://www.someco.com/model/content/1.0"
   prefix="sc" />
 </namespaces>
```

5. Next, add the types. A **Whitepaper** is a type of marketing document that, in turn, is only one of several types of content SomeCo deals with. It's a hierarchy. That hierarchy will be reflected in the model. Add this XML to the model file below the "namespaces" element:

```
<types>
 <!-- Enterprise-wide generic document type -->
 <type name="sc:doc">
  <title>Someco Document</title>
  <parent>cm:content</parent>
 </type>
 <type name="sc:marketingDoc">
  <title>Someco Marketing Document</title>
  <parent>sc:doc</parent>
 </type>
 <type name="sc:whitepaper">
 <title>Someco Whitepaper</title>
 <parent>sc:marketingDoc</parent>
 </type>
</types>
```

6. Be sure to close the model tag so the XML is valid.

```
</model>
```

7. The final step is to deploy the changes and then restart Tomcat so that Alfresco will load the custom model. Copy the `build.xml` file from the source code that accompanies this chapter into the root of your Eclipse project, replacing the `build.xml` file you used in the exercises for the previous chapter.
8. Run `ant deploy`.
9. Restart Tomcat.

Watch the log during the restart. You should see no errors related to loading the custom model. If there is a problem, the message usually looks something like "Could not import bootstrap model".

With this change in place, the repository is now capable of telling the difference between a generic piece of content and SomeCo-specific pieces of content such as marketing documents and Whitepapers.

## Types

Types are like types or classes in the object-oriented world. They can be used to model business objects, they have properties, and they can inherit from a parent type. "Content", "Person", and "Folder" are three important types defined out of the box. Custom types are limited only by your imagination and business requirements. Examples include things such as "Expense Report", "Medical Record", "Movie", "Song", and "Comment".

Did you notice the names of the types you created in the example? Names are made unique across the repository by using a namespace specific to the model. The namespace has an abbreviation. The model you created for SomeCo defines a custom model, which declares a namespace with the URI of `http://www.someco.com/model/content/1.0` and a prefix of "sc". Any type defined as a part of the model will have a name prefixed with "sc:". Using namespaces in this way helps to prevent name collisions when content models are shared across repositories.

# Step-by-Step: Adding Properties to Types

The Marketing department thinks in terms of marketing campaigns. In fact, they want to be able to search by a specific campaign and find all of the content tied to a particular campaign. Not to hurt the Marketing team's feelings, but the HR, Sales, and Operations teams couldn't care less about campaigns.

You are going to address this by adding a property to the `sc:marketingDoc` type to capture the marketing campaigns the document is related to. You could make this property a text field. But letting users enter a campaign as free-form text is a recipe for disaster if you care about getting valid search results later because of the potential for misspelling the campaign name. Plus, SomeCo wants to let each document be associated with multiple campaigns, which compounds the free-form text entry problem. So, for this particular property it makes sense to constrain its values to a list of valid campaigns.

To allow the Marketing team to "tag" a piece of content with one or more campaigns selected from a list of valid campaign names, update the model by following these steps:

1. Edit the `scModel.xml` file in **config | alfresco | extension**. Replace the `sc:marketingDoc` type definition with the following:

```
<type name="sc:marketingDoc">
   <title>Someco Marketing Document</title>
   <parent>sc:doc</parent>
   <properties>
     <property name="sc:campaign">
      <type>d:text</type>
      <multiple>true</multiple>
      <constraints>
        <constraint ref="sc:campaignList" />
      </constraints>
     </property>
   </properties>
</type>
```

2. Now, define the campaign list constraint. Between the "namespaces" and "types" elements, add a new "constraints" element as follows:

```
<constraints>
  <constraint name="sc:campaignList" type="LIST">
            <parameter name="allowedValues">
                <list>
                    <value>Application Syndication</value>
        <value>Private Event Retailing</value>
                    <value>Social Shopping</value>
                </list>
            </parameter>
    </constraint>
</constraints>
```

3. Save the `model` file.
4. Run `ant deploy`.
5. Restart Tomcat.

Again, Tomcat should start cleanly with no complaints about the content model.

## Properties and Property Types

Properties are pieces of metadata associated with a particular type. In the previous example, the property was the marketing campaign. The properties of a SomeCo Expense Report might include things such as "Employee Name", "Date submitted", "Project", "Client", "Expense Report Number", "Total amount", and "Currency". The Expense Report might also include a "content" property to hold the actual expense report file (maybe it is a PDF or an Excel spreadsheet, for example).

You may be wondering about the `sc:whitepaper` type. Does anything special need to happen to make sure whitepapers can be tied to campaigns as well? Nope! In Alfresco, content types inherit the properties of their parent. The `sc:whitepaper` type will automatically have an `sc:campaign` property. In fact, it will have all sorts of properties inherited from its ancestor types. The file name, content property, and creation date are three important examples.

Property types (or data types) describe the fundamental types of data the repository will use to store properties. The data type of the `sc:campaign` property is `d:text`. Other examples include things such as dates, floats, Booleans, and content that is the property type of the property used to store content in a node. Because these data types literally are fundamental, they are pretty much the same for everyone. So they are defined for you out of the box. Even though these data types are defined out of the box, if you wanted to change the Alfresco data type "text" so that it maps to your own custom class rather than `java.lang.String`, you could.

## Constraints

Constraints can optionally be used to restrict the values that Alfresco will store in a property. In the following example, the `sc:campaign` property used a LIST constraint. There are three other types of constraints available: REGEX, MINMAX, and LENGTH. REGEX is used to make sure that a property value matches a regular expression pattern. MINMAX provides a numeric range for a property value. LENGTH sets a restriction on the length of a string.

Constraints can be defined once and reused across a model. For example, out of the box, Alfresco makes available a constraint named `cm:filename` that defines a regular expression constraint for file names. If a property in a custom type needs to restrict values to those matching the filename pattern, the custom model doesn't have to define the constraint again. It simply refers to the `cm:filename` constraint.

# Step-by-Step: Relating Types with Associations

SomeCo has a generic need to be able to identify documents that relate to each other for any reason. A Whitepaper might be tied to a solution offering data sheet, for example. Or maybe a project proposal the Legal department has should be related to the project plan the Operations team is managing. These relationships are called **associations**.

Let's update the model file to include a related-documents association in the `sc:doc` type so that any SomeCo document can be related to any other.

To add assocations to the model, follow these steps:

1. Edit the `scModel.xml` file.
2. Add the following associations element to the `sc:doc` type. Notice that the target of the association must be an `sc:doc` or one of its child types. The association is not mandatory, and there may be more than one related document:

```
<type name="sc:doc">
  <title>Someco Document</title>
  <parent>cm:content</parent>
  <associations>
   <association name="sc:relatedDocuments">
   <title>Related Documents</title>
   <source>
    <mandatory>false</mandatory>
    <many>true</many>
   </source>
   <target>
    <class>sc:doc</class>
    <mandatory>false</mandatory>
    <many>true</many>
   </target>
   </association>
  </associations>
</type>
```

3. Save the `model` file.
4. Deploy your changes using `ant deploy` and restart Tomcat.

When you restart, watch the log for errors related to the content model. If everything is clean, keep going.

## Associations

Associations define relationships between types. Without associations, models would be full of types with properties that store "pointers" to other pieces of content. Going back to the expense report example, suppose each expense is stored as an individual object. In addition to an Expense Report type, there would also be an Expense type. In this example, associations can be used to tell Alfresco about the relationship between an Expense Report and one or more Expenses.

Here's an important note about the content model schema that may save you some time: Order matters. For example, if a type has both properties and associations, properties go first. If you get the order wrong, Alfresco won't be able to parse your model. There is an XML Schema file that declares the syntax for a content model XML file. It is called `modelSchema.xsd`, and it resides in the Alfresco web application under **WEB-INF | classes | alfresco | model**.

In the `sc:relatedDocuments` association you just defined, note that both the source and target class of the association is `sc:doc`. That's because SomeCo wants to associate documents with each other regardless of content type. Defining the association at the `sc:doc` level allows any instance of `sc:doc` or its children to be associated with zero or more instances of `sc:doc` or its children. It also assumes that SomeCo is using `sc:doc` or children of that type for all of its content. Content stored as the more generic `cm:content` type would not be able to be the target of an `sc:relatedDocuments` association.

Associations come in two flavors: Peer Associations and Child Associations. (Note that Alfresco refers to Peer Associations simply as "Associations", but that's confusing. So the book will use the "Peer" distinction.) Peer Associations are just that—they define a relationship between two objects, but neither is subordinate to the other. Child Associations, on the other hand, are used when the target of the association (or child) should not exist when the source (or parent) goes away. This works like a cascaded delete in a relational database: Delete the parent and the child goes away.

An out of the box association that's easy to relate to is `cm:contains`. The `cm:contains` association defines a Child Association between folders (`cm:folder`) and all other objects (instances of `sys:base` or its child types). So, for example, a folder named `Human Resources` (an instance of `cm:folder`) would have a `cm:contains` association between itself and all of its immediate children. The children could be instances of custom types like Resume, Policy, or Performance Review. If you delete a folder, the folder's children are also deleted.

Another example might be a "Whitepaper" and its "Related Documents". Suppose that SomeCo publishes Whitepapers on its web site. The Whitepaper might be related to other documents such as product marketing materials or other research. If the relationship between the Whitepaper and its related documents is formalized, it can be shown in the user interface. To implement this, as part of the Whitepaper content type, you'd define a Peer Association. You could use `sys:base` as the target type to allow any piece of content in the repository to be associated with a Whitepaper, or you could restrict the association to a specific type. In this case, because it uses a Peer association, related documents don't get deleted when the Whitepaper gets deleted. You can imagine the headaches that would cause if that weren't the case!

# Step-by-Step: Adding Aspects to the Content Model

SomeCo wants to track the client name and, optionally, the project name for pieces of client-related content. But any piece of content in the repository might be client-related. Proposals and Status Reports are both project-related, but the two will be in different parts of the model (one is a type of legal document while the other is a type of operations document). Whether a piece of content is client-related or not, it transcends department—almost anything can be client-related. The grouping of properties that need to be tracked for content that is client-related is called an **aspect**.

Here's another example. SomeCo would like to selectively pull content from the repository to show on its web site. Again, any piece of content could be published on the site. So an indication of whether or not a piece of content is "webable" should be captured in an aspect. Specifically, content that needs to be shown on the web site needs to have a flag that indicates the content is "active" and a date when the content was set to active. These will be the aspect's properties.

Let's modify the content model to include these two aspects. To add the client-related and webable aspects to the content model, follow these steps:

1. Edit the `scModel.xml` file.
2. Add a new `aspects` element below the `types` element to contain the new aspects. Add one `aspect` element to define the client-related aspect and another to define the web-related aspect. You'll notice that the syntax for the `aspect` element is identical to the `type` element:

```
<aspects>
  <aspect name="sc:webable">
   <title>Someco Webable</title>
   <properties>
    <property name="sc:published">
     <type>d:date</type>
    </property>
    <property name="sc:isActive">
     <type>d:boolean</type>
     <default>false</default>
    </property>
   </properties>
  </aspect>
  <aspect name="sc:clientRelated">
   <title>Someco Client Metadata</title>
   <properties>
    <property name="sc:clientName">
     <type>d:text</type>
     <mandatory>true</mandatory>
    </property>
     <property name="sc:projectName">
     <type>d:text</type>
     <mandatory>false</mandatory>
    </property>
   </properties>
  </aspect>
</aspects>
```

3. Save the `model` file.
4. Run `ant deploy` and restart Tomcat.

Alfresco should start cleanly without making any model-related complaints.

## Aspects

To appreciate aspects, first consider how inheritance works and its implications on the content model. Suppose that SomeCo only wants to display a subset of the repository's content on the web site. (In fact, this is the case. The SomeCo write-up in Chapter 1 said that, except for job postings, HR content shouldn't go near the public web.) In the recent example, webable content needs to have a flag that indicates whether or not it is "active", and a date that indicates when it became active.

Without aspects, there would only be two options to model these properties. The first option would be to put the properties on `sc:doc`, the root object. All child content types would inherit from this root type, thus making the properties available everywhere. The second option would be to individually define the two properties only in the content types that will be published to the portal.

Neither of these is a great option. In the first option, there would be properties in each and every piece of content in the repository that may or may not ultimately be used. This can lead to performance and maintenance problems. The second option too isn't much better for several reasons. First, it assumes that the content types to be published to the portal are known when you design the model. Second, it opens up the possibility that the same type of metadata might get defined differently across content types. Third, it doesn't provide an easy way to encapsulate behavior or business logic that might be tied to the published date. Finally, property names must be unique across the model. So you'd have to modify the names of the properties in every type in which they were used, otherwise it would be a serious pain later when you try to run queries across types.

As you already know, the best option is to use aspects. Aspects allow "cross-cutting" of the content model with properties and associations by attaching them to content types (or even specific instances of content at runtime rather than design time) when and where they are needed.

In this case, SomeCo's webable aspect will be added to any piece of content that needs to be displayed on the web site, regardless of type.

Another nice thing about aspects is that they give you a way to have multiple inheritances. As you saw in the model, types can only inherit from a single parent. But you can add as many aspects to a type or object instance as you need.

# Step-by-Step: Finishing up the Model

Let's finish up the model by doing two things: First, the remaining departments need content types added to them. Second, there is an out of the box aspect that needs to be applied to all the content. It's called `generalclassifiable`. It allows content to be categorized. SomeCo wants all of its content to be classifiable as soon as it hits the repository. To make that happen, you need to define the aspect as mandatory. Because SomeCo wants it across the board, you can do it on the root type `sc:doc`, and have it trickle down to all of SomeCo's types.

To add the remaining departmental content types as well as make the `generalclassifiable` aspect mandatory, follow these steps:

1. Edit the `scModel.xml` file.
2. Add the the following types:

```
<type name="sc:hrDoc">
 <title>Someco HR Document</title>
 <parent>sc:doc</parent>
</type>
<type name="sc:salesDoc">
 <title>Someco Sales Document</title>
 <parent>sc:doc</parent>
</type>
<type name="sc:opsDoc">
 <title>Someco Operations Document</title>
 <parent>sc:doc</parent>
</type>
<type name="sc:legalDoc">
 <title>Someco Legal Document</title>
 <parent>sc:doc</parent>
</type>
```

3. Modify the `sc:doc` type to include `cm:generalclassifiable` as a mandatory aspect. Note that you can add as many mandatory aspects as you need:

```
<type name="sc:doc">
 <title>Someco Document</title>
 <parent>cm:content</parent>
 <associations>
 </associations>
 <mandatory-aspects>
 <aspect>cm:generalclassifiable</aspect>
 </mandatory-aspects>
</type>
```

4. Save the `model` file.
5. Run `ant deploy` and restart Tomcat.

Watch the log for content model-related errors. If everything starts up cleanly, you are ready to move on. In the next set of examples, you'll configure the web client so that you can work with your new model.

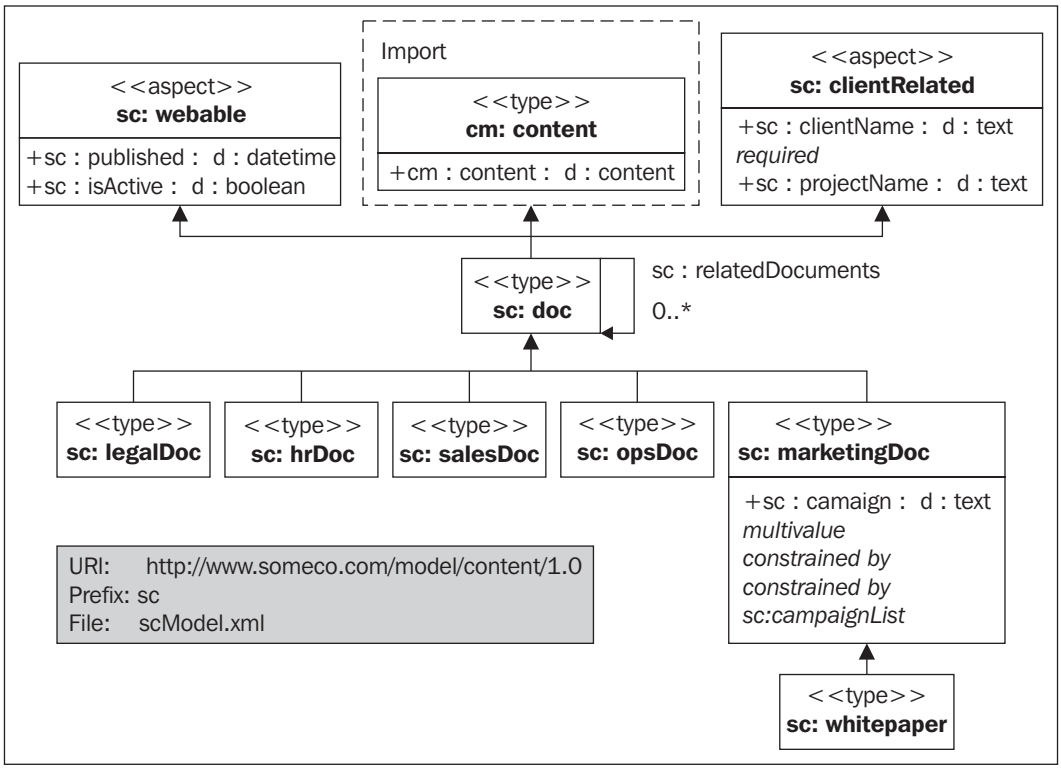
# Modeling Summary

A content model describes the data being stored in the repository. The content model is critical. Without it, Alfresco would be little more than a file system. Here is a list of key information that the content model provides Alfresco:

- Fundamental data types and how those data types should persist to the database. For example, without a content model, Alfresco wouldn't know the difference between a string and a date.
- Higher order data types such as "content" and "folder" as well as custom content types such as "SomeCo Standard Operating Procedure" or "SomeCo Sales Contract".
- Out of the box aspects such as "auditable" and "classifiable" as well as SomeCo-specific aspects such as "rateable", "commentable", or "client-related".
- Properties (or metadata) specific to each content type.
- Constraints placed on properties (such as property values that must match a certain pattern or property values that must come from a specific list of possible values).
- Relationships or "associations" between content types.

Alfresco content models are built using a small set of building blocks: Types, Properties, Property Types, Constraints, Associations, and Aspects. When planning your Alfresco implementation, it may make sense to diagram the proposed content model just as you would a data model in a traditional web application.

The content model implemented in the examples could be diagrammed as follows:




The Appendix contains similar diagrams for the out of the box content models for your reference.

# Custom Behavior

You may find that your custom aspect or custom type needs to have behavior or business logic associated with it. For example, every time an Expense Report is checked in, you might want to recalculate the total by iterating through the associated Expenses. One option would be to incorporate this logic into rules or actions in the Alfresco web client or your custom web application. But some behaviors are so fundamental to the aspect or type that they should really be "bound" to the aspect or type, and invoked any time Alfresco works with those objects. Behavior really gets out of the realm of modeling and into "handling content automatically", which is the subject of Chapter 4. For now, just realize that associating business logic with your custom aspects and types (or overriding out of the box behavior) is possible.

# Modeling Best Practices

Now that you know the building blocks of a content model, it makes sense to consider some best practices. Here are the top ten:

1. **Don't change Alfresco's out of the box content model.** If you can possibly avoid it, do not change Alfresco's out of the box content model. Instead, extend it with your own custom content model. If requirements call for several different types of content to be stored in the repository, create a content type for each one that extends from `cm:content` or from an enterprise-wide root content type.
2. **Consider implementing an enterprise-wide root type.** Although the need for a common ancestor type is lessened through the use of aspects, it still might be a good idea to define an enterprise-wide root content type such as `sc:doc` from which all other content types in the repository inherit, if for no other reason, than it gives content managers a "catch-all" type to use when no other type will do.
3. **Be conservative early on by adding only what you know you need.** A corollary to that is to be prepared to blow away the repository multiple times, until the content model stabilizes. Once you get content in the repository (that implements the types in your model), making model additions is easy, but subtractions aren't. Alfresco will complain about "integrity errors" and may make content inaccessible when the content's type or properties don't match the content model definition. When this happens to you (and it will happen), you can choose one of these options:
   - Leave the old model in place
   - Attempt to export the content, modify the exported data (see "ACP Files" in the Appendix), and re-import
   - Drop the Alfresco tables, clear the data directory, and start fresh

   As long as everyone in the team is aware of this, option three is not a big deal in development. But make sure expectations are set appropriately and have a plan for handling model changes once you get to production. This might be an area where Alfresco will improve in future releases, but for now it is something you have to watch out for.
4. **Avoid unnecessary content model depth.** There don't seem to be any Alfresco Content Modeling Commandments that say, "Thou shall not exceed X levels of depth in thine content model, lest thou suffer the wrath of poor performance". But it seems logical that degradation would occur at some point. If your model has several levels of depth beyond `cm:content`, you should at least do a proof-of-concept with a realistic amount of data, software, and hardware to make sure you aren't creating a problem for yourself that might be very difficult to reverse down the road.

5. **Take advantage of aspects.** In addition to the potential performance and overhead savings through the use of aspects, aspects promote reuse across the model, the business logic, and the presentation layer. When working on your model, you may find that two or more content types have properties in common such as `sc:webable` and `sc:clientRelated`. Ask yourself if those properties are being used to describe some higher-level characteristic common across the types that might be modeled better as an aspect.
6. **It may make sense to define types that have no properties or associations.** You may find yourself defining a type that gets everything it needs through either inheritance from a parent type or from an aspect (or both). In the SomeCo model `sc:marketingDoc` is the only type with a property. You might ask yourself if the empty type is really necessary. It should at least be considered. It might be worth it, just to distinguish the content from other types of content for search purposes, for example. Or, while you might not have any specialized properties or associations for the content type, you could have specialized behavior that's only applicable to instances of the content type.
7. **Remember that folders are types too.** Like everything else in the repository, folders are instances of types, which means they can be extended. Content that "contains" other content is common. In the earlier expense report example, one way to keep track of the expenses associated with an expense report would be to model the expense report as a sub-type of `cm:folder`.
8. **Don't be afraid to have more than one content model XML file.** When it is time to implement your model, keep this in mind: It might make sense to segment your models into multiple namespaces and multiple XML files. Names should be descriptive. Don't deploy a model file called `customModel.xml` or `myModel.xml`.
9. **Implement a Java interface that corresponds to each custom content model you define.** Within each content model Java class, define constants that correspond to model namespaces, type names, property names, aspect names, and so on. You'll find yourself referring to the qualified name (`Qname`, for short) of types, properties, and aspects quite often; so it helps to have constants defined in an intuitive way. The constants should be `QName` objects except in cases where the Web Services API needs to leverage them. The Web Services API doesn't have the `QName` class, so there will need to be a string representation of the names as well in that case.
10. **Use the source!** The out of the box content model is a great example of what's possible. The `forumModel` and `recordsModel` have some particularly useful examples. In the next section you'll learn where the model files live and what's in each. So you'll know where to look later when you say to yourself, "Surely, the folks at Alfresco have done this before".

This last point is important enough to spend a little more time on. The next section discusses the out of the box models in additional detail.

# Out of the Box Models

The Alfresco source code is an indispensable reference tool that you should always have ready along with the documentation, wiki, forums, and Jira. With that said, if you are following along with this chapter but have not yet downloaded the source, you are in luck. The out of the box content model files are written in XML and get deployed with the web client. They can be found in the `alfresco.war` file in **|WEB-INF|classes|alfresco|model**. The following table describes several of the model files that can be found in the directory:

| File | Namespaces* | Prefix | Imports | Description |
|---|---|---|---|---|
| `dictionaryModel.xml` | `model|dictionary|1.0` | d | None | Fundamental data types used in all other models like text, int, Boolean, datetime, and content. |
| `systemModel.xml` | `model|system|1.0`<br>`system|registry|1.0`<br>`system|modules|1.0` | sys<br>reg<br>module | d | System-level objects like base, store root, and reference. |
| `contentModel.xml` | `model|content|1.0` | cm | d<br>sys | Types and aspects extended most often by your models like Content, Folder, Versionable, and Auditable. |
| `bpmModel.xml` | `model|bpm|1.0` | bpm | d<br>sys<br>cm | Advanced workflow types like task and startTask. Extend these when writing your own custom advanced workflows. |
| `forumModel.xml` | `model|forum|1.0` | fm | d<br>cm | Types and aspects related to adding discussion threads to objects like forum, topic, and post. |

The table lists the most often referenced models. Alfresco also includes two WCM-related model files, the JCR model and the web client application model, which may also be worth looking at, depending on what you are trying to do with your model.

In addition to the model files, the `modelSchema.xsd` file can be a good reference. As the name suggests, it defines the XML vocabulary Alfresco content model XML files must adhere to.

# Configuring the UI

Now that the model is defined, you could begin using it right away by writing code against one of Alfresco's APIs that creates instances of your custom types, adds aspects, and so on. In practice, it is usually a good idea to do just that to make sure the model behaves as you expect. But you'd probably like to log in to the web client to see the fruits of your labor from the last section, so let's discuss what it takes to make that happen. By the end of this discussion, you will be able to use the web client to work with the SomeCo-specific content model to do things such as these:

- Display and update custom properties and associations
- Create instances of SomeCo-specific content types
- Configure actions that involve SomeCo types and aspects
- Use Advanced Search to query with SomeCo-specific parameters

Configuring the UI to expose the custom content model involves overriding and extending Alfresco's out of the box web client configuration. To do this, you'll create your own SomeCo-specific version of the web client configuration XML that overrides Alfresco's. For more details on how the out of the box web client configuration is structured and what is available to be extended, refer to the Appendix.

## Step-by-Step: Adding Properties to the Property Sheet

When a user looks at a property sheet for a piece of content stored as one of the custom types or with one of the custom aspects attached, the property sheet should show the custom properties. If there are associations, those should be shown as well:


In order to configure the properties sheet to show custom properties and associations, follow these steps:

1. In the `client-extensions` Eclipse project, create a new XML file called `web-client-config-custom.xml` in the extension directory if it isn't there already. If you are creating it from scratch, populate it with an empty `alfresco-config` element. You'll add child elements to it in the subsequent steps.
2. To add properties to property sheets, use the `aspect-name` evaluator for aspects and the `node-type` evaluator for content types. SomeCo has two aspects that need to be added to the properties sheet: `sc:webable` and `sc:clientRelated`. For `sc:webable`, add the following `config` element to `web-client-config-custom.xml` as a child of `alfresco-config`:

```
<!--  add webable aspect properties to property sheet -->
 <config evaluator="aspect-name" condition="sc:webable">
  <property-sheet>
   <show-property name="sc:published" display-label-id=
                                                    "published" />
   <show-property name="sc:isActive" display-label-id="isActive"
                                                read-only="true" />
  </property-sheet>
 </config>
```

3. Add the `config` element to show the properties for the `clientRelated` aspect on your own.

4. Add the following to display the `relatedDocuments` association for SomeCo documents and Whitepapers:

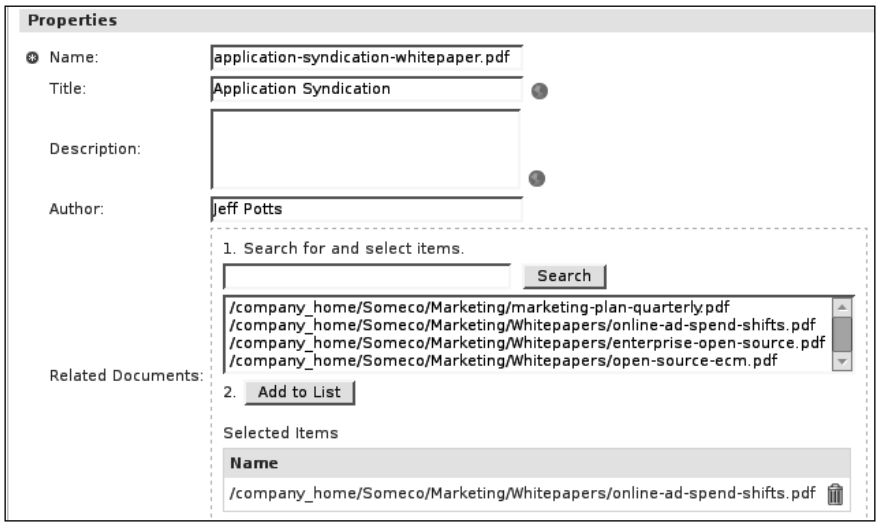
```
<config evaluator="node-type" condition="sc:doc">
 <property-sheet>
  <show-association name="sc:relatedDocuments" />
 </property-sheet>
</config>
<config evaluator="node-type" condition="sc:whitepaper">
 <property-sheet>
  <show-association name="sc:relatedDocuments" />
 </property-sheet>
</config>
```

5. Add the following to display the campaign property for `SomeCo` Marketing Documents:

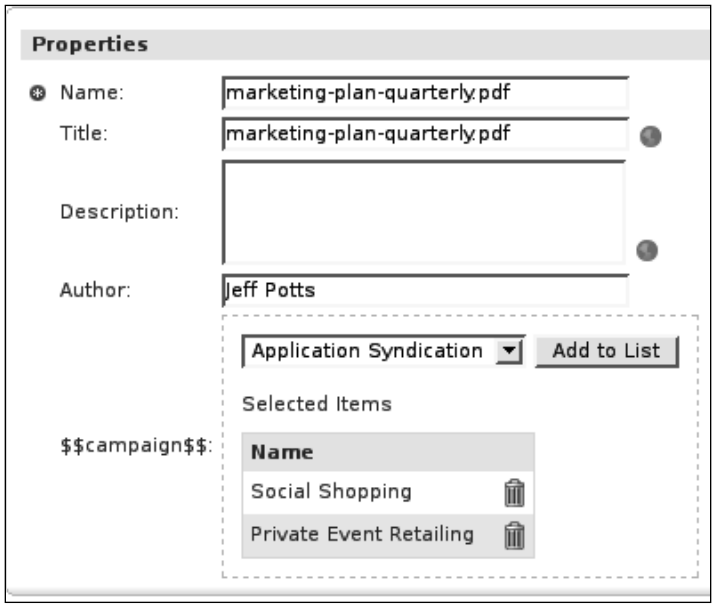
```
<!-- show campaign on maraketingDoc property sheet -->
<config evaluator="node-type" condition="sc:marketingDoc">
 <property-sheet>
  <show-property name="sc:campaign" display-label-id="campaign" />
 </property-sheet>
</config
```

6. Save the file.
7. Deploy the customizations (`ant deploy`), restart Tomcat, and test.

In Chapter 2, you deployed a small set of customizations to validate your development environment. With the web client configuration modifications you just made, you should be able to log in to Alfresco and create instances of Marketing and Whitepaper documents. When you look at the properties for an instance of a Whitepaper, you should see a component that lets you pick **Related Documents**:



When you look at the properties for an instance of a marketing document, you should see a component that lets you choose the associated campaign:




Note that you may see some placeholder text for the **Campaign** label (and corresponding warnings in the log) because you haven't externalized that label yet. We'll fix that shortly.

## Externalizing Display Labels

Note the `display-label-id` attribute of the `show-property` element in the previous code. An alternative is to specify the label explicitly in this file using the `label` attribute. But a better practice is to externalize the string so that the interface can be localized if needed, so this example uses `display-label-id`. The actual label value is defined elsewhere in a resource bundle.

## Making Properties Read-Only

In the previous code, the read-only attribute on the show-property element for `sc:isActive` prevents web client users from editing the property. It does not, however, prevent the property from being set through scripts or API calls. In this case, SomeCo wants it to be "read-only", so that it can be set through other means such as by actions or during an approval step in a workflow.

# Step-by-Step: Adding Types and Aspects to WebClient Dropdowns

When a user clicks **Create** or **Add Content**, the custom types should be a choice in the list of content types. And when a user configures a rule on a space and uses content types or aspects as a criterion or action parameter, the custom types and aspects should be included in the dropdowns.

When you tested your last change you probably noticed that the types you created in Chapter 2 were shown, but not the types you added in this chapter for Legal, HR, Sales, and so on. Similarly, if you tried to add the client-related or webable aspects to an object, you weren't able to select either aspect from the list.

Just as you did for custom properties and associations in the previous example, you have to override Alfresco's web client configuration to get custom types and aspects to show up in the web client.

To add custom types and aspects to the appropriate dropdowns, follow these steps:

1. Edit `web-client-config-custom.xml`.
2. To add content types to the list of available types in the **create content** and **add content** dialogs, use the `string-compare` evaluator and the `Content Wizards` condition. Add the following to `web-client-config-custom.xml` beneath the previous `config` element:

```
<!--  add someco types to add content list -->
config evaluator="string-compare" condition="Content Wizards">
 <content-types>
  <type name="sc:doc" />
  <type name="sc:whitepaper" />
  <type name="sc:legalDoc" />
  <type name="sc:marketingDoc" />++
  <type name="sc:hrDoc" />
  <type name="sc:salesDoc" />
  <type name="sc:opsDoc" />
 </content-types>
</config>
```

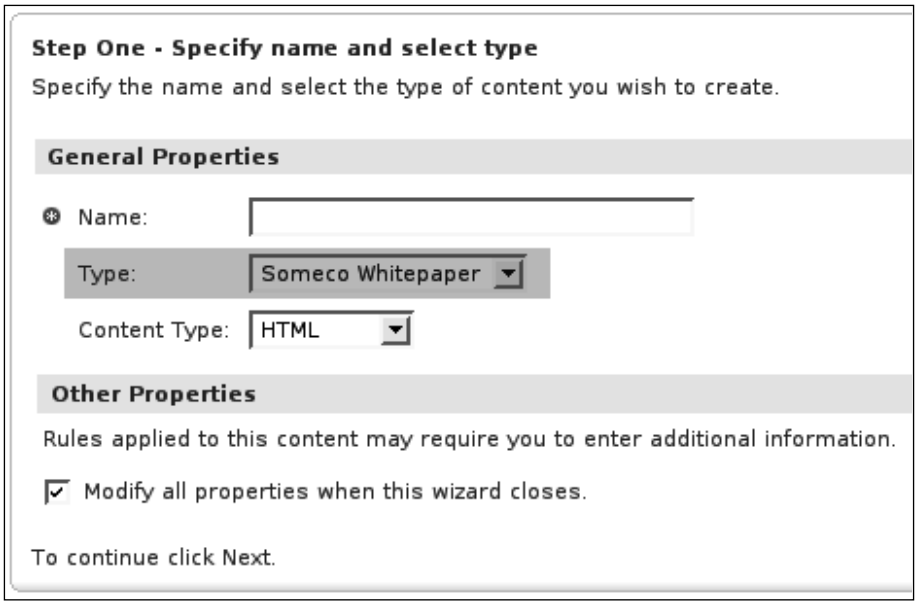
3. The list of types and aspects used when rule actions are configured are all part of the same `config` element. The `Action Wizards` config has several child elements that can be used. The aspects element defines the list of aspects shown when the `add aspect` action is configured. The `subtypes` element lists types that show up in the dropdown when configuring the

content type criteria for a rule. The `specialise-types` element (note the UK spelling) lists the types available to the `specialize type` action. Add the following to `web-client-config-custom.xml` below the previously added `config` element:

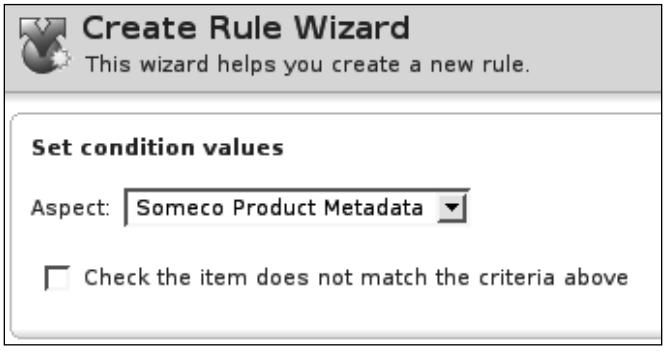
```
<config evaluator="string-compare" condition="Action Wizards">
 <!-- The list of aspects to show in the add/remove features
                                                        action -->
    <!-- and the has-aspect condition -->
  <aspects>
   <aspect name="sc:webable"/>
   <aspect name="sc:clientRelated"/>
  </aspects>
  <!-- The list of types shown in the is-subtype condition -->
  <subtypes>
   <type name="sc:doc" />
   <type name="sc:whitepaper" />
   <type name="sc:legalDoc" />
   <type name="sc:marketingDoc" />
   <type name="sc:hrDoc" />
   <type name="sc:salesDoc" />
   <type name="sc:opsDoc" />
  </subtypes>
   <!-- The list of content and/or folder types shown in the
                                       specialise-type action -->
  <specialise-types>
   <type name="sc:doc" />
   <type name="sc:whitepaper" />
   <type name="sc:legalDoc" />
   <type name="sc:marketingDoc" />
   <type name="sc:hrDoc" />
   <type name="sc:salesDoc" />
   <type name="sc:opsDoc" />
  </specialise-types>
 </config>
```

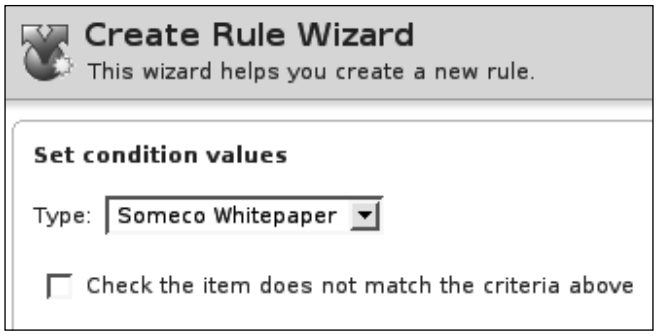
4. Save the `web-client-config-custom.xml` file.
5. Deploy your changes using `ant deploy`, restart Tomcat, and test.

To test these changes out, log in to the web client. Now when you create new content, all of the SomeCo types should be in the content type dropdown:




To test the aspect-related changes, configure a new rule on a space. The first step when defining a rule action is to identify the criteria for running the action. If you select either **Items that have a specific aspect applied** or **Items of a specified type or its sub-types**, you should see the SomeCo custom types when you click **Set Values and Add:**






The SomeCo custom types should also be listed as content type choices for the **specialize type** action, and custom aspects should be listed as choices for the **add aspect** action. To test this, view the details for a folder or a piece of content and then click **Run Action** to launch the rule action wizard. When choosing either the **specialize type** or **add aspect** actions, the list that gets displayed when you click **Set Values** and **Add** should include items from the custom model.

# Step-by-Step: Adding Properties and Types to Advanced Search

When a user runs an advanced search, he or she should be able to restrict search results to instances of custom types and/or content with specific values for the properties of custom types. As before, this involves modifying `web-client-config-custom.xml`.

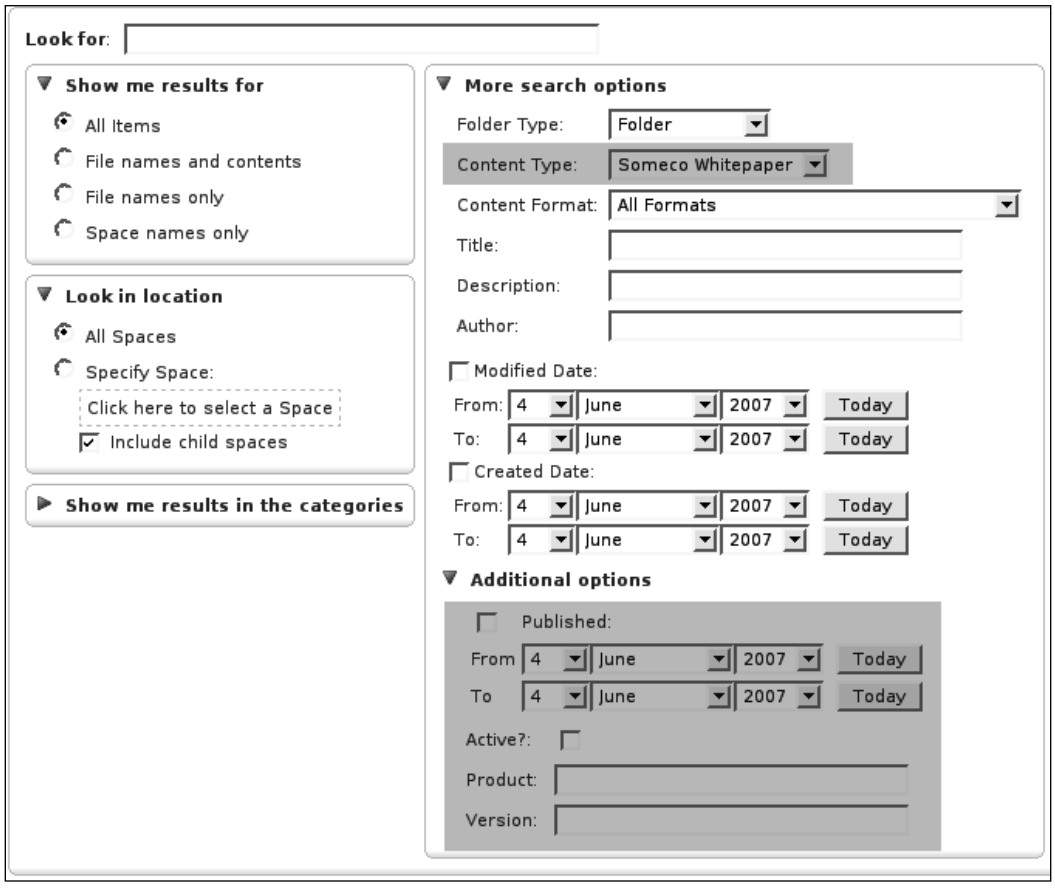
To add custom properties and types to the advanced search dialog, follow these steps:

1. The `Advanced Search config` specifies which content types and properties can be used to refine an advanced search result set. Add the following to `web-client-config-custom.xml` below the previously-added `config` element.

```
<config evaluator="string-compare" condition="Advanced Search">
 <advanced-search>
  <content-types>
   <type name="sc:doc" />
   <type name="sc:whitepaper" />
   <type name="sc:legalDoc" />
   <type name="sc:marketingDoc" />
   <type name="sc:hrDoc" />
   <type name="sc:salesDoc" />
   <type name="sc:opsDoc" />
  </content-types>
```

```
    <custom-properties>
     <meta-data aspect="sc:webable" property="sc:published" display-
      label-id="published" />
     <meta-data aspect="sc:webable" property="sc:isActive" display-
      label-id="isActive" />
     <meta-data aspect="sc:clientRelated" property="sc:clientName"
      display-label-id="client" />
     <meta-data aspect="sc:clientRelated" property="sc:projectName"
      display-label-id="project" />
    </custom-properties>
   </advanced-search>
  </config>
```

2. Deploy the changes by running `ant deploy`, restart Tomcat, and test.
3. To test out this change, log in to the web client and go to **Advanced Search**. The SomeCo types should be listed in the **Content Type** dropdown. The custom properties should be listed under **Additional Options:**



# Step-by-Step: Setting Externalized Label Values

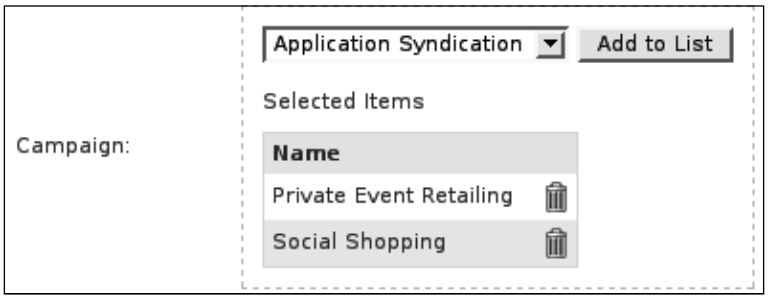
You probably noticed when you were testing the web client changes that all of the labels were showing unresolved label IDs. You need to create a new properties file to fix that. The file will hold name-value pairs that match the `display-label-id` attributes in `web-client-config-custom.xml`.

To configure the label IDs, follow these steps:

1. Create a file called `webclient.properties` in the same directory as `web-client-config-custom.xml`.
2. In this example, there are five properties that need labels. Populate `webclient.properties` as follows.

```
#sc:webable
published=Published
isActive=Active?
#sc:clientRelated
client=Client
project=Project
#sc:marketingDoc
campaign=Campaign
```

3. Deploy your changes by running `ant deploy`, restart Tomcat, and test.
4. Log in to the web client and open the properties for a document. Now the label IDs should be resolving to the externalized string values.




At the time of this writing, there was a bug related to web client configuration property files and AMPs (AWC-1149). If you want to deploy the project as an AMP, your `webclient.properties` file will need to go into **alfresco|extension** instead of your module-specific directory. Until this issue is resolved, this means there is the possibility of an overwrite if the directory already contains a `webclient.properties` file. For more information on deploying AMPs, see the Appendix.

## Setting up Additional Locales

The whole point in externalizing the labels is that the client can be localized to multiple languages. If you want to create a set of labels for a specific locale, you would create a file in the extension directory called `webclient_[locale].properties` with the same keys and their localized values.

# Working with Content Programmatically

Now the repository has a custom model and that model has been exposed to the Alfresco web client. For simple document management solutions, this may be enough. But often, code will also be required as a part of your implementation. It might be code in a web application that needs to work with the repository, code that implements custom behavior for custom content types, code that implements Alfresco web client customizations, or code that implements a controller for a web script.

As mentioned in Chapter 1, there are several APIs available depending on what you want to do. Let's learn how to use code to create content, create associations between content, search for content, and delete content. You'll see a JavaScript example, several examples using the Web Services API with Java, and one example showing the API with PHP. Additional API examples can be found in the Appendix.

# Step-by-Step: Creating Content with JavaScript

The first example shows how to create some content and add aspects to that content using JavaScript.

To create content, add aspects, and set properties using JavaScript, follow these steps:

1. Create a new file in **src | scripts** called `createContent.js`.
2. Set up some variables you'll use later in the script.

```
var contentType = "whitepaper";
var contentName = "sample-a";
var timestamp = new Date().getTime();
```

3. Write code that will create the new node as a child of the current space. The "space" variable is a root object that is available when the script is executed against a folder. Notice the `contentName` and `timestamp` variables are being concatenated to make sure the name is unique on successive runs.

```
var whitepaperNode = space.createNode(contentName + timestamp,
                                      "sc:" + contentType);
```

4. Add a statement that uses the ScriptNode API to add the `sc:webable` aspect.

```
whitepaperNode.addAspect(sc:webable);
```

5. Add code to set some properties. These properties include out of the box properties such as `cm:name` as well as SomeCo-specific properties.

```
whitepaperNode.properties["cm:name"] = contentName + " (" +
                                                    timestamp + ")";
whitepaperNode.properties["sc:isActive"] = true;
whitepaperNode.properties["sc:published"] = new Date("04/01/2007");
```

6. The ScriptNode API can work with the content property directly. Add a statement to store some content on the node, and then call save to persist the changes.

```
whitepaperNode.content = "This is a sample " + contentType + "
                                    document called " + contentName;
whitepaperNode.save();
```

7. Test the script by uploading it to the repository and then running it against a folder. Using the Web Client, add the file to the Data `Dictionary/Scripts` folder.
8. Navigate to the Whitepapers folder. Then do **View Details**, **Run Action**, **Execute Script** to initiate the Run Action Wizard.
9. Use **Set Values and Add** to select `createContent.js` from the available scripts.
10. Click **OK,** and then **Finish** to execute the script.

A new Whitepaper should now be sitting in the folder. Later in the book, you'll add a new UI action that makes executing scripts even easier.

## Leveraging Root Objects

In this example, you used `space` to refer to the space the script was executed against. There is also a `document` root object that can be used when running the script against a document. Refer to the Appendix for the full list of root objects.

## Knowing When to Save Documents

Method calls that affect a node's properties require `save` to persist the changes. If in this example we were only adding an aspect, we wouldn't have to save the document because the change is persisted immediately.

## Using JavaScript for Batch Manipulation

During projects you will often find that you need to perform batch operations on nodes in the repository. You might want to execute an action against all documents in all subfolders starting at a given path, for example. JavaScript is a quick way to perform such mass operations and doesn't require code to be compiled and packaged.

## Writing Content to the Content Property

Properties store data about a node. The terms "metadata" and "attributes" are synonymous with "properties".

Content refers to the main unit of data being managed by the system: a file. A PDF file, for example, is a piece of content. Plain-text data such as XML, HTML, or JavaScript are also examples of content.

Content is stored as a property on the node. In Alfresco, the content is really stored on the file system, but as developers using the API, we don't care about where the content is physically stored. In this example, we created plain-text content simply by writing a string to the content property. This means it is really easy to create content in the repository, especially if it is plain text.

# Creating Content with Java Web Services

JavaScript is fast to develop and very succinct, but it must run on the Alfresco server. Alfresco's Web Services API is one option to consider when you want to run code on a different machine than the Alfresco server.

Let's look at the same task that was in the previous example (creating a SomeCo Whitepaper, adding the `sc:webable` aspect, and setting some properties), but this time using the Java Web Services API. The code used for creating content is almost exactly the same code that comes with the Alfresco SDK Samples, but it is helpful to break it down to see what's going on.

An overview of the steps involved is:

Authenticate to start a session.

1. Get a reference to the folder where the content will be created.
2. Create an array of `NamedValue` objects. Each `NamedValue` object corresponds to a property that will be set on the new object.
3. Create a series of **Content Manipulation Language** (**CML**) objects that encapsulate the operations to be executed.

4. Execute the CML and dump the results.
5. Update the new node with content.

Rather than creating this class yourself, follow along by looking at the class called `com.someco.examples.SomeCoDataCreator` in **src | java** in the source code included with the chapter. Then, you can follow the steps for running this class on your local machine.

`SomeCoDataCreator` is a runnable Java class that accepts arguments for the username, password, and folder in which to create the content, type of content to create, and a name for the new content.

The first thing the code does is to start a session by authenticating with the server:

```
AuthenticationUtils.startSession(getUser(), getPassword());
```

Next, a `timeStamp` is saved. The timestamp will be used to make the content name unique. Then, the code gets a reference to the folder where the content will be created:

```
String timeStamp = new Long(System.currentTimeMillis()).toString();
Store storeRef = new Store(Constants.WORKSPACE_STORE, "SpacesStore");
ParentReference docParent = new ParentReference(
 storeRef,
 null,
 getFolderPath(),
 Constants.ASSOC_CONTAINS,
 Constants.createQNameString(
  SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL,
  getContentName() + timeStamp));
```

Refer to the highlighted code. What is a Store? An Alfresco repository is a collection of stores. In JCR parlance, the stores are called **workspaces**. But in this book, unless the example is dealing with the JCR specifically, they will be referred to as **stores.** A given Alfresco instance has one repository with multiple stores. When you are working with the API, you sometimes have to specify which store you are working with.

Notice that in addition to the folder path, the `ParentReference` constructor used to create the folder reference expects the type of association being created (contains) as well as the name of the child object.

Next, the code creates a `NamedValue` for each property that's going to be set on the new object and then creates an array of all `NamedValue` objects:

```
NamedValue nameValue = Utils.createNamedValue(Constants.PROP_NAME,
getContentName() + " (" + timeStamp + ")");

NamedValue activeValue = Utils.createNamedValue
(Constants.createQNameString(SomeCoModel.NAMESPACE_SOMECO_CONTENT_
MODEL, SomeCoModel.PROP_IS_ACTIVE_STRING), "true");

NamedValue publishDateValue = Utils.createNamedValue(
 Constants.createQNameString(SomeCoModel.NAMESPACE_SOMECO_CONTENT_
MODEL, SomeCoModel.PROP_PUBLISHED_STRING),
 "2007-04-01T00:00:00.000-05:00");

NamedValue[] contentProps = new NamedValue[] {nameValue, activeValue,
publishDateValue};
```

Take a look at the date string (for the curious, it's the ISO 8601 format). That `-05:00` is the GMT timezone offset.

Now CML comes into play. The web services API uses CML objects to encapsulate various content operations. In this case, the example code needs to create a node and add aspects to the node so it uses `CMLCreate` and `CMLAddAspect`.

Note the `ref1` string. That's an arbitrary reference that Alfresco uses to relate the CML statements. Without it, Alfresco wouldn't know which content object to add the aspects to. So if you were creating multiple objects in one shot, for example, you would use unique reference strings for each object. The value isn't persisted anywhere. It is discarded after the CML is executed.

First, the `CMLCreate` object gets created. The `CMLCreate` constructor needs to know the parent reference (`docParent`), the type of content being created, and an array of property values to set:

```
CMLCreate createDoc = new CMLCreate(
 "ref1",
 docParent,
 null,
 null,
 null,
 Constants.createQNameString(SomeCoModel.NAMESPACE_SOMECO_CONTENT_
MODEL, SomeCoModel.TYPE_SC_DOC_STRING),
 contentProps);
```

Then, one `CMLAddAspect` object gets created for each aspect to be added:

```
CMLAddAspect addWebableAspectToDoc = new CMLAddAspect
(Constants.createQNameString(SomeCoModel.NAMESPACE_SOMECO_CONTENT_
MODEL, SomeCoModel.ASPECT_SC_WEBABLE_STRING),
```

```
 null,
 null,
 "ref1");
CMLAddAspect addClientRelatedAspectToDoc = new CMLAddAspect(
 Constants.createQNameString(SomeCoModel.NAMESPACE_SOMECO_CONTENT_
MODEL, SomeCoModel.ASPECT_SC_CLIENT_RELATED_STRING),
 null,
 null,
 "ref1");
```

To execute the CML, the code instantiates a new CML object. Setters on the CML object specify the operations to perform, in this case one document creation and two aspect additions. The code then uses the `RepositoryService` to run the update, which passes back an array of `UpdateResults`. The `dumpUpdateResults` method just iterates through the `UpdateResult` array and writes some information to sysout:

```
// Construct CML Block
CML cml = new CML();
cml.setCreate(new CMLCreate[] {createDoc});
cml.setAddAspect(new CMLAddAspect[] {addWebableAspectToDoc,
addClientRelatedAspectToDoc});
// Execute CML Block
UpdateResult[] results = WebServiceFactory.getRepositoryService().
update(cml);
Reference docRef = results[0].getDestination();
dumpUpdateResults(results);
```

Now the node exists, but it doesn't have any content. The last chunk of code writes some text content to the newly created node. This example uses a string for the content, but it could just as easily write the bytes from a file on the local file system:

```
// Nodes are created, now write some content
ContentServiceSoapBindingStub contentService = WebServiceFactory.
getContentService();
ContentFormat contentFormat = new ContentFormat("text/plain", "UTF-
8");
String docText = "This is a sample " + getContentType() + " document
called " + getContentName();
Content docContentRef = contentService.write(docRef, Constants.PROP_
CONTENT, docText.getBytes(), contentFormat);
System.out.println("Content Length: " + docContentRef.getLength());
```

As you can see, this code accomplishes exactly the same end result as the JavaScript example; but it is a bit more verbose.

## Step-by-Step: Run SomeCoDataCreator Class to Create Content

To run the SomeCoDataCreator class to create some content, follow these steps:

1. Copy the `com.someco.examples.SomeCoDataCreator.java` file into **src | java** within your client-extensions project in Eclipse.
2. The Web Services API needs to know the hostname of the Alfresco server that the remote classes are communicating with. Create a file called `webserviceclient.properties` in the client-extensions project's **config | alfresco | extension** directory. Assuming both your Alfresco server and your code reside on the same machine, the file should look like this:

   ```
   # Set the following property to reference the Alfresco server
   that you would like the web service client to communicate with

   repository.location=http://localhost:8080/alfresco/api
   ```
3. If you haven't already, log in to Alfresco and create the following folder structure in your repository: **Someco | Marketing | Whitepapers**.
4. Execute the class by running `ant data-creator`. The `Ant` target will compile and execute the class.

If everything is successful, the result should be something like:

```
Command = create; Source = none; Destination = b901941e-12d3-11dc-
9bf3-e998e07a8da1
Command = addAspect; Source = b901941e-12d3-11dc-9bf3-e998e07a8da1;
Destination = b901941e-12d3-11dc-9bf3-e998e07a8da1
Command = addAspect; Source = b901941e-12d3-11dc-9bf3-e998e07a8da1;
Destination = b901941e-12d3-11dc-9bf3-e998e07a8da1
Content Length: 26
```

If you decide to use Eclipse or command-line Java to run the class rather than the `Ant` target, make sure you have the `webserviceclient.properties` file on your classpath or the Web Services API will not be able to locate the Alfresco server.

# Creating Content with PHP Web Services

Java is not a requirement for SOAP-based web services. Alfresco also delivers PHP classes that use the Web Services API. Here's how the "create content and add aspects" example would look like in PHP:

```
<?php
  require_once "Alfresco/Service/Session.php";
  require_once "Alfresco/Service/SpacesStore.php";
```

```
  require_once "Alfresco/Service/Node.php";

...snip...

  function createContent($username, $password, $folderPath,
$contentType, $contentName) {

 // Start and create the session
 $repository = new Repository("http://localhost:8080/alfresco/api");
 $ticket = $repository->authenticate($username, $password);
$session = $repository->createSession($ticket);

 $store = new Store($session, "SpacesStore");

 // Grab a reference to the SomeCo folder
 $results = $session->query($store, 'PATH:"' . $folderPath . '"');
 $rootFolderNode = $results[0];

 if ($rootFolderNode == null) {
  echo "Root folder node (" . $folderPath . ") is null<br>";
  exit;
 }

 $timestamp = time();

 $newNode = $rootFolderNode>createChild
("{http://www.someco.com/model/content/1.0}" . $contentType, "cm_
contains", "{http://www.someco.com/model/content/1.0}" . $contentType
. "_" . $timestamp );

 if ($newNode == null) {
  echo "New node is null<br>";
  exit;
 }

 // Add the two aspects
 $newNode->addAspect("{http://www.someco.com/model/content/
1.0}webable");
 $newNode->addAspect("{http://www.someco.com/model/content/
1.0}clientRelated");

 echo "Aspects added<br>";

 // Set the properties
 $properties = $newNode->getProperties();

 $properties["{http://www.alfresco.org/model/content/1.0}name"] =
$contentName . " (" . $timestamp . ")";
 $properties["{http://www.someco.com/model/content/1.0}isActive"] =
"true";
 $properties["{http://www.someco.com/model/content/1.0}published"] =
"2007-04-01T00:00:00.000-05:00";

 $newNode->setProperties($properties);
```

```
 echo "Props set<br>";
 $newNode->setContent("cm_content", "text/plain", "UTF-8", "This is a
sample " . $contentType . " document named " . $contentName);
 echo "Content set<br>";
 $session->save();
 echo "Saved changes to " . $newNode->getId() . "<br>";
  }
?>
```

```
Running the PHP script in a web browser produces:
Aspects added
Props set
Content set
Saved changes to 5a8dac5e-1314-11dc-ab93-3b56af79ba48
```

The PHP file lives in the `src/php` folder in the `client-extensions` Eclipse project included with the source. See the Appendix for instructions on how to set up your environment to run this PHP example.

The rest of the chapter includes Java Web Services API examples. Refer to the source code that will be provided in Chapter 6 and the Appendix for additional JavaScript examples.

# Creating Associations

Now let's switch back to the Java Web Services API and look at a class that creates a related-documents association between two documents.

The high-level steps are essentially the same as in the earlier Java example:

1. Create the references and objects the CML needs.
2. Set up the CML objects.
3. Execute the CML and dump the results.

This class is called `com.someco.examples.SomeCoDataRelater`. The class is runnable and accepts a source UUID and a target UUID as arguments. You can get them from the output of the `SomeCoDataCreator` class.

After logging in, the code creates references to the source and target using the UUIDs passed in as arguments:

```
 Reference docRefSource = new Reference(storeRef, getSourceUuid(),
null);
 Reference docRefTarget = new Reference(storeRef, getTargetUuid(),
null);
```

Then, the code creates a `CMLCreateAssociation` object. The constructor accepts predicate objects, which are easily created using the reference objects, and the type of association being created:

```
 CMLCreateAssociation relatedDocAssoc = new CMLCreateAssociation(new
Predicate(new Reference[]{docRefSource}, null, null),
  null,
  new Predicate(new Reference[] {docRefTarget}, null, null),
  null,   Constants.createQNameString(SomeCoModel.NAMESPACE_SOMECO_
CONTENT_MODEL,
   SomeCoModel.ASSN_RELATED_DOCUMENTS_STRING));
```

The rest should look very familiar. A new CML object is instantiated and its `setCreateAssociation` method is called with an array of `CMLCreateAssociation` objects. In this case, there is only one association being created:

```
 // Setup CML block
 CML cml = new CML();
 cml.setCreateAssociation(new CMLCreateAssociation[]
{relatedDocAssoc});
```

Then, the Repository Service executes the CML and returns the array of `UpdateResults`, which get passed to the `dumpUpdateResults` method:

```
// Execute CML Block
 UpdateResult[] results = WebServiceFactory.getRepositoryService().
update(cml);
 dumpUpdateResults(results);
```

Just to confirm everything worked out as expected, the code calls a method to dump the associations of the source object:

```
 System.out.println("Associations of sourceUuid:" + getSourceUuid());
 dumpAssociations(docRefSource, Constants.createQNameString
(SomeCoModel.NAMESPACE_SOMECO_CONTENT_MODEL, SomeCoModel.ASSN_RELATED_
DOCUMENTS_STRING));
```

## Step-by-Step: Run SomeCoDataRelater Class to Create Association

To run the `SomeCoDataRelater` class to create an association between two objects, follow these steps:

1. Copy the `com.someco.examples.SomeCoDataRelater.java` file from the chapter source code to your client-extension project.
2. If you haven't already done so, make sure you have created at least two instances of `sc:doc` in your repository. A fast way to do that is to run `ant data-creator` a couple of times. Make sure you note the source UUID from the console output.

3. Run the class using the `Ant` task called `data-relater`. The `Ant` task accepts two arguments, `${sourceId}` and `${targetId}` for the source and target UUIDs. For example,

```
ant data-relater -DsourceId=1355e60e-160b-11dc-a66f-bb03ffd77ac6
-DtargetId=bd0bd57d-160c-11dc-a66f-bb03ffd77ac6
```

Running the `SomeCoDataRelater` Java class produces:

```
Command = createAssociation; Source = 1355e60e-160b-11dc-a66f-
bb03ffd77ac6; Destination = bd0bd57d-160c-11dc-a66f-bb03ffd77ac6
Associations of sourceUuid:1355e60e-160b-11dc-a66f-bb03ffd77ac6
bd0bd57d-160c-11dc-a66f-bb03ffd77ac6
{http://www.alfresco.org/model/content/1.0}name:Test Document 2
(1181340487582)
```

Now you can use the Alfresco Web Client to view the associations. Remember the `web-client-config-custom.xml` file? It specified that the property sheet for `sc:doc` or `sc:whitepaper` objects should show the `sc:relatedDocuments` associations. Alternatively, the Node Browser that is available in the Administration Console is a handy way to view associations.

# Searching for Content

Now that you have some content in the repository, you can test out Alfresco's full-text search engine, **Lucene**. Content in the repository is synchronously indexed by Lucene when it is created. Query strings use the Lucene query syntax to find content based on full-text content, property values, path, and content type.

Let's review some code that will show several different examples of Alfresco queries using Lucene. The code will:

1. Authenticate to start a session.
2. Get a reference to the node where the search should start.
3. Establish a query object using the Lucene query string.
4. Execute the query and dump the results.

The class is called `com.someco.examples.SomeCoDataQueries`. Just like the content creation code, the class will be a runnable Java application that accepts the username, password, and folder name as arguments.

There are two methods of interest in this class: `getQueryResults()` and `doExamples()`. The `getQueryResults()` method is a generic method that executes a specified query string and returns a list of `ContentResult` objects. (`ContentResult` is an inner class that is used as a helper to manage the query result properties). The `doExamples()` method calls `getQueryResults()` repeatedly to show different search string examples.

Let's take a look at `getQueryResults()`. First, the code sets up the query object and executes the query using the `query()` method of the `RepositoryService`:

```
public List<ContentResult> getQueryResults(String queryString) throws
Exception {
 List<ContentResult> results = new ArrayList<ContentResult>();
 Query query = new Query(Constants.QUERY_LANG_LUCENE, queryString );
 // Execute the query
 QueryResult queryResult = getRepositoryService().query(getStoreRef(),
query, false);
 // Display the results
 ResultSet resultSet = queryResult.getResultSet();
 ResultSetRow[] rows = resultSet.getRows();
```

Next, the code iterates through the results, extracting property values from the search results and storing them in a helper object called `contentResult`.

```
 if (rows != null) {
  // Get the infomation from the result set
  for(ResultSetRow row : rows) {
   String nodeId = row.getNode().getId();
   ContentResult contentResult = new ContentResult(nodeId);
   // iterate through the columns of the result set to extract
   // specific named values
   for (NamedValue namedValue : row.getColumns()) {
    if (namedValue.getName().endsWith(Constants.PROP_CREATED) == true)
{
  contentResult.setCreateDate(namedValue.getValue());
      } else if (namedValue.getName().endsWith(Constants.PROP_NAME) ==
true) {
     contentResult.setName(namedValue.getValue());
    }
   }
   results.add(contentResult);
  } //next row
 } // end if
 return results;
}
```

The `doExamples()` method sets up query strings and calls `getQueryResults()`. One such call is shown here:

```
System.out.println("Finding content of type:" +
 SomeCoModel.TYPE_SC_DOC_STRING);
queryString = "+TYPE:\"" +
 Constants.createQNameString(SomeCoModel.NAMESPACE_SOMECO_CONTENT_
MODEL,
  SomeCoModel.TYPE_SC_DOC_STRING) + "\"";
dumpQueryResults(getQueryResults(queryString));
```

## Step-by-Step: Run SomeCoDataQueries Class to See Lucene Example

Running the `SomeCoDataQueries` class is a good way to see some example Lucene search strings and the resulting output:

1. Copy the `com.someco.examples.SomeCoDataQueries.java` file from the source code to the client-extensions project.
2. Create one or more instances of `sc:doc` in the repository by running `SomeCoDataCreator` or by adding content to the repository manually (remember to choose a SomeCo content type).
3. Execute the `data-queries` `Ant` task.

Your results will vary based on how much content you've created and the values you've set in the content properties. The output should look something like:

```
======================
Finding content of type:doc
----------------------
Result 1:
id=1355e60e-160b-11dc-a66f-bb03ffd77ac6
name=Test Whitepaper (1181339773331)
created=2007-06-08T16:56:13.932-05:00
----------------------
Result 2:
id=bd0bd57d-160c-11dc-a66f-bb03ffd77ac6
name=Test Document 2 (1181340487582)
created=2007-06-08T17:08:08.150-05:00
----------------------
Result 3:
id=1fe9cf04-160b-11dc-a66f-bb03ffd77ac6
name=Test Document (1181339794431)
created=2007-06-08T16:56:35.028-05:00
======================
Find content in the root folder with text like 'sample'
----------------------
Result 1:
id=bd0bd57d-160c-11dc-a66f-bb03ffd77ac6
name=Test Document 2 (1181340487582)
created=2007-06-08T17:08:08.150-05:00
----------------------
Result 2:
id=1fe9cf04-160b-11dc-a66f-bb03ffd77ac6
name=Test Document (1181339794431)
```

```
created=2007-06-08T16:56:35.028-05:00
----------------------
Result 3:
id=1355e60e-160b-11dc-a66f-bb03ffd77ac6
name=Test Whitepaper (1181339773331)
created=2007-06-08T16:56:13.932-05:00
======================
Find active content
----------------------
Result 1:
id=bd0bd57d-160c-11dc-a66f-bb03ffd77ac6
name=Test Document 2 (1181340487582)
created=2007-06-08T17:08:08.150-05:00
----------------------
Result 2:
id=1fe9cf04-160b-11dc-a66f-bb03ffd77ac6
name=Test Document (1181339794431)
created=2007-06-08T16:56:35.028-05:00
----------------------
Result 3:
id=1355e60e-160b-11dc-a66f-bb03ffd77ac6
name=Test Whitepaper (1181339773331)
created=2007-06-08T16:56:13.932-05:00
======================
Find active content with a client property containing 'Lebowski'
======================
Find content of type sc:whitepaper published between 1/1/2006 and
6/1/2007
----------------------
Result 1:
id=1355e60e-160b-11dc-a66f-bb03ffd77ac6
name=Test Whitepaper (1181339773331)
created=2007-06-08T16:56:13.932-05:00
```

There are a couple of other useful tidbits in this class that have been omitted here such as how to use the `ContentService` to get the URL for the content and how the UUID for the root folder is retrieved. Explore the code that accompanies this chapter to see the class in its entirety.

See the Appendix for more information on the Lucene search syntax.

# Deleting Content

Now it is time to clean up after yourself by deleting content from the repository. Deleting follows the same pattern as searching except that instead of dumping the results, the class will create `CMLDelete` objects for each result and then will execute the CML to perform the delete.

Let's review the `com.someco.examples.SomeCoDataCleaner` class. This runnable class optionally accepts a content type and a folder path to narrow down the scope of what's being deleted.

First, the code sets up the query object:

```
// Create a query object, looking for all items of a particular type
String queryString = "TYPE:\"" + Constants.createQNameString(SomeCoMod
el.NAMESPACE_SOMECO_CONTENT_MODEL, getContentType()) + "\"";
                if (getFolderPath() != null) queryString =
queryString + " AND PATH:\"" + getFolderPath() + "/*\"";
            Query query = new Query(Constants.QUERY_LANG_LUCENE,
queryString);
```

Then, the `RepositoryService` executes the query:

```
// Execute the query
QueryResult queryResult = repositoryService.query(storeRef, query,
false);

// Get the resultset
ResultSet resultSet = queryResult.getResultSet();
ResultSetRow[] rows = resultSet.getRows();
```

A `CMLDelete` object is created for each row returned and added to an array:

```
// if we found some rows, create an array of DeleteCML objects
if (rows != null) {
   System.out.println("Found " + rows.length + " objects to
delete.");

 CMLDelete[] deleteCMLArray = new CMLDelete[rows.length];
 for (int index = 0; index < rows.length; index++) {
  ResultSetRow row = rows[index];
  deleteCMLArray[index] = new CMLDelete(new Predicate(new Reference[]
{new Reference(storeRef, row.getNode().getId(), null)}, null, null));
 }
```

As in prior examples, the final step is to set up the CML object, execute the CML using the `RepositoryService`, and dump the results:

```
// Construct CML Block
CML cml = new CML();
cml.setDelete(deleteCMLArray);

// Execute CML Block
UpdateResult[] results =
 WebServiceFactory.getRepositoryService().update(cml);
dumpUpdateResults(results);
} //end if
```

Note that this code deletes every matching object in the repository (or the specified folder path) of type `sc:doc` (or the specified content type) and its children. You would definitely want to "think twice and cut once" if you were running this code on a production repository!

## Step-by-Step: Running SomeCoDataCleaner Class to Delete Content

To execute the `SomeCoDataCleaner` class to delete content from your repository, follow these steps:

1. Copy the `com.someco.examples.SomeCoDataCleaner.java` file from the source code for the chapter to your `client-extensions` project.
2. Running this class isn't too exciting if there isn't any content in the repository. Create some if you don't have any. The `Ant` task assumes you will create one or more `sc:whitepaper` objects.
3. Run the `data-cleaner` `Ant` task.

Again, your results will vary based on the content you've created. The output should look similar to the following:

```
Found 2 objects to delete.
Command = delete; Source = b6c3f8b0-12fb-11dc-ab93-3b56af79ba48;
Destination = none
Command = delete; Source = d932365a-12fb-11dc-ab93-3b56af79ba48;
Destination = none
```

# Summary

This chapter was about customizing Alfresco's content model, configuring the web client to allow end users to work with the custom content model via the web client, and using the Web Services API and JavaScript API to create, search, update, and delete objects in the repository. Specifically, you learned:

- The Alfresco repository is a hierarchical collection of stores and nodes.
- The Alfresco content model defines the data types of nodes and properties, and the relationships between nodes.
- Extending the content model to make it relevant to your business problem involves creating an XML file to describe the model, then telling Alfresco about it through a Spring bean configuration file.
- The fundamental building blocks used to define the content model include: Types, Aspects, Properties, and Associations.
- Best practices for creating your own content models include using aspects as much as possible, considering the use of a root content type, and leveraging the out of the box content model as a reference.
- Configuring the web client to expose your custom content model via the user interface involves overriding configuration elements in Alfresco's out of the box web client configuration.

There are several options for interacting with the repository with code. Examples in this chapter included the Web Services API (both PHP and Java) and the JavaScript API.

## Where to buy this book

You can buy Alfresco Developer Guide from the Packt Publishing website:
`http://www.packtpub.com/alfresco-developer-guide/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.


**www.PacktPub.com**