

# C

## Installation, AMPs, and Surf

### Installation and Configuration

When compared to installing the products of many commercial, legacy vendors, Alfresco installation is a breeze, but there are a few common installation and configuration questions that always pop up:

- First timers may need a few pointers getting Alfresco going with their database of choice. So the *Basic Installation* section includes some tips.
- Many people run multiple versions of Alfresco on their development workstations. For them, a few tips are provided in *Running Multiple Versions of Alfresco*.
- Chapter 3 showed some PHP examples, but didn't go into the details about its setup, and so those are provided in *Setting Up the PHP API*.
- Running web script backed portlets in a JSR-168 portal like Liferay Portal is becoming a frequently recurring pattern. In *Running Alfresco and Liferay in the Same Process*, you'll get step-by-step instructions for running Alfresco and Liferay together.

### Basic Installation

The core Alfresco distribution comes in two flavors: a bundle that includes Alfresco as well as Apache Tomcat, and a deployable WAR. Which one you use doesn't matter, but most developer workstations and production servers already have an application server. So, WAR is the most popular option for real work, while the Alfresco-Tomcat bundle is used most often by people who are evaluating the software.

To install the bundle, just uncompress the archive, and then run the start-up script. To install the WAR file, use your normal application server deployment mechanism. For example, for Apache Tomcat, you'd copy your WAR file to `$TOMCAT_HOME|webapps` and then start up Tomcat.

## Step-by-Step: Changing the Default Database and Data Directory

By default, the bundle will use the embedded Hypersonic database, whereas the WAR installation will attempt to find a MySQL database named **alfresco** with a username and password also set to **alfresco**.

Alfresco stores files and Lucene indexes on the file system. This is sometimes referred to as the Alfresco "data directory". Mostly, you'll explicitly set the path of the data directory.

To change the configuration to point to a different database and specify a file path for the data directory, do the following:

1. If you are running the Alfresco-Tomcat bundle, you already have an extension directory under `$ALFRESCO_HOME|tomcat|shared|classes|alfresco|extension`. If you are running the Alfresco WAR and don't have one already, create a directory named `extension` under `$TOMCAT_HOME|shared|classes|alfresco`.
2. If you are running the Alfresco-Tomcat bundle, the configuration files you need to modify are already in your extension directory. If you are running the Alfresco WAR and they aren't in there already, copy the following files from the Alfresco distribution's `extensions|extension` directory into `$TOMCAT_HOME|shared/classes|alfresco|extension`:
  - `custom-hibernate-dialect.properties`
  - `custom-repository-context.xml`
  - `custom-repository.properties`
3. There are two files you need to change. First, edit `custom-hibernate-dialect.properties`. Comment out the HSQL entry, then uncomment the dialect appropriate for your database, and then save the file.
4. Now edit `custom-repository.properties`. Change the `dir.root` property to point to the location you want to use as your Alfresco data directory. For example, on my machine, I use `|srv|alfresco-2.2-enterprise|data`. (Or some suitable alternative that matches the Alfresco version I am using. See [Running Multiple Versions of Alfresco](#) to understand why.)

5. Change the `db.name`, `db.username`, and `db.password` properties to match the settings appropriate to your database.
6. The final change is to set the appropriate database connection string. Comment out the HSQL connection string. Uncomment the `db.driver` and `db.url` properties appropriate to your database. In my configuration, I always change the hardcoded database name, `alfresco`, to use the `db.name` property as follows:

```
db.url=jdbc:mysql://localhost/${db.name}
```

7. Depending on the version of the MySQL database and driver, you may have to add `?useServerPrepStmts=false`, as shown below. If you don't know what this is and you aren't seeing any error messages, ignore it:

```
db.url=jdbc:mysql://localhost/${db.name}?useServerPrepStmts=false
```

Assuming you've copied your database driver into Tomcat's classpath (`$TOMCAT_HOME | common | lib` will work), Alfresco should now leverage your specific database and data directory.

## Running Multiple Versions of Alfresco

At any given time, many developers have at least three versions running: the latest Enterprise release, the Enterprise head, and the Labs head. If you are a consultant and one or more of your clients are running older releases, it is likely that you'll need to run those older releases as well.

Everyone has their favorite way to configure their development environments. Here's one way that works for me:

- Use a single Tomcat instance that can run a given Alfresco version at a time.
- Use a single MySQL instance with one database for each version of Alfresco. Use a naming convention to separate the databases.
- Create version-specific Alfresco data directories that follow a naming convention.
- Create one Tomcat extension directory for each version named according to a naming convention.
- Create one Alfresco web application directory that contains the expanded web application that follows a naming convention.
- Write a shell script that creates (and deletes) links to the appropriate set of directories based on the version of Alfresco you want to run.



Symbolic links aren't just for Linux and Unix OSs. If you are running Windows, you can do symbolic links too—they are called Junctions. See <http://technet.microsoft.com/en-us/sysinternals/bb896768.aspx> for more information.

For example, for your databases, you could use "alfrescoXXn", where "XX" is the version number and "n" is the network abbreviation ("l" for Labs, "e" for Enterprise, and so on). So the database for 2.2 Enterprise would be "alfresco22e".

For directories, you could use the same abbreviation or something more verbose like "alfresco-X.X-nnnn". So, the directories for Alfresco 2.2 Enterprise would be:

- | home | jpotts | alfresco | alfresco-2.2-enterprise | data
- | home | jpotts | alfresco | alfresco-2.2-enterprise | webapp
- | home | jpotts | alfresco | alfresco-2.2-enterprise | extension
- | home | jpotts | alfresco | alfresco-2.2-enterprise | virtual

When you want to switch between versions, you can call a shell script with the version as the argument like this:

```
~/alf-switch.sh 2.2-enterprise
```

The shell script takes care of deleting the existing links and creating the new ones:

```
echo Switching to $1
echo Switching webapp...
rm $TOMCAT_HOME/webapps/alfresco
ln -s /home/jpotts/alfresco/alfresco-$1/webapp $TOMCAT_HOME/webapps/
alfresco
echo Switching extension directory...
rm $TOMCAT_HOME/shared/classes/alfresco/extension
ln -s /home/jpotts/alfresco/alfresco-$1/extension $TOMCAT_HOME/shared/
classes/alfresco/extension
echo Clearing Tomcat work directory...
rm -rf $TOMCAT_HOME/work/Catalina/localhost/alfresco
echo Clearing Tomcat temp directory...
rm -rf $TOMCAT_HOME/temp/*
echo Switching virtual tomcat directory...
rm $VIRTUAL_TOMCAT_HOME
ln -s /home/jpotts/alfresco/alfresco-$1/virtual $VIRTUAL_TOMCAT_HOME
```

Note that in this set up, the data directory and the Alfresco database don't have to be dealt with directly by the shell script. This is because they are referenced by the `custom-repository.properties` file residing in the version-specific extension directory.

---

It's also handy to write a "clean" script that cleans out your data directory and then runs the `db_remove` and `db_setup` SQL scripts. Remember that you'll either need version-specific copies of these scripts, or you'll need to modify them to accept the version-specific database as an argument.

With this setup in place, you can set up new versions, switch between versions, and start over with a clean repository quickly and easily.

## Setting Up the PHP API

Once you get all of the pieces together, it isn't that hard to run the Alfresco PHP samples. But it's not extremely well-documented and so this section includes some pointers on getting it set up.

### Step-by-Step: Using the Alfresco PHP API

To configure PHP and Alfresco's PHP API, follow these steps:

1. Download PHP 5 from <http://www.php.net>. Follow the directions for building, configuring, and making PHP. **When you configure PHP, make sure to include the enable-soap flag.** You'll also need to uncomment the `extension=php_soap.dll` line in your `php.ini` file.
2. The Alfresco PHP API on the download site may not be the latest version. You can always get the latest version of the Alfresco PHP API from the source tree under `modules`.
3. Follow the instructions in `modules | php-sdk | source | php | remote | installation.txt` to set up the samples. This will vary depending on your workstation setup. As an example, this is my setup:
  - I created a directory called `|usr|local|lib|alfresco`.
  - I created an alias and a Directory entry in `httpd.conf` that points to `|usr|local|lib|alfresco` as per the `installation.txt` file.
  - I copied the contents of `modules | php-sdk | source | php | remote` to `|usr|local|lib|alfresco`.
  - I then added `|usr|local|lib|alfresco` to the `include_path` in my `php.ini` file.

If your Alfresco repository is running and you've restarted Apache with the updated `httpd.conf` and `php.ini`, you should be ready to test. If everything is OK, when you point your browser to <http://localhost/alfresco/Examples/SimpleBrowse>, you will see a list of the spaces and content in your Alfresco store. You can navigate the list, open documents, and so on.

The other example is the QueryExecuter example. It performs a full-text query and presents a node list of the query results.

Both of these are very simple examples. But by looking at the code, the PHP API source under the Alfresco directory, and the Alfresco Web Services SDK, you should be able to see how you can incorporate Alfresco into your PHP-based application.

## **Alfresco and Web Script-Backed Portlets**

Combining Alfresco with a portal is a common way to build community sites that need more robust content management functionality than portals provide on their own. A nice feature of the web script framework is that web scripts are automatically made available as JSR-168 portlets. The fine print is that in the current release this is only automatic when the portal and the repository are running in the same process. That won't be a preferred architecture for many, who prefer to separate the presentation from the repository. But for departmental implementations, it is probably OK.

## **Step-by-Step: Running Alfresco and Liferay in the Same Process**

Suppose you wanted to build a community site that serves up documents stored in Alfresco. You could write Java against the Java Portlet API, which would leverage the Alfresco Web Services API, and that is a good approach. Web scripts offer a faster development cycle, however. The steps that come next show how to integrate Alfresco with a popular open source portal called Liferay. After completing these steps, you should be able to run the out of the box web scripts such as the web client portlet or the **My Spaces** portlet within the context of the Liferay portal.

1. Download the Liferay 4.3.6 + Tomcat 5.5 JDK5 bundle. You may also be able to use 4.4.2 or 5.x. You may be tempted to try to download the WAR-only distribution of Liferay and configure it in your existing Tomcat instance. But the WAR-only distribution of Liferay is a little more complex to configure than the WAR-only distribution of Alfresco. Unless you've done it before, save yourself the time and headache and get the bundle. Once you get Alfresco and Liferay working together, you can circle back and deal with the WAR distribution later.
2. Unpack the Liferay distribution and fire it up. Make sure you can log in as the test@liferay.com (**password: test**) user to validate that all is well with the Liferay installation.

3. Create a test user. (**Create Account** on the Liferay login screen). Make the screen name and email address match the username and email address of one of your existing Alfresco test users. For this example, let's assume you created a user with a screen name of **tuser1** and an email address of `tuser1@someco.com`. Make sure you create a home directory. In this example, we'll call it `tuser1`.
4. Verify that you can log in to Liferay as your test user.
5. Shut down the server.
6. Expand your Alfresco WAR into Liferay's **webapps | alfresco** directory (which you'll have to create the first time). If you are tweaking the install (such as pointing to a specific MySQL database, using something other than MySQL, pointing to a different data directory, and so on), make sure you have copied your good set of extensions into the **shared | classes | alfresco | extension** directory under Liferay's Tomcat installation.
7. Copy the MySQL connector into Tomcat's **common | lib** directory.
8. Start Tomcat. When it comes up, you'll have Liferay as well as Alfresco running, but Liferay won't yet know about Alfresco.
9. If you don't already have a test user in Alfresco, create one now. You need to create a user account that has an email address that matches the test user account you created in Liferay. In this example, you created Test User1 with a screen name of **tuser1** and an email address of `tuser1@someco.com`. So you need to create an Alfresco user with the same settings. You'll log in to Alfresco as **tuser1**. You'll log in to Liferay as `tuser1@someco.com`.
10. Verify that you can log in to Alfresco as **tuser1**.
11. Shut down Tomcat.
12. Now you need to configure Alfresco as a Liferay plug-in. This involves adding four files to Alfresco's WEB-INF directory: `liferay-display.xml`, `liferay-plugin-package.xml`, `liferay-portlet.xml`, and `portlet.xml`. These files are not available as part of the Alfresco source. There are sample versions of the files together with sample `faces-config.xml` and `web.xml` files in the source code that accompanies this chapter in the `someco-server-extensions` project under **config | liferay**. You can also check the Alfresco wiki for sample versions of these files.
13. Remove the `portlet-api-lib.jar` file from Alfresco's **WEB-INF | lib** directory.
14. Re-package `alfresco.war`. It is now ready to hand over to Liferay.
15. Start Tomcat.

16. Find your Liferay deploy directory. If you are running out of the box on Linux, Liferay's deploy directory is called **liferay | deploy**. It resides in the home directory of the user who started Tomcat. On my machine, I'm running it as root and so my Liferay deploy directory is **|root|liferay|deploy**.
17. Copy the `alfresco.war` file you just created into the deploy directory. Watch the log. You should see Liferay working on the WAR, finding the plug-in config files, and essentially deploying the Alfresco portlets.
18. Now log in to Liferay using the Liferay admin account (`test@liferay.com`). Go to a page, and then use the global navigation drop-down to select **Add Content**. The list of portlets should appear and you should see the **Alfresco** category. If you don't, look at the log because something is amiss. Add the **My Spaces** portlet to the page. You may see an error at this point, but ignore it for now. The problem is probably that you don't have a user in Alfresco who has an email address of `test@liferay.com`, which is the currently logged in user.
19. Log out.
20. Log in as your test user who exists in both Alfresco and Liferay (`tuser1@someco.com`).
21. Go to the page to which you added the portlet in the earlier step. You should see the **My Spaces** portlet. You should be able to upload content, create spaces, and so on.

## Exposing Your Own Web Scripts as Portlets

All Alfresco Web Scripts are automatically exposed as JSR-168 portlets, including the ones you create. To add your web scripts as portlets, first make sure you have authentication set to **user** and transaction set to **required** in your web script's descriptor. Then, update `portlet.xml`, `liferay-portlet.xml`, and `liferay-display.xml`. Follow the pattern that's already in those files and you'll be fine. For example, if you deploy the Hello World web script from Chapter 6, you need to add a new portlet to `portlet.xml` with a `scriptUrl` such as **|alfresco|168s|someco|helloworld?name=jeff**. Then, update `liferay-portlet.xml` and `liferay-display.xml` with the new portlet name or portlet ID.

## Single Sign-On with No Single Sign-On

The web script runtime has a JSR-168 authenticator. So when your web scripts get invoked by the portlet, the current credentials are passed in. That's why your web script can run without requiring an additional sign in. Prior to putting this in place, people had to implement JA-SIG CAS (or something similar) to get SSO between Liferay and Alfresco Web Scripts. (See Chapter 9 for more information on using CAS with Alfresco.)

What's not covered in these instructions is that you'll probably want to (1) configure both Alfresco and Liferay to authenticate against LDAP (see Chapter 9 for information on configuring Alfresco to authenticate against LDAP) and (2) change the configuration of either Alfresco or Liferay to use the same credential (either username or email address) for both systems so that if you do have users logging in to both, they don't have to remember that one requires the full email address while the other doesn't.

## Tools and Utilities

There are a couple of Alfresco-provided utilities that you should know about. The first one is the **Alfresco Module Package (AMP)** and its associated tools. AMPs provide a way of packaging everything you need for a particular add-on or module into a single distribution file and tools that know how to manage multiple add-ons in the context of a deployable Alfresco WAR file.

The second useful tool is an **Alfresco Content Package (ACP)**. An ACP is really just a ZIP with an `acp` extension plus an XML manifest. ACPs are used to import and export data into and out of the repository.

## The Power of AMPs

In general, one of the non-AMP package and deployment options will be the best for development and most production scenarios. However, if you are creating a reusable module—maybe it is something you are contributing to the community, for example—it might make more sense to package your customizations as an AMP. Note that an AMP is often referred to in the more generic sense as a "module", and the two will be used interchangeably here.

## What is an AMP?

An AMP is a ZIP with an `amp` extension that contains all files needed to customize Alfresco for a given set of functionality. Records Management, Facebook Integration, and Blog Integration are all examples of the functionality that has been packaged as an AMP. The idea is that everything needs to be packaged in one self-contained, easy-to-share archive.

An AMP file might include any or all of the types of customizations discussed earlier in this chapter. In addition, it might include data that you want uploaded into the repository when the module is installed.

## AMP Advantages

Packaging up customizations as an AMP and integrating it with an Alfresco WAR is little more trouble than the standard web application approaches previously discussed. Is it worth the extra trouble? As usual, it depends. To help you decide, consider the advantages:

- AMPs offer a namespace that keeps some (but not all) of the files from colliding with other customizations. This is nice if you've chosen non-unique names for your files such as `myCustomModel.xml` and more than one set of customizations needs to be installed into the same Alfresco instance.
- The **Module Management Tool (MMT)**, which is used to install AMPs into an Alfresco WAR, is able to interrogate an Alfresco WAR to figure out which, if any, AMPs have already been applied. Again, if multiple sets of customizations are being installed, this could be a real time saver.
- AMPs can declare the minimum and maximum version of Alfresco they require. They can also declare other modules on which they depend. And, AMP modules themselves can be versioned. This kind of dependency management simply isn't available with the straight-copy techniques. On a related note, future versions may allow modules to automatically check if updates are available as is the case with a Mozilla plug-in or a Linux package.
- If your customizations include content that needs to be available when the module is loaded, the content can be included in the AMP. When the AMP is installed, the content will be uploaded to the repository. Examples of this might include end-user consumable content or it could be things such as Freemarker templates or server-side JavaScript files, which need to be placed in the Data Dictionary.
- Sometimes it is helpful to be able to run some initialization code when an AMP is installed. For example, maybe you want to check for the existence of a folder or category and create whatever is missing. Or perhaps you'd like to create a few groups if they aren't there. AMP initialization code has full access to the Alfresco API.

Although there are advantages, it is important to realize that the "M" in AMP doesn't stand for "Magical". If two modules need to change the same file, an AMP won't save you. There is no diff, merge, or patch capability within the tool as of yet. The good news is that AMP's MMT does make a backup copy of the Alfresco WAR. So if something does get stepped on, it can be recovered easily.

---

## AMP File Structure

Files in an AMP are organized in a standard directory structure. When AMPs are installed into an Alfresco WAR, the MMT simply maps the standard directories to directories within the Alfresco WAR, and then copies the files and subdirectories into the WAR. The standard directory structure for an AMP is shown here:

```
/config
  /alfresco
    /extension
    /module
      /arbitrary.package.id
        /bootstrap
        /model
        /scripts
        /ui
  /lib
  /licenses
  /web
    /jsp
    /css
    /images
    /scripts
```

Note that everything under the `config` directory will be mapped to the **WEB-INF | classes** directory in the Alfresco WAR. Everything under `web` goes into the root of the Alfresco WAR, including subdirectories. For example, in the directory listing above, the **| config | alfresco | extension** directory and everything under it will get mapped to **WEB-INF | classes | alfresco | extension**.

The `module` directory is special. It contains a directory named using the module's ID. By convention, this ID should begin with a reverse domain name, just like Java packages, to make it unique. This is what is going to keep the bulk of your customizations separate from everyone else's.

It helps others figure out how your module is built if you stick to the standard structure. But if you need to stray for some reason, that's not a problem. AMP files can include a file mapping that tells Alfresco how to map the directories in the AMP to the directories in the Alfresco WAR.

## Required AMP Content

There are two files that must be present in an AMP: `module.properties` and `module-context.xml`. The `module.properties` file specifies things such as the module's ID, version, title, and description, which are all required. For example:

```
module.id=com.someco.module.extensions.Core
module.version=1.0
module.title=SomeCo Extensions
module.description=SomeCo Extensions
```

Module dependencies are also expressed in this file. The module might have a dependency on a specific Alfresco version or it might be dependent on other modules:

```
module.repo.version.min=2.1
module.depends.com.someco.module.ImageLibrary=1.0-*
module.depends.com.someco.module.SlickWidgets=*
```

In this case, the module declares that it requires Alfresco 2.1 or higher, version 1.0 and later of `com.someco.module.ImageLibrary` and any version of `com.someco.module.SlickWidgets`.

The `module.properties` file resides in the root of the AMP archive.

The second required file is `module-context.xml`. As the name implies, this is a Spring bean configuration file. All the beans within it will get initialized when Alfresco starts up and loads the module. This is where you would declare a bean to run some initialization code or import some content. It is also where you must import other context files that may exist within your AMP. For example, the following `module-context.xml` file points to a custom actions context file that resides in **|alfresco|module|com.someco.module.extensions.Core** within the AMP:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" 'http://www.
springframework.org/dtd/spring-beans.dtd'>
<beans>
    <!-- Import the actions context -->
    <import resource="classpath:alfresco/module/com.someco.module.
extensions.Core/someco-actions-context.xml" />
</beans>
```

The `module-context.xml` file must reside in **|alfresco|module|[module ID]** within the AMP. Make sure the module ID specified in `module.properties` matches the module ID directory.

---

## Changing the AMP's Default File Mapping

The `file-mapping.properties` file is used when the AMP doesn't strictly conform to the standard directory structure. It consists of a set of name-value pairs. The left side is the path in the AMP file. The right side is the Alfresco web application directory. The `include.default` flag states whether the default mappings should be extended (`true`) or if you are defining an entirely new mapping (`false`). For example, suppose rather than using `/images` and `/scripts` for your images and JavaScript in the web application, you wanted to use `/someco/images` and `/someco/javascript`. You would need a `file-mapping.properties` file that looked like this:

```
include.default=true
/web/images=/someco/images
/web/scripts=/someco/javascript
```

If you wanted to package your AMP using a non-standard directory structure `/web/someco/images` instead of `/web/images`, for example, you could do so by adjusting the mapping as:

```
/web/someco/images=/someco/images
```

The `file-mapping.properties` file resides in the root of the AMP.

## AMP Package and Deployment Example

What if you wanted to deploy the same two customizations you made earlier (the custom login page and the extended content model) as an AMP? Let's try it.

First, understand this: Nothing about the customizations or even the Eclipse project structure has to change significantly. The differences between the straight-copy approach and the AMP approach are that:

- There is a standard directory structure the archive will conform to.
- An Alfresco tool will be used to merge the archive with the Alfresco WAR. The mechanics are no different than the straight-copy approach.

In a real-world project, you won't need to have a project that can be deployed using either the straight-copy approach or as an AMP. You will pick one or the other. Based on the choice, you might choose to alter the Eclipse project folder structure. In this example, there is already a project set up. The goal is to now see how it could be packaged as an AMP. Rather than shift the Eclipse project folder structure to match up to the AMP standard, in this case, it is easier to leave things as they are and just use Ant to build the AMP archive with the right structure.

The only thing that needs to be added to what was built in the previous example is the module ID folder. The module ID folder will hold the module-specific configuration files. The module ID directory is called `/config/alfresco/module/[module ID]`. This ID could be anything. But for this example, use `com.someco.module.extensions.Core` as the module ID. Given that, you need to create a new folder in your Eclipse project according to this structure. Eclipse might give you a hard time with the dot (".") character, as part of the directory name. It may try to convert the module ID to a package folder structure. If so, create the folder outside of Eclipse and then refresh the project.

Three files will go into the module ID folder: `module.properties`, `module-context.xml`, and `file-mapping.properties`. You've already seen the `module.properties` file and the `file-mapping.properties` as sample code listings earlier in the chapter. The `module-context.xml` file is worth a look, though.

The `module-context.xml` file is the root Spring bean configuration file for the module. In the non-AMP approach, custom configuration and Spring beans get picked up automatically because they reside in the `alfresco/extension` directory, and Alfresco has been configured to look there. But the addition of a (hopefully) globally unique module ID directory makes that impossible. Somehow, Alfresco needs to find out about the extensions in the module directory. That's where the `module-context.xml` file comes in.

In this example, it needs to tell Alfresco about the custom web client configuration and the custom model file. The custom web client configuration is referenced like this:

```
<!-- web client configuration bootstrap bean -->
<bean id="somecoModuleCore.configBootstrap" class=
"org.alfresco.web.config.WebClientConfigBootstrap" init-
method="init">
  <property name="configs">
    <list>
      <value>classpath:alfresco/module/com.someco.module.
extensions.Core
      /ui/web-client-config-custom.xml</value>
    </list>
  </property>
</bean>
```

Note that the path isn't where the file exists in Eclipse. It is the path where the file will exist in the AMP archive.

For the content model there are two options. First, you could simply import the `someco-model-context.xml` file created in the earlier example as follows:

```
<!-- Import the model context -->
<import resource="classpath:alfresco/module/ com.someco.module.
extensions.Core/someco-model-context.xml" />
```

If the project has a large number of beans to configure, this is a good approach because the bean configurations can be spread across multiple configuration files, and then imported into the `module-context.xml` as shown above.

In this case, though, there is just one bean: `dictionaryBootstrap`. The problem is that it references the custom model file, `scModel.xml`, by path. In the non-AMP deployment, the file resides in `alfresco/extension/model`. In the AMP deployment, the file will reside under the module ID directory per the standard. That means if we used the import approach to pull the `dictionaryBootstrap` bean into the `module-context.xml` file, the path in the `model-context.xml` file would have to be updated every time you decided to switch between AMP and non-AMP deployments. That's a pain.

To address this, the `dictionaryBootstrap` bean will simply be duplicated in the `module-context.xml`. When the Ant task packages a non-AMP deployment, it will use the `someco-model-context` file. When it packages the AMP, it will ignore the `someco-model-context.xml` file because the `dictionaryBootstrap` is being configured in `module-context.xml`. There are other ways to handle it, but, believe it or not, this seemed like the most straightforward approach.

To configure the `dictionaryBootstrap` bean, the `module-context.xml` needs to be updated with:

```
<bean id="someco.dictionaryBootstrap" parent="dictionaryModelBootstrap"
" depends-on="dictionaryBootstrap">
  <property name="models">
    <list>
      <value>alfresco/module/com.someco.module.extensions.
Core/model/
      scModel.xml</value>
    </list>
  </property>
</bean>
```

This is the exact same bean from `someco-model-context.xml`, but with the module ID-specific path to the model file.

Now that the AMP-related configuration files have been added to the module ID directory, it is time to package the AMP, install the AMP into the Alfresco WAR file, and then deploy it.

The MMT is used to install AMPs into the Alfresco WAR. You can either use the tool from the command line or from an Ant task. The easiest way to use the command line is to use the `alfresco-mmt.jar`. Check the SDK's `build/dist` directory. If it isn't in there already, you can build it by running the SDK's **distribute-mmt** Ant target like this:

```
ant -f continuous.xml distribute-mmt
```

You can then get the usage by running the **alfresco-mmt** JAR like this:

```
java -jar alfresco-mmt.jar
```

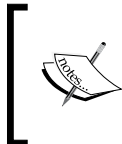
For this example, it is easier to call the MMT from Ant. To do this, the `build.xml` file included with this chapter's source code has been updated with a few new AMP-related targets. The descriptions of the new Ant targets are shown in the following table:

Ant Target	What it Does
<b>package-amp</b>	Packages the customizations as an Alfresco module in <code>\${package.file.amp}</code>
<b>install-amp</b>	Uses the Alfresco MMT to install the AMP into <code>\${alfresco.war.path}</code>
<b>deploy-amp</b>	Unzips the AMP's WAR file into <code>\${alfresco.web.dir}</code>

The **package-amp** target creates a ZIP of the customizations that conforms to the AMP folder structure. The resulting `someco-alfresco.amp` file in the build directory is the installable module file you would upload to an open source module directory, for example.

In this example, though, you'll want to go ahead and install the module into your Alfresco WAR. The **install-amp** Ant target does just that by calling the MMT. Although the MMT makes a backup copy of the Alfresco WAR, it is a good idea to use a copy of the `alfresco.war` file from your distribution rather than your "good" copy of the Alfresco WAR.

The result of running the Ant **install-amp** target is a deployable Alfresco WAR that now has your module installed in it. You can either deploy the WAR to your application server, or, if you are running an exploded Alfresco web application in Tomcat, you can use the **deploy-amp** step to unzip the modified WAR into the web application root directory.



If you ran the non-AMP package and deployment example earlier, you should run the **clean-tomcat** Ant target before running the **deploy-amp** target. The **clean-tomcat** target cleans out the customizations that were copied into the exploded Alfresco web app directories.

For example, assuming you have copied your Alfresco WAR into your **client-extensions | build** directory, you can build the AMP, install the AMP into the Alfresco WAR, and deploy the modified Alfresco WAR in one step by running:

```
ant deploy-amp -Dalfresco.war.path=[path to your Eclipse workspace]/
client-extensions/build/alfresco.war
```

When you start up Tomcat, you should see messages in the log similar to the following when Alfresco loads the module:

```
23:43:31,653 User:System INFO [repo.module.ModuleServiceImpl] Found 1
module(s) .
23:43:31,707 User:System INFO [repo.module.ModuleServiceImpl]
Starting module 'com.someco.module.extensions.Core' version 1.0.
```

When you log in, you should see the same thing you saw in the non-AMP example—a modified login page and two new content types in the content type drop-down.

## Bootstrapping Data and Initialization Logic with an AMP

The previous example showed how to package and deploy the same set of customizations as the non-AMP example. But, as you've seen, AMPs can do more than package customizations. They can also be used to "bootstrap" or import data as well as execute initialization logic.

In this example, suppose you want to do two things. In addition to the customized login page and extended content model, you want to:

- Create some users and groups if they aren't in the repository already
- Upload some content to the repository

### Writing Custom AMP Initialization Logic

Creating users and groups is possible through the Alfresco API. Remember `module-context.xml`? It is the root Spring configuration file for the module. Any beans configured in it get initialized when the module is loaded. All you need, then, is a bean that creates the appropriate users and groups. A bean that does that has already been created for you and is included in the source code for this chapter.

The class resides in **java | src** and is called `com.someco.module.BootstrapAuthorityCreator`. It uses various Alfresco services available as part of the Alfresco API to create users, create groups, and then populate the groups with the users. The details of those API calls aren't as important right now as how you configure the bean as part of the module. To do that, update `module-context.xml` with the following:

```
<bean id="somecoModuleCore.bootstrapAuthorityCreator" class="com.
someco.module.BootstrapAuthorityCreator" init-method="init">
  <property name="personService">
    <ref bean="personService" />
  </property>
  <property name="authorityService">
    <ref bean="authorityService" />
  </property>
  <property name="transactionService">
    <ref bean="transactionService" />
  </property>
  <property name="authenticationService">
    <ref bean="authenticationService" />
  </property>
</bean>
```

That's all there is to it. The `BootstrapAuthorityCreator` class will reside in a JAR file packaged as part of the module. When Alfresco loads the module, the bean will initialize by calling the `init` method. Java code within the `init` method will use the Alfresco Java API to create users, create groups, and then populate the groups with users.

## Importing Content into the Repository when an AMP Is Loaded

What if you wanted to import some content into the repository as part of our module? A script might need to go into the Data Dictionary or you might want to create some folders and upload some content, for example.

One way to get that to happen is to first export the content out of the repository as an ACP file. The ACP file can then be packaged as part of the module. It is common to have bootstrap data as part of your project, so go ahead and create a **data directory** in the root of your project. An ACP file has already been created for you called `whitepapers.acp`. The **package-amp** Ant target will move the ACP file to a module-specific location within the AMP archive.

The final step is to update the `module-context.xml` file with a new bean that tells Alfresco to import the ACP file when the module is initialized. In this case, Alfresco already has a bean that will handle the import for you called `org.alfresco.repo.module.ImporterModuleComponent`. Here is what needs to be added to the `module-context.xml` file:

```
<bean id="somecoModule.bootstrap" class="org.alfresco.repo.module.
ImporterModuleComponent" parent="module.baseComponent">
  <!-- Module Details -->
  <property name="moduleId" value="com.someco.module.
extensions.Core" />
  <property name="name" value="somecoModuleBootstrap" />
  <property name="description" value="Someco module extensions
initial data
requirements" />
  <property name="sinceVersion" value="1.0" />
  <property name="appliesFromVersion" value="1.0" />
  <!-- Data properties -->
  <property name="importer" ref="spacesBootstrap"/>
  <property name="bootstrapViews">
    <list>
      <props>
        <prop key="path">/${spaces.company_home.
childname}</prop>
        <prop key="location">
          alfresco/module/com.someco.
module.extensions.Core
          /bootstrap/whitepapers.acp</prop>
      </props>
    </list>
  </property>
</bean>
```

Note the `sinceVersion` and `appliesFromVersion` properties. Use these if the data to be imported is specific to a particular version of the module. The `path` property tells the importer where in the repository to import the data. In this case, the ACP is being imported into Company Home.

This time, if you deploy the AMP and restart Tomcat, you should see four new users named Test UserX (where X is a number from 1 to 4), a Sales group with Test User1 and Test User2 as members, and a Marketing group with Test User3 and Test User4 as members. You should be able to log in as any of the test users using username **userX** and **password** as the password. There should also be a test PDF sitting in the **| Company Home | Someco | Whitepapers** folder.

By the way, if you happen to clear out the **WEB-INF | classes | alfresco | module** directory and restart, you are effectively yanking the modules out of Alfresco. But Alfresco remembers the installed modules. On startup, you are likely to see messages like:

```
23:46:43,555 User:System WARN [repo.module.ModuleServiceImpl] A
previously-installed module 'com.someco.module.extensions.Core'
(version 1.0) is not present in your distribution.
```

Unfortunately, you may have to live with these. In the current releases, there is no way to uninstall or disable a module.

## ACP Files

**Alfresco Content Packages (ACP)** are one way to import and export content into and out of the repository.

## What Is an ACP File?

An ACP file is really just a ZIP with the extension changed from `.zip` to `.acp`. The ZIP contains content as well as metadata. The metadata is expressed in an XML file.

What's interesting is that the metadata is not just the user-visible metadata for content nodes. It includes every bit of data Alfresco knows about the node, which could include things such as permissions, rules, and associations.

This means that through an ACP file it is possible to create an exact copy of a repository or a piece of repository in some other Alfresco instance.

The following is a small subset of the XML manifest included in an exported repository just to give you an idea of what it looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<view:view xmlns:view="http://www.alfresco.org/view/repository/1.0">
  <view:metadata>
    <view:exportBy>admin</view:exportBy>
    <view:exportDate>2007-07-10T18:05:05.137-05:00</view:exportDate>
    <view:exporterVersion>2.0.1 (54)</view:exporterVersion>
    <view:exportOf></view:exportOf>
  </view:metadata>
  <view:reference xmlns:act="http://www.alfresco.org/model/
action/1.0" xmlns:app="http://www.alfresco.org/model/application/1.0"
xmlns:ver="http://www.alfresco.org/model/versionstore/1.0" xmlns:
mix="http://www.jcp.org/jcr/mix/1.0" xmlns:jcr="http://www.jcp.
org/jcr/1.0" xmlns:wcm="http://www.alfresco.org/model/wcmmodel/1.0"
xmlns:wcmwf="http://www.alfresco.org/model/wcmworkflow/1.0" xmlns:
```

---

```

rule="http://www.alfresco.org/model/rule/1.0" xmlns:fm="http://www.
alfresco.org/model/forum/1.0" xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
xmlns:alf="http://www.alfresco.org" xmlns:d="http://www.alfresco.
org/model/dictionary/1.0" xmlns:view="http://www.alfresco.org/view/
repository/1.0" xmlns:wf="http://www.alfresco.org/model/workflow/1.0"
xmlns:usr="http://www.alfresco.org/model/user/1.0" xmlns:cm="http://
www.alfresco.org/model/content/1.0" xmlns:sv="http://www.jcp.org/
jcr/sv/1.0" xmlns:sys="http://www.alfresco.org/model/system/1.0"
xmlns:wca="http://www.alfresco.org/model/wcmappmodel/1.0" xmlns:
scwf="http://www.someco.com/model/workflow/1.0" xmlns:bpm="http://www.
alfresco.org/model/bpm/1.0" xmlns:sc="http://www.someco.com/model/
content/1.0" xmlns:reg="http://www.alfresco.org/system/registry/1.0"
xmlns:module="http://www.alfresco.org/system/modules/1.0" xmlns=""
view:pathref="/">
  <view:aspects>
    <sys:aspect_root></sys:aspect_root>
  </view:aspects>
  <view:acl>
    <view:ace view:access="ALLOWED">
      <view:authority>GROUP_EVERYONE</view:authority>
      <view:permission>Read</view:permission>
    </view:ace>
  </view:acl>
  <view:properties>
    <sys:node-uuid>9f2d00f9-12cb-11dc-bce2-f5831fa23b69</sys:node-
uuid>
    <sys:node-dbid>776</sys:node-dbid>
    <sys:store-protocol>workspace</sys:store-protocol>
    <cm:name>9f2d00f9-12cb-11dc-bce2-f5831fa23b69</cm:name>
    <sys:store-identifier>SpacesStore</sys:store-identifier>
  </view:properties>
  <view:associations>
    <sys:children>
      <cm:folder view:childName="app:company_home">
        <view:aspects>
          <cm:auditable></cm:auditable>
          <app:uifacets></app:uifacets>
          <sys:referenceable></sys:referenceable>
        </view:aspects>
        <view:acl view:inherit="false">
          <view:ace view:access="ALLOWED">
            <view:authority>GROUP_EVERYONE</view:authority>
            <view:permission>Consumer</view:permission>
          </view:ace>
        </view:acl>
        <view:properties>

```

```
node-uuid> <cm:description>The company root space</cm:description>
<app:icon>space-icon-default</app:icon>
<sys:node-uuid>9f92527b-12cb-11dc-bce2-f5831fa23b69</sys:
node-uuid>
<sys:node-dbid>777</sys:node-dbid>
<cm:title>Company Home</cm:title>
<cm:created>2007-06-04T13:44:28.073-05:00</cm:created>
<cm:modifier>admin</cm:modifier>
<cm:modified>2007-06-05T08:47:49.801-05:00</cm:modified>
<cm:creator>System</cm:creator>
<sys:store-protocol>workspace</sys:store-protocol>
<cm:name>Company Home</cm:name>
<sys:store-identifier>SpacesStore</sys:store-identifier>
</view:properties>
<view:associations>
  <cm:contains>
    <cm:folder view:childName="app:dictionary">
      <view:aspects>
        <cm:auditable></cm:auditable>
        <app:uifacets></app:uifacets>
        <sys:referenceable></sys:referenceable>
      </view:aspects>
      <view:acl view:inherit="false">
        <view:ace view:access="ALLOWED">
          <view:authority>GROUP_EVERYONE</view:authority>
        </view:ace>
      </view:acl>
    </cm:folder>
  </cm:contains>
</view:associations>
...

```

## Web Client Example

ACP files are imported and exported through the admin console. This isn't obvious the first time you do it, but the import/export is contextual, based on where you were in the repository when you entered the admin console. So, for example, if you want to export the `SomeCo` folder, you need to be sitting in the `SomeCo` folder before you go to the admin console to click the **Export** link.

Note that when you perform an export, the ACP file gets created and stored in the repository. It is then up to you to download the ACP file out of the repository.

## Generating ACP Files with Non-Alfresco Tools

Because an ACP file is really just a ZIP'd set of binaries with metadata expressed as XML, it means you can create your own ACP files using your tool of choice (mine's Perl, but to each his or her own) and then import them into the repository. This is a handy way to generate an ACP with several test files. Or maybe another system

wants to put data in the repository, but can't use CIFS, FTP, WebDAV, or one of the APIs for some reason. The system could create an ACP file and then place the file on a file share. Then, some other process could use the Alfresco API to import the incoming ACP files.

The ability to work with ACP files external to the repository can also be leveraged to make mass changes to the repository. For example, you could export a portion of the repository, parse and transform the metadata XML to make the changes you need, and then re-import the ACP to put the changes into effect.

## **Alfresco 3.0**

At the time of this writing, Alfresco 3.0 had not yet been officially released in either Enterprise or Labs. The big change with 3.0 is a new web application development framework called Surf and a new set of web clients built with Surf. Although Surf is new, you should already be familiar with the basic building blocks because it is based on the web script framework. This section will provide a brief look at Surf so that you'll have an understanding of what's coming. Rest assured that the web script, JavaScript API, and Foundation API work you've done in this book will be immediately applicable to 3.0 and Surf as soon as it becomes available.

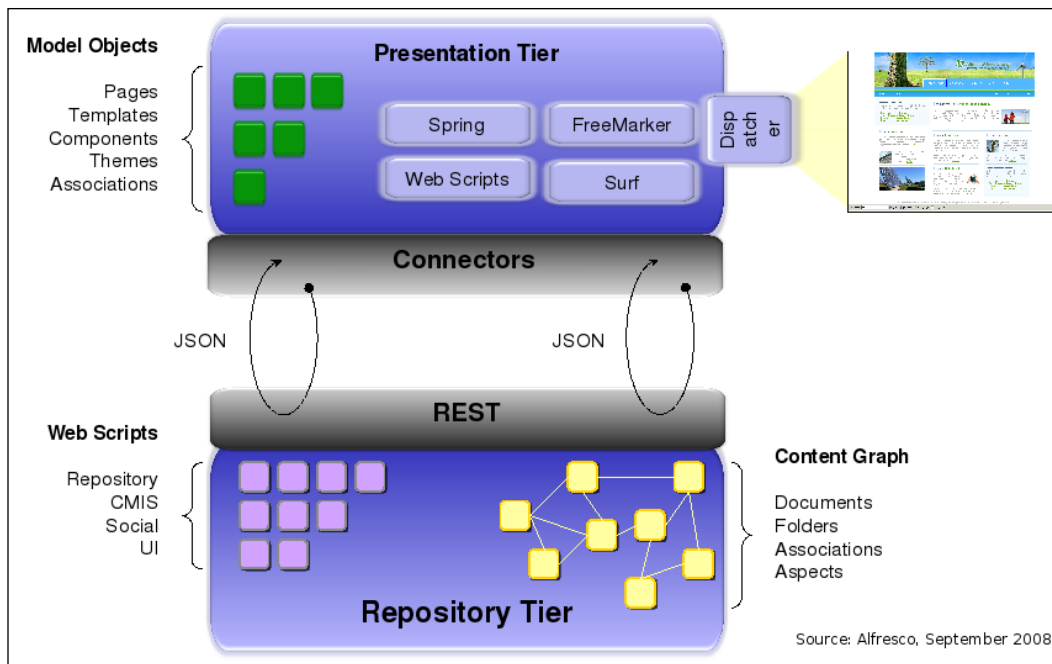
## **Surf: A New Web Framework**

Surf is a web application development framework that can be used to build custom clients on top of Alfresco. Alfresco isn't even a requirement—you can use Surf to integrate data from any backend that understands REST. Surf takes the web script framework (covered in Chapter 6) and blows it out into a full-fledged, presentation-tier, application framework that allows you to assemble content-centric components into pages. If you're thinking "content management mash-up", you are on the right track.


Before Surf, if you wanted to build a custom, content-centric application, you'd either use a portal framework such as Liferay or JBoss Portal, or a general-purpose application development framework such as Symfony (PHP), Django (Python), or Spring (Java). Surf is a framework in the same sense as these general frameworks but it is Alfresco-aware, which means a lot of the plumbing is done for you. On the other end of the spectrum, it isn't quite a portal and that means you don't have the overhead of a portal server to deal with.

One of the main drivers behind the new framework is the goal of moving away from JavaServer Faces (JSF) in Alfresco's clients. It's just too difficult to extend and customize for most customers. Another important goal is to create better separation between the web tier and the backend or repository tier. In the current Enterprise release, the web client and the repository run in the same process. In the Labs release and the 3.0 Enterprise release, this does not have to be the case. Web sites built with Surf can run in a completely separate application server on a completely different host.

The following diagram shows this separation:



As you can see, **Surf** applications sit in the **Presentation Tier**. They use **Web Scripts** to render pages and make calls to end points in the **Repository Tier**. The **Repository Tier** is most likely one or more Alfresco repositories, but it doesn't have to be.

 Surf is not generally available yet. It is currently only available in Labs. As such, some of this terminology and the underlying details are subject to change.

---

## Surf Overview

The new web framework introduces the concepts of **Pages** and **Templates**.

**Templates** are implemented with **FreeMarker**. A template defines a layout and regions within that layout. Those regions are then bound to **Components**. A given page – Home Page, for example, or Products – uses a template to figure out its layout. The home page might be a three-column layout, while the products page might be a two-column layout, for example due to which those two pages would use two different templates.

So **Templates** describe the layout of a page including regions, which are then filled with **Components**. What is a component then? A component is really just a web script. Instead of running in the same process as the Alfresco repository, these web scripts are running on the **Presentation Tier**. **Surf** adds the ability for web scripts to make remote calls to end points to retrieve data. **Surf** provides connectors, which handle the handshake with the end point. Because of this, as a developer, all you have to worry about is which end point you want to call and what URL you want to give it.

When a page is invoked, all of the components for that page get invoked. Once the **Components** on a page are processed, the rendered page is returned to the user. The rendered page might make additional client-side JavaScript calls to **Web Scripts**. These calls are made using the **Yahoo User Interface (YUI)** library. The end point of the call can either be the **Surf** web application running in the web tier, web scripts running on an Alfresco repository, or any other URL.

Other features of **Surf** include:

- Authentication and credential vaulting. Once authenticated, a ticket is cached, which is subsequently used for all remote calls to Alfresco. **Surf** includes a credential vault, which can be used to provide authentication into various backends, even if the user's credentials vary by end point.
- The framework only holds basic user session data, which essentially includes the authenticated user information and data. Everything else is re-retrieved with subsequent requests. This makes the interaction near stateless, but can mean quite a few backend REST calls for each browser request for a given page.
- The framework provides a "proxy" to make direct Alfresco callbacks. This works around the cross-browser scripting limitation, which will certainly be in effect because it is highly likely that the web tier and the Alfresco repository are on different hosts. Plus, you might want to call all kinds of RESTful services running anywhere on the network.
- **FreeMarker Templates** will be the most widely used and powerful component rendering engine, but JSPs are also supported.

For more information on the 3.0 Surf web framework as it develops, see:

[http://wiki.alfresco.com/wiki/Web\\_Framework\\_3.0](http://wiki.alfresco.com/wiki/Web_Framework_3.0).

## Separation of the Repository from Presentation

As mentioned earlier, Alfresco wants to separate the repository process from the web application tier. Picture a "headless" repository, still deployed as a web application, that would be accessed by one or more web applications. The web applications will retrieve data from the repository via REST. This frees up the web client applications from running in the same process as the repository. They won't even have to run on the same physical machine.

One of the new Surf-based client applications Alfresco is delivering with 3.0 is called "Share", which was previously known by its code name, "Slingshot". In Alfresco Share, repository data is fetched and presented in two different ways:

1. **Web Scripts** in the **Presentation Tier** make a call to the Alfresco server to assemble a full page.
2. A user requests a web page, which consists of several components, or calls a webscript running in the **Presentation Tier**.
3. Each component, implemented as a web script, fetches the required parameters from the request and/or the component configuration, then performs remote calls to web scripts running on the backend Alfresco repository to fetch the required data.
4. Just like the web scripts you are familiar with, the controller for the web script running in the **Presentation Tier** places the data it just retrieved from the repository in a model. The view, a **FreeMarker** template, then reads data from the model to produce the rendered component.
5. Once all components have rendered, the assembled page is returned to the client. JavaScript in the browser client makes AJAX calls.
6. A component is loaded or its state is changed.
7. The component fetches new data using an AJAX call either to **Web Scripts** running in the **Presentation Tier** (for example, to load an edit form) or to **Web Scripts** running in the Alfresco repository (to fetch data). The latter will be "proxied" through **Surf**, thus avoiding cross-domain browser limitations.
8. The returned data is processed and rendered on the page.

---

## Site Configuration

Share and the other 3.0 web clients are really just dynamic web sites. In fact, they can be managed using Alfresco's **Web Content Management (WCM)** functionality just like any other web site because things such as page, template, and component definitions are just XML files. XML-based configuration has several benefits:

1. Surf-based web applications can be built using rudimentary tools such as plain text editors and can often be round-tripped without a server restart.
2. Surf-based web applications are completely virtualizable in Alfresco's virtualization server, can be easily sandboxed, and all of the other features that Alfresco WCM provides can be performed on them.
3. The XML structure and some naming conventions make it easy to construct the site, but also lend themselves to automation through tooling. Expect to see Alfresco provide a **Site Builder** type of tool in the near future.

As you can imagine, it doesn't take long for the XML files that define a Surf site to pile up. Luckily, a directory structure convention puts some method to the madness. Let's look at some of the more interesting directories.

### Site-Data

Each web application is configured by the XML files found in the site-data folder. The XML files configure the different web pages and components available in the web application. The most important ones are:

- **Components:** Defines all the components that can be pulled in by a page template. Components can have different scopes (page, template, global), which means you can have some components globally configured for all pages (for example, the page header/footer), template-level configured (for example, the page title for the different dashboard templates), or page-level configured (which would mean that components on a page would look different depending on the template that loaded the component in).
- **Pages:** Defines the different pages available in the web application. Similar to a web script descriptor, each page defines an authentication level (for example, **None**, **User**, **Admin**) and the template instance to use.
- **Template instances:** Defines what template is loaded, and, optionally, properties for the template. Templates can either be FreeMarker templates or JSP pages.

- **Content associations:** Defines the page which should be used for a given content type. For example, when someone links directly to an object, Share can inspect that object's content type and then display the object using the appropriate page/template.
- **Page associations:** Defines the structure or page hierarchy of the web application. This information can be used to drive navigation components.

There are other directories, which are not covered here. Review the Surf documentation on the Alfresco Wiki for more information.

## Site-Webscripts

This folder contains all of the **Web Scripts** that live on the **Presentation Tier** for a site. **Web Scripts** can either be called directly or can be used to assemble a page by binding them to regions in a template. When they are bound to a page, they are called **Components**.

## Templates

This directory contains all FreeMarker templates that act as page templates. The FreeMarker is used to lay out the page, and then define regions into which components will be bound.

For more information on this structure, see: [http://wiki.alfresco.com/wiki/Web\\_Framework\\_3.0\\_-\\_Developers\\_Guide](http://wiki.alfresco.com/wiki/Web_Framework_3.0_-_Developers_Guide).

For information on writing web script-based components, including components that make client-side web script invocations, see [http://wiki.alfresco.com/wiki/3.0\\_Component\\_Standards](http://wiki.alfresco.com/wiki/3.0_Component_Standards).

## Dynamic Deployment

Starting with version 2.9 Community, content models, web client configurations, and message bundles can be placed in the repository and dynamically reloaded. This is a huge step forward for highly available environments because, unlike the deployment options discussed so far, no application server restart is required.

To use dynamic deployment, the files have to reside in specific folders within the Data Dictionary. Once there, they are activated using the appropriate console. The following table provides the specifics:

Customization	Folder in which it resides within the Data Dictionary	Reloaded using this console
Custom content models	Models	Repository Admin Console
Web client configuration	Web Client Extension (the file <i>must</i> be named <code>web-client-config-custom.xml</code> )	Web Client Config Console
Web client properties	Web Client Extension (the file <i>must</i> be named <code>webclient.properties</code> )	Reloaded automatically on next user login
Message bundles	Messages	Repository Admin Console

None of the current versions of Alfresco have links to these consoles as part of the interface. You have to know they are there. The URL for the Repository Admin Console is:

`http://localhost:8080/alfresco/faces/jsp/admin/repoadmin-console.jsp`

The URL for the Web Client Config Console is:

`http://localhost:8080/alfresco/faces/jsp/admin/webclientconfig-console.jsp`

In both cases use the **Help** command to find the reload syntax specific to the file that needs to be reloaded.

## Summary

This appendix has provided you with several references or other details that didn't fit neatly into one of the book's chapters. As always, the Alfresco source and Javadoc are the ultimate references for the kind of information presented here. The Alfresco Wiki and the Alfresco Forums are also valuable resources. The hope is that this appendix saved you some digging.

