Building an Alfresco Custom User Interface - DoCASU

www.DoCASU.org

Optaros White Paper
October 2008

Assemble the Future

Building an Alfresco Custom User Interface - DoCASU

This white paper illustrates how a complete web application - in this case, an Alfresco custom user interface - can be built using Alfresco Web Scripts and Rich Internet Application frameworks. Our focus is DoCASU, a real-world application Optaros built and made freely available as an open source project on www.docasu.org. The paper discusses how Optaros designed and built the user interface, reviews some of the major architectural and design drivers, explains some of the main parts of the DoCASU codebase, and finally describes how to download and install the application on top of your own Alfresco repository.

Introducing DoCASU

The Alfresco standard user interface is aimed at users with a broad spectrum of expertise and needs, from standard users to power users and administrators. It was designed as a general purpose document management tool that provides all the capabilities of the underlying Alfresco repository to the end user. However, many of the customers Optaros talks to have expressed the need for a simpler and a much more targeted user interface. Their knowledge workers must interact with the repository on a day-to-day basis to perform a basic set of functions: access, exchange, and contribute content to the document repository. The out-of-the-box web client provides too many advanced features that they rarely use.

The goal of DoCASU, which stands for Document Access for Casual Users, is not to replace Alfresco's user interface, but to provide the Alfresco community with a custom user interface which:

- Has a strong focus on user experience (e.g. ease of use, responsiveness)
- Targets average end-users rather than the small percentage of power users
- Fosters a broader acceptance of the Alfresco-based solution by a larger group of users
- Is conducive to a large scale deployment
- Proposes a framework/model for building similar content-centric applications

The screenshot below shows the main DoCASU page with folders on the left, a selectable, right-clickable list of files in the center, and a preview pane on the right:

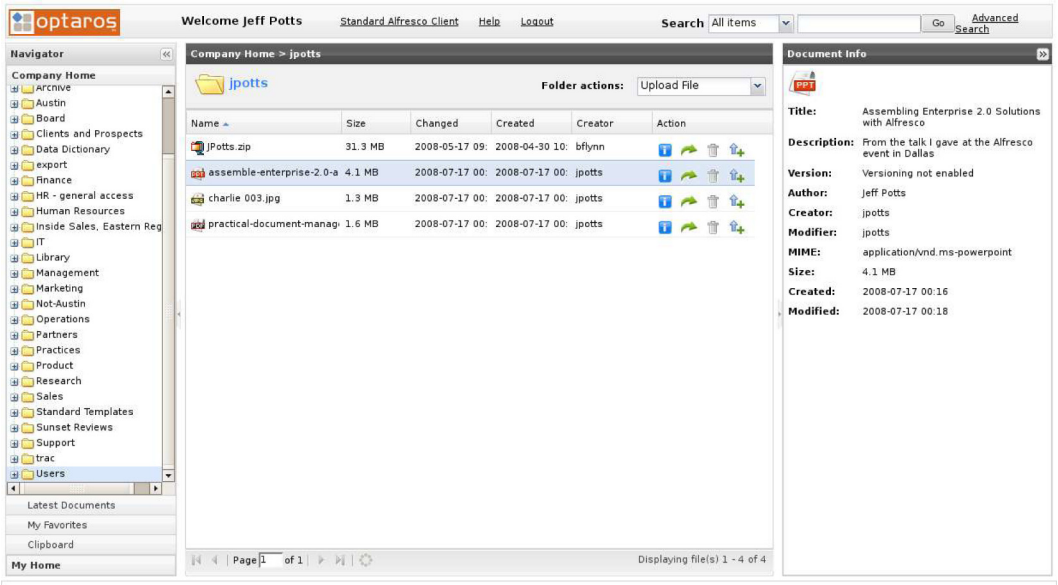

Figure 1: User interface

Clicking the information icon for a given document launches a tabbed dialog with metadata, versioning information, and buttons for common tasks like copying the download link to the clipboard:

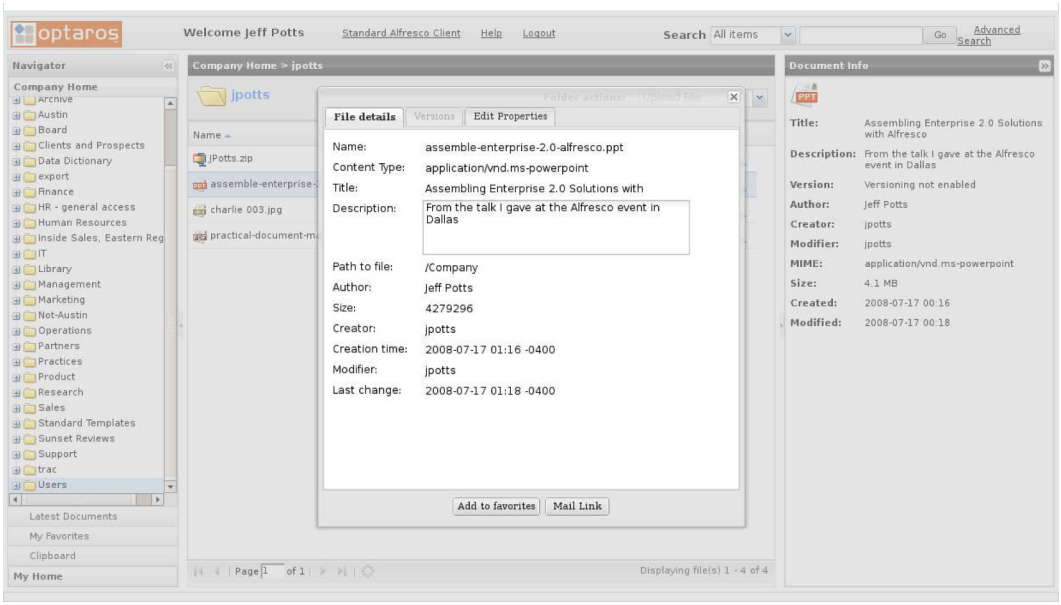

Figure 2: Property sheet

Designing the User Interface

Building a new user interface starts by putting the user in the center of the picture. This is the cornerstone of Optaros' User-Centric Design methodology, but you would be surprised how many people don't do this. Let's go through the steps Optaros followed to design the DoCASU user experience, which include:

1. Identifying key users
2. Categorizing key users into personas
3. Creating scenarios for each persona
4. Iteratively creating rough wireframes
5. Optionally, transforming the wireframes into a high-fidelity prototype

Identifying Key Users

The first step is to identify the key users and determining the tasks they perform (or try to perform) in their daily work. This can be done through interviews, surveys, observation, and market studies.

Categorizing Users into Personas

The next step is to categorize the different user groups in User Personas (Kimberley, Knowledge Worker, New Employee), with each persona representing a class of user with specific usability requirements and a list of tasks they need to achieve.

Creating Scenarios

Each task corresponds to a scenario that describes the work to be performed. That scenario must be supported by the system if it is to be successful. This process is also valuable in determining the pain points and in exposing potential interaction problems with the current system.

Here is a sample of a user task scenario for the persona, “Kimberley - Knowledge Worker, New Employee”:

1. Kimberley’s boss asks her to modify the “Annual Report” executive summary paragraph. Kimberley is new to the company and does not know where the document is.
2. Kimberley logs in to the system using her corporate credentials.
3. Kimberley searches by keyword for “Annual Report 2008” and finds the document in “Company_Home\Management\Annual Reports\2008”.
4. Kimberley thinks, “This probably won’t be the last time I’m asked to update this document,” so she adds the folder to her favorites for next time.
5. She checks out the document.
6. She modifies the working copy using Microsoft Word.
7. When she is finished making the changes, she saves the document and checks it back in.
8. The system increments the document version number.
9. Kimberley informs her boss that the document is updated and includes a direct link to the updated document in an email.

Iterating over Wireframes

Once the personas are identified, wireframes can be created. Wireframes serve as a visual way to illustrate how the system will behave. Wireframes are created iteratively so that users can provide feedback and designers can incorporate that feedback. The process repeats until the users agree that the user task scenario can be accomplished.

Here is an example of an initial wireframe – using greyscale allows user to focus on functionality, information architecture, and user experience rather than on colors.

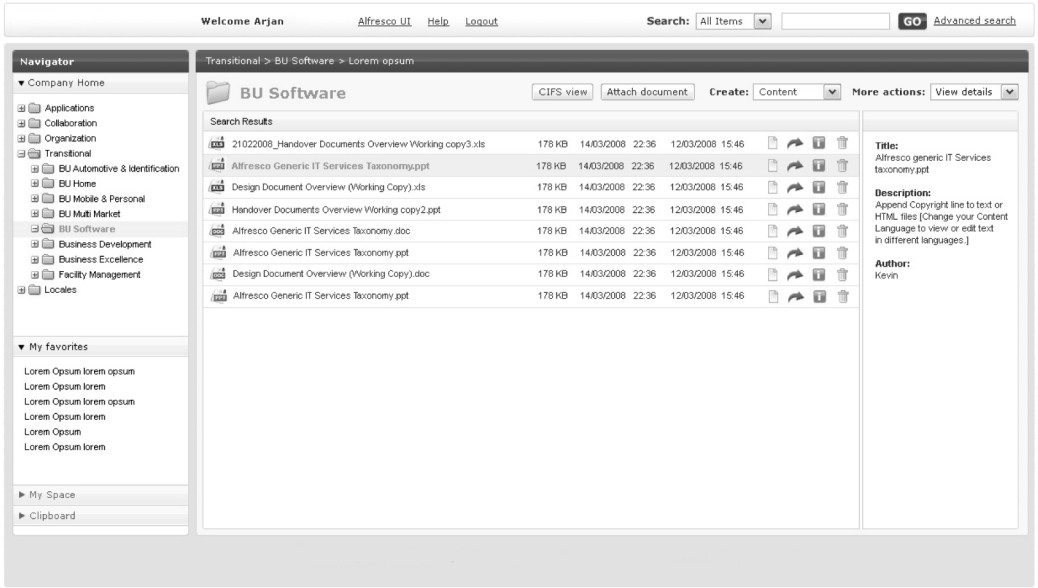

Figure 3: Wireframe

Creating a Prototype

When the wireframes are good enough, they can then be refined into a high fidelity prototype including final look-and-feel, color scheme, and branding information. This step isn’t always required. For DoCASU the wireframes were good enough and the development fast enough that heavy prototyping was not needed.

DoCASU Architecture

Let's go through some of the thoughts behind why Optaros built DoCASU the way they did and how the high-level architecture is set up.

High-level Approach

Alfresco provides several integration/extension points that can be leveraged to create a custom user interface. The main options Optaros considered while designing DoCASU included the following:

- Use Alfresco's default web-client extension mechanism and create some additional pages as part of the web-client.
- Implement a different web-application that will interact with Alfresco using one of Alfresco's APIs (Java, SOAP, JCR, REST).

Ultimately, Optaros decided to not extend the existing web client and instead build a new web application built on web scripts and the Ext JS JavaScript library (http://extjs.com). The decision was driven by the following factors:

- The web script framework provides a RESTful API that offers a huge advantage in terms of flexibility and implementation speed while maintaining full access to the Alfresco Foundation API.
- This approach makes it possible to build a page-centric, Rich Internet Application leveraging state of the art Ajax libraries, without being locked in to a presentation approach.
- This approach addresses some of the limitations of the JSF-based, out-of-the-box web-client (e.g., tab navigation, browser history, etc.).
- Implementing a REST API frees up the front-end, which opens the door to alternative user interfaces (e.g., Rich Desktop Application, Mobile Application, Office Plug-ins, etc.).
- The approach is consistent with the Alfresco roadmap and should offer a safe migration path to Alfresco 3.0 which will include multiple new web clients, all built on web scripts and the Remote Web Script Runtime.

High-Level Architecture

The following diagram provides a high-level view of the DoCASU architecture as well as interactions between the web front-end and the Alfresco repository:

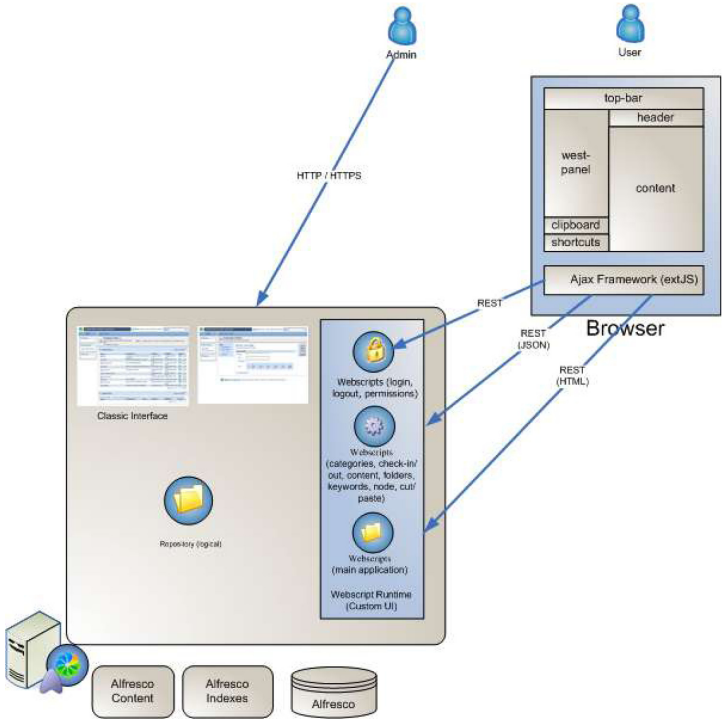

Figure 4: Architecture

As you can see, web scripts are being used in several different, but related, ways:

- Web scripts handle login, logout, and other security services.
- Web scripts interact with the content repository to provide a RESTful API for core repository services such as categorization, check-in/check-out, content and folder creation, cut-and-paste, and more.
- Web scripts serve the main page of the application.

Designing the RESTful API

The RESTful API represents a clean separation between the repository and the front-end components so it is important to spend some time designing that interface. Although it will likely be available in the 3.0 release and through the new CMIS standard, the web script framework is just that—a framework. There isn't much "API" to it. For DoCASU, the REST API that exposed the core services had to be created from scratch. The good news is that because it serves up mostly data, this interface isn't tightly-coupled to DoCASU—it can also be leveraged by any other application with similar functional coverage. For example, maybe you want to create something similar to DoCASU that runs on Adobe AIR or a full-featured web client for the Apple iPhone or a set of Eclipse plug-ins or... well, you get the picture.

Determining the URL, HTTP Method, and Response Format

The key elements to consider when designing the REST API are the URL of the script, the HTTP method used to make the request, and the output format. Each resource that needs to be manipulated or accessed should have a unique URL. For example, here are some of the web script URLs that DoCASU leverages:

/docasu/ui/node/{nodeId} or /docasu/ui/node?nodeId={nodeId} addresses a node
/docasu/ui/node/content/{nodeId} addresses the content of a node
/docasu/ui/node/properties/{nodeId} addresses the properties of a node

DoCASU's RESTful API uses the following syntax to decide which HTTP method to use:

- **GET:** Retrieves the representation of a resource. The URL should return the same resource no matter how many times it is called. Calling this URL should have no effect on the repository. A GET usually doesn't require a transaction.
- **POST:** Creates a new resource. The server decides on that resource's URL.
- **PUT:** Creates a new resource but the client decides on the URL, or updates an existing resource. Like a GET, the URL should update the same resource no matter how many times it is called.
- **DELETE:** Deletes a resource, or stops a resource from being accessible.

The following output formats are used in the API:

- JSON is used for better separation between repository structure and presentation. Plus, it's fast to parse. JSON is the default output of this API.
- HTML is used for the homepage of the application that is served via a Web Script. It is also used for services that require uploading attachments (multi-part forms).
- XML is not currently used but could be for particular use cases such as when it is needed to be compliant to XML-based standards (e.g., RSS, OpenSearch, etc.).

The following tables describe, for each Web Script, the URL, the HTTP method, and the response output format. The tables are divided into rough functional areas

Manage Nodes API

This API manages the repository nodes/documents.

URL	Method	Description	Response Format
/docasu/ui/node/properties/{nodeId}	GET	Get the properties/ details for a node based on the node id	JSON
/docasu/ui/node/properties/{nodeId}	POST*	Update the properties/details for a node based on the node id	HTML
/docasu/ui/node/permissions/{nodeId}	GET	Get permissions of a given node	JSON
/docasu/ui/node/versions/{nodeId}	GET	List version of a given document	JSON
/docasu/ui/node/{nodeId}	DELETE	Remove a given node	JSON
/docasu/ui/node/name/{nodeId}? newName={newName}	PUT	Rename a node	JSON
/docasu/ui/node/create/{folderId}	POST	Create a document in a given folder	HTML
/docasu/ui/node/update/{folderId}	POST	Update the version of a node	HTML
/docasu/ui/node/recentdocs	GET	List the documents recently modified	JSON

Manage Folders API

This API manages repository folders.

URL	Method	Description	Response Format
/docasu/ui/folders?node={nodeId}	GET	List subfolders of a given folder	JSON
/docasu/ui/folder/{folderID}?folderName={folderName}	POST	Create a folder with a given name	JSON
/docasu/ui/folder/name/{folderId}?newName={newName}	PUT	Rename a folder	JSON
/docasu/ui/folder/properties/{folderId}	GET	Get the properties/ details for a node based on the node id	JSON
/docasu/ui/folder/paste/{folderId}?c={commaSeparatedItemIds}	POST	Paste all items from the clipboard to the given folder	JSON
/docasu/ui/folder/permissions/{folderId}	GET	Get permissions of a given folder	JSON
/docasu/ui/folder/docs?nodeId={folderId?}&start={rowsFrom?}&limit={rowsTo?}&sort={orderBy?}&dir={orderDirection?}	GET	Get the files in a given folder. If no folder is specified, returns the Company Home file list.	JSON

Manage Folders API

This API manages content.

URL	Method	Description	Response Format
/docasu/ui/node/content/{nodeId}	GET	Get content of a given node	JSON
/docasu/ui/node/content/update/{nodeId}	POST	Update file content	HTML
/docasu/ui/node/content/upload/{folderId}	POST	Create a file in a given folder	HTML

Check-in/Check-out API

This API manages check-in and check-out of documents.

URL	Method	Description	Response Format
/docasu/ui/node/checkout/{nodeId}	PUT	Check out a document	JSON
/docasu/ui/node/checkout/{nodeId}	DELETE	Undo check in a document	JSON
/docasu/ui/node/checkin/{nodeId}	PUT	Check in a document	JSON

Search API

This API manages search.

URL	Method	Description	Response Format
/docasu/ui/search?q={searchText} &searchType={searchType?} &nodeId={lookIn?} &createdFrom={createdFrom?} &createdTo={createdTo?} &modifiedFrom={modifiedFrom?} &modifiedTo={modifiedTo?} &start={rowsFrom?} &limit={rowsTo?} &sort={orderBy?} &dir={orderDirection?}	GET	Perform a full text search	JSON

Categories API

This API manages categories.

URL	Method	Description	Response Format
/docasu/ui/categories?node={parentCategoryId}	GET	Get sub-categories of a given category	JSON

Shortcuts API

This API manages user shortcuts.

URL	Method	Description	Response Format
/docasu/ui/shortcuts	GET	Get shortcuts for the current user	JSON
/docasu/ui/shortcut/{nodeId}	DELETE	Remove a shortcut	JSON
/docasu/ui/shortcut/{nodeId}	PUT	Add a shortcut for the given nodeId	JSON

User Interface API

This API is for utility or security functionality.

URL	Method	Description	Response Format
/docasu/ui	GET	Load the custom UI main page	HTML
/docasu/ui/user	GET	Get the currently logged in user	JSON
/docasu/ui/logout	POST	Logout the user	JSON

Additional Design Considerations

With the API planned, there were only a few loose ends to be tied up before coding could begin: transactions, controller language, and authentication.

Handling Transactions

Web scripts can declare their transaction needs. The framework doesn't care whether you are doing a GET or a POST with regard to transactions. If the web script needs one, it can have one.

For DoCASU, all scripts that make modifications to the repository are declared as transactional. In DoCASU, services using a GET method won't require a transaction as they are not modifying the state of the repository.

Choosing a Controller Language

With multiple developers working on the REST implementation, it made sense to choose a default controller language and specify guidelines for when an alternative is acceptable.

In DoCASU, most of the controllers are implemented using JavaScript. JavaScript controllers provide access to most of the repository API, are very easy and straightforward to implement, debug and maintain, and provide an acceptable performance level in most cases.

Not every controller can do what it needs to with the JavaScript API, though. In those cases, Java was used. For example:

- Shortcut management
- Search
- Controllers that need access to server configuration information
- The logout script that invalidates the HTTP session

Dealing with authentication and sessions

As mentioned in the introduction, DoCASU is not a replacement for the out-of-the-box web client. By design, it implements a small subset of the standard web client features. Therefore, it was important for DoCASU users to be able to painlessly switch back-and-forth between the standard web client and the DoCASU web client.

In order to be able to switch between clients without having to login multiple times, the two clients need to share the same security context or HTTP session. This is easily done by having the start page of the DoCASU application served by a web script that leverages the JSF/JSP web script run-time, and requiring the user to be authenticated in the descriptor (<authentication>user</authentication>). Doing this causes the standard Alfresco login page to be used for user authentication and allows the HTTP session to be shared between the two clients.

One of the annoying limitations of Alfresco's user interface (inherited from JSF) is that it does not support tabbed browsing. The RESTful model, by keeping the state on the browser side, not only provides better scalability due to its stateless model, but also allows several applications to be running in different browser tabs independently.

Diving into the code

You've seen how DoCASU was designed, both from a user-centric perspective and from a technical perspective. Now it is time to get in to some of the details.

Main Components

The diagram below shows the main components used in DoCASU:

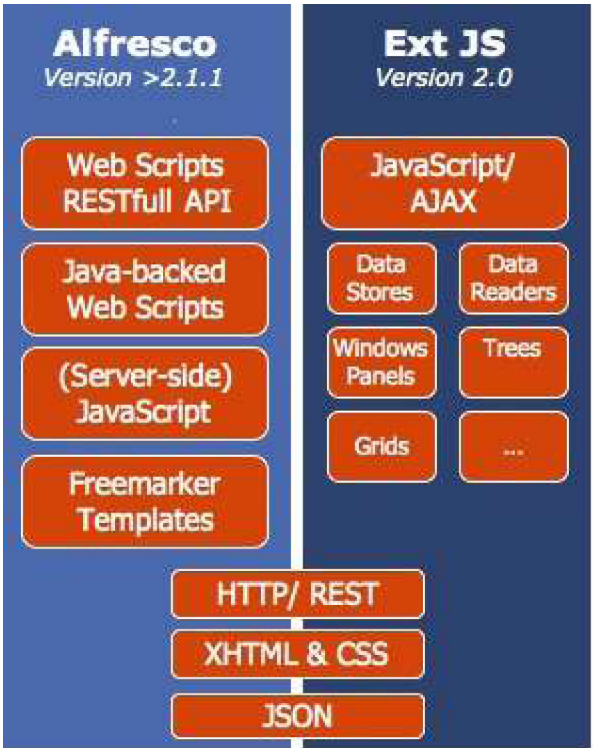

Figure 5: Components

The left side of the diagram is Alfresco's domain. It consists of the Alfresco Web Scripts engine, Java and JavaScript controllers, and the FreeMarker templating engine.

The right side of the diagram is the user interface or presentation layer. DoCASU's presentation layer was built with the Ext JS JavaScript library. Ext JS is a JavaScript library well-suited for building Rich Internet Applications. It includes:

- An extensive Rich User Interface widget library
- Out-of-the-box AJAX and REST support
- An extensible component model
- Cross browser support
- An intuitive and easy to use API

This library is dual licensed with commercial and GPLv3 licenses available. For more information refer to http://extjs.com/deploy/dev/docs/.

Structuring the Source Code

The DoCASU application is packaged as an AMP (Alfresco Module Package). An AMP file is composed of code, XML, images and CSS that extend the standard Alfresco functionality. You can learn more about how to package and deploy an AMP on the Alfresco Wiki at http://wiki.alfresco.com/wiki/AMP_Files or in the Alfresco Developer Guide by Jeff Potts (Packt Publishing, 2008).

The code base is composed of the following main folders/packages including both the application code and build/packaging information.

- module: Contains the AMP module configuration (module.properties, file-mapping.properties).
- module/docasu: Contains the custom user interface code.
- repository/2/java/com/optaros/alfresco/docasu: Contains the Java controller code for Alfresco 2.x – due to some non-backward compatible changes in 3.x the Alfresco 3.x code is in repository/2/java/com/optaros/alfresco/docasu
- src/main/extension: Contains the configuration files for registering Java controllers.
- src/main/webscripts: contains the application Web Scripts. Web Scripts are grouped by packages corresponding to different pieces of functionality and the resources they are managing (e.g., node, folder, shortcut, check-in, clipboard, search, etc.) /html.status.ftl and /json.status.ftl are templates used to wrap/format the error messages.

Building the AJAX-based File Browser

As previously mentioned, the main page of the DoCASU application is served by a web script as HTML. This web script has user authentication configured to enforce the user to login using Alfresco's standard login process. The main page is responsible for loading JavaScript that builds the different graphical components using the Ext JS component library and custom DoCASU libraries.

Laying out the Main Page

The four areas of the main page are defined as illustrated in the screenshot below:

- Header section on the top: Generic navigation and search
- Space/folder navigator on the left: Direct access to user's Favorites, Clipboard, and Latest Documents
- Main file grid in the center: Selectable rows and a context menu (right mouse click)
- Document details panel on the right: Image preview and node/ document metadata

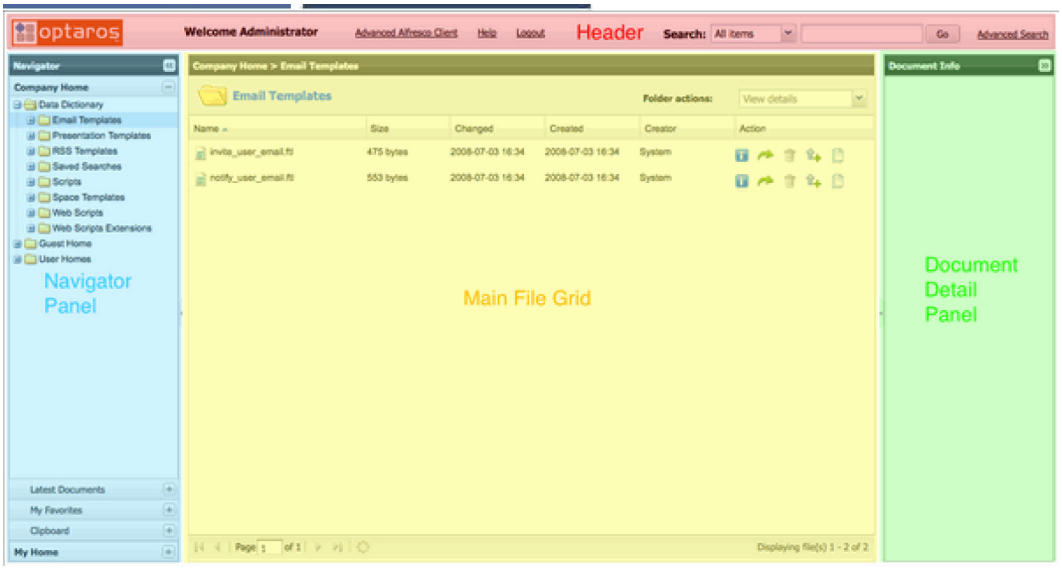

Figure 6: Layout

Let's look at the code that creates the main application view and the different regions of the page (header, navigator, center, east, footer). Each of the regions on the main page is a panel that uses the Ext JS library to display its part of the application. This code lays out the regions:

```
      var viewport = new Ext.Viewport({
   layout: 'border',
   items: [
               header,
               navigator,
               center,
               east,
               footer
          ]
    });
```

Now let's see how a specific region of the application—the center—is rendered. The following code creates the center panel and renders it in the center region of the application viewport. The center panel is split into two sub-panels. One contains the information on the currently selected folder ("centerHeader") and the other contains the file list ("centerView").

```
var center = new Ext.Panel({
           id: 'centerPanel',
           layout:'border',
           region:'center',
           margins: '0 0 0 0',
           header: true,
           items: [centerHeader, gridList]
      });
```

Populating Panels with Data

The above code gives you an idea of how the different regions or panels are defined, but how do the panels get populated with data? Take the list of files in a folder as an example. Populating the list involes a proxy, a reader, a data store, and a data grid. Let's look at each of these.

Proxy

First, there is the HTTP proxy that uses an AJAX request to the “ui/folder/docs” web script. As you saw in the API definition, the web script returns the file list of a given folder or of Company Home, if no folder is specified..

```
var docsProxy = new Ext.data.HttpProxy({
                url: ' ui/folder/docs',
                method: 'GET'
           })
```

JSON Reader

The second component that needs to be configured is a JSON Reader that parses the response and loads the document meta-data required to display or interact with the document.

```
var docsReader = new Ext.data.JsonReader({
            root: 'rows',
            totalProperty: 'total',
            id: 'nodeId',
                   fields: [
                      {name: 'created', type:'string'},
                      {name: 'author', type:'string'},
                      {name: 'creator', type:'string'},
                      {name: 'description', mapping:'description'},
                      {name: 'filePath', type:'string'},
                      {name: 'mimetype', type:'string'},
                      {name: 'url', type:'string'},
                      {name: 'downloadUrl', type:'string'},
                      {name: 'modified', type:'string'},
                      {name: 'modifier', type:'string'},
                      {name: 'name', type:'string'},
                      {name: 'nodeId', type:'string'},
                      {name: 'link', type: 'string'},
                      {name: 'size', type:'int'},
                      {name: 'title', type:'string'},
                      {name: 'version', type:'string'},
                      {name: 'versionable', type:'boolean'},
                      {name: 'writePermission', type:'boolean'},
                      {name: 'createPermission', type:'boolean'},
                      {name: 'deletePermission', type:'boolean'},
                      {name: 'locked', type:'boolean'},
                      {name: 'editable', type:'boolean'},
                      {name: 'isWorkingCopy', type:'boolean'},
                      {name: 'iconUrl', type:'string'},
                      {name: 'icon32Url', type:'string'},
                      {name: 'icon64Url', type:'string'},
                      {name: 'isFolder', type:'boolean'},
            ]
})
```

Data Store

The data store relies on the two previously configured elements: the HTTP proxy and the JSON reader. When the data store is triggered, it executes an AJAX request against the REST API and parses the JSON that gets returned:

```
var gridStore = new Ext.data.Store({
     proxy: docsProxy,
     reader: docsReader
});
```

Grid Panel

Once the data is loaded into the gridStore it can be shown in an Ext JS GridPanel. The GridPanel displays the required columns and handles the selection of an element in the list. It also handles paging and sorting on columns. (Not to sound like a lament for the good old days of web development circa 1997, but do you remember when you had to handle all of this on your own?)

```
fileSelectionModel = new Ext.grid.RowSelectionModel({singleSelect:true});
var gridList = new Ext.grid.GridPanel({
        id: 'fileGrid',
           region: 'center',
           store: gridStore,
           border: false,
           columns: [
              {id:'nodeId', header: "Name", width: 110, sortable: true, dataIndex:
'name', renderer: fileNameRenderer},
              {header: "Size", width: 20, sortable: true, dataIndex: 'size',
renderer: Ext.util.Format.fileSize},
              {header: "Changed", width: 60, sortable: true, dataIndex: 'modified',
renderer: timeZoneAwareRenderer},
              {header: "Created", width: 60, sortable: true, dataIndex: 'created',
renderer: timeZoneAwareRenderer},
                 {header: "Creator", width: 60, sortable: true, dataIndex:
'creator'}
              ,{header: "Action",  sortable: false, dataIndex: 'nodeId', renderer:
actionRenderer}
          ],
          viewConfig: {
              forceFit: true
          },
           bbar: new Ext.PagingToolbar({
            pageSize: 50,
            store: gridStore,
            displayInfo: true,
            displayMsg: 'Displaying file(s) {0} - {1} of {2}',
            emptyMsg: "No files to display"
      }),
      sm: fileSelectionModel,
});
```

Formatting Dates and Numbers

Some specific renderers are registered to improve the data being displayed such as creation date, file name, and file size. These renderers are JavaScript methods that return a text representation of a cell based on the current context (the current record, column, and value). For example, here is how to display all dates using the browser time zone, using Ext JS date utilities.

```
function timeZoneAwareRenderer(value, column, record) {
      var dateValue = Date.parseDate(value, "Y-m-d H:i O");
      var formattedDate = Ext.util.Format.date(dateValue, "Y-m-d H:i");
      return formattedDate;
}
```

Adding File Download and Folder Links

Renderers are also used to add the icon corresponding to the file or folder, along with some additional behaviour when clicking on the file name (downloading the file), or a sub-folder (changing the current folder selection):

```
function fileNameRenderer(value, column, record) {
      var html = '';
      if (record.get('isFolder') == true) {
html += '<a href="#" onClick="selectFolder(\''+record.get('nodeId')+'\')">';
      } else {
            html += '<a href="'+record.get('downloadUrl')+'">';
      }
      html += '<div style="float:left">';
      if (record.get('isFolder') == true) {
html += '<img src="../../docasu/lib/extjs/resources/images/default/tree/folder.
gif"';
      } else {
            html += '<img src="'+record.get('iconUrl')+'"';
      }
      html += ' alt="'+record.get('name')+'"';
      html += '</div>';
      html += '<span> '+record.get('name')+'</span>';
      html += '</a>';
      return html;
}

Triggering the Data Load
The last bit of code required is actually triggering the data load and setting the
default sort order based on filename:
      gridStore.setDefaultSort('name', 'asc');
      gridStore.load();
```

Handling the Data Request on the Server

When the data load is triggered, it invokes a web script. The JavaScript controller gets input parameters, fetches the nodes in the repository using the Alfresco repository API, and sets the entries to the web script's model before forwarding on to the FreeMarker view.

Here is a simplified version of the controller excluding exception management, handling of special node types, and paging.

Moreover, the real web script has been re-written in Java.

```
var nodeId = companyhome.id;
if (args.nodeId != null && args.nodeId.length>0) {
      nodeId = args.nodeId;
}
var nodeRef = "workspace://SpacesStore/" + nodeId;

var folder = search.findNode(nodeRef);
var total = 0;
var entries = new Array();
for each (child in folder.children) {
var fileName = child.name;
      var canEdit = isInlineEditable(child);
      entries[arrayIndex] = new Array(2);
      entries[arrayIndex][0] = child;
      entries[arrayIndex][1] = canEdit;
total++;
}

model.entries = entries;
model.total = total;
model.randomNumber = Math.random();
model.path = folder.displayPath + "/" + folder.name;
model.folderName = folder.name;
```

The Freemarker template renders the JSON response using the following template:

```
{"total":${total},"path":"${path}","folderName":"${folderName}","rows" : [
<#list entries as child>
{
"nodeId"         :"${child[0].id}",
"name"           :"${child[0].name}",
"title"          :"<#if child[0].properties.title?exists>${child[0].properties.
title}<#else></#if>",
"modified"       :"${child[0].properties.modified?string("yyyy-MM-dd HH:mm Z")}",
"created"        :"${child[0].properties.created?string("yyyy-MM-dd HH:mm Z")}",
"author"         :"<#if child[0].properties.author?exists>${child[0].properties.
author}<#else></#if>",
"size"                 : ${child[0].size?c},
"link"         : "${url.context}${child[0].url}?${randomNumber?string}",
"creator"        : "${child[0].properties.creator}",
"description"    : "${(child[0].properties.description!)?js_string}",
"filePath"       : "${path}/${child[0].name}",
"modifier"       : "${child[0].properties.modifier}",
"mimetype"       : "<#if child[0].mimetype?exists>${child[0].
mimetype}<#else></#if>",
"downloadUrl"    : "${url.context}${child[0].downloadUrl}",
"versionable"    : "${(hasAspect(child[0], "cm:versionable") == 1)?string}",
<#if hasAspect(child[0], "cm:versionable") == 1>
```

```
"version"       : "<#if child[0].properties.versionLabel?exists>${child[0].
properties.versionLabel}<#else>1.0</#if>",
<#else>
"version"       : "Versioning not enabled",
</#if>
"writePermission" : "${(hasPermission(child[0], "Write") == 1)?string}",
"createPermission": "${(hasPermission(child[0], "CreateChildren") == 1)?string}",
"deletePermission": "${(hasPermission(child[0], "Delete") == 1)?string}",
"locked"        : "${(child[0].isLocked)?string}",
"editable"      : "${(child[1])?string}",
"isWorkingCopy": "${(child[0].hasAspect("cm:workingcopy"))?string}",
"iconUrl"       : "${url.context}${child[0].icon16}",
"icon32Url"     : "${url.context}${child[0].icon32}",
"icon64Url"     : "${url.context}${child[0].icon64}",
"isFolder"     : "${child[0].isContainer?string}"
}
<#if child_has_next>,</#if>
</#list>
]
}
```

Here's an example of the JSON the template produces—this is what gets parsed by the Ext JS JSON Reader:

```
{"total":1,"path":"/Company Home","folderName":"Company Home",
  "rows" : [ {
"nodeId"        :"720817dd-4b76-11dd-a598-c76bc4a66276",
 "name"       :"Guest Home",
      "title"      :"Guest Home",
      "modified" :"2008-07-06 18:13 +0200",
      "created" :"2008-07-06 18:13 +0200",
      "author"  :"",
      "size"          : 0,
      "link"     : "720817dd-4b76-11dd-a598-c76bc4a66276?0.828",
      "creator"       : "System",
      "description"   : "The guest root space",
      "filePath"      : "/Company Home/Guest Home",
      "modifier"      : "System",
      "mimetype"          : "",
      "downloadUrl"   : "/alfresco",
      "versionable"   : "false",
      "version"       : "Versioning not enabled",
      "writePermission" : "true",
      "createPermission": "true",
      "deletePermission": "true",
      "locked"        : "false",
      "editable"          : "false",
      "isWorkingCopy": "false",
      "iconUrl"       : "space-icon-default-16.gif",
       "icon32Url"    : "space-icon-default.gif",
       "icon64Url"    : "space-icon-default-64.png",
       "isFolder"   : "true"
```

```
        }
    ]
}
```

Handling Sub-folder Clicks

Back on the user interface, clicking on a sub-folder in the list calls, an event listener that is configured on the folder name gets called. The event listener is responsible for reloading the grid store for the newly selected folder. It might also refresh other panels of the application that are dependent on the selected folder.

```
function selectFolder(nodeId) {
      gridStore.baseParams.nodeId = nodeId;
      gridStore.load();
}
```

Here is the resulting screen:

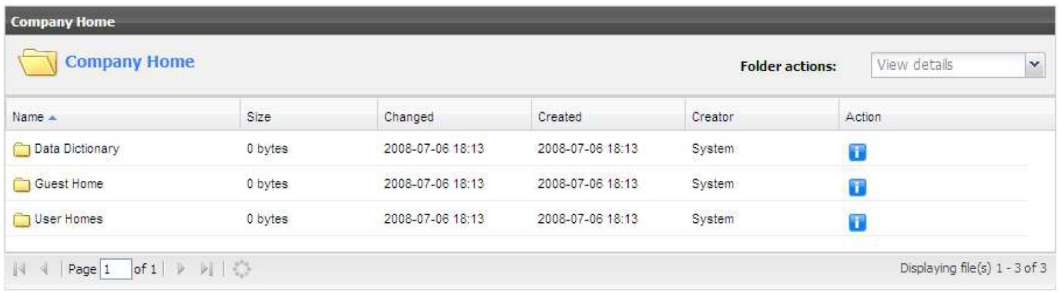

Figure 7: Folder view

Additional actions on files and folders are available either in the action list column (shown in the screenshot above as the column with the blue information icon), or by using the contextual (right-click) menu. You'll see how that works next.

Adding Contextual Menus for Files and Folders

The DoCASU client features a right-click menu for invoking actions on documents in a list. Let's deconstruct the checkout action to see how this works. The pattern illustrated by this example has been used for all contextual interactions with files and folders in the user interface that make calls to the RESTful API for core content services such as delete, check-in, and versioning.

Ext JS provides an effective way to create contextual menus called ExtJS Menu. Here's a code snippet that shows how the menu that gets launched when you right-click a row in the grid list is defined:

```
gridList.on('rowcontextmenu', function (grid, rowIndex, e) {
  e.preventDefault();
  var record = grid.getStore().getAt(rowIndex);

  if (!record.get('isFolder')) {
      this.contextMenu = new Ext.menu.Menu({
          id: 'gridCtxMenu',
          items: createActionItems(record),
          listeners: {
```

```
                click: function(menu, menuItem, e){
                      menu.hide();
                }
          }
     });
  }
```

The context menu gets populated by different actions you can perform on files depending on the state of the node (or "record") and the permissions set on the node. For example, the following code populates the menu with the check-out button under the following conditions:

- the file is not locked
- the node is not already the working copy
- the user has write permission on the file

```
function createActionItems(record) {
      var result = new Array();
      if (!record.get('locked')) {
            if (record.get('writePermission')) {
                  if (!record.get('isWorkingCopy')) {
                        result.push({
                              text: 'Checkout',
                              icon: '../../docasu/images/checkout.gif',
                              handler: function()    {
                                    checkoutFile(record.get('nodeId'))}});
```

The menu opens when the user does a right-click and replaces the system context menu.

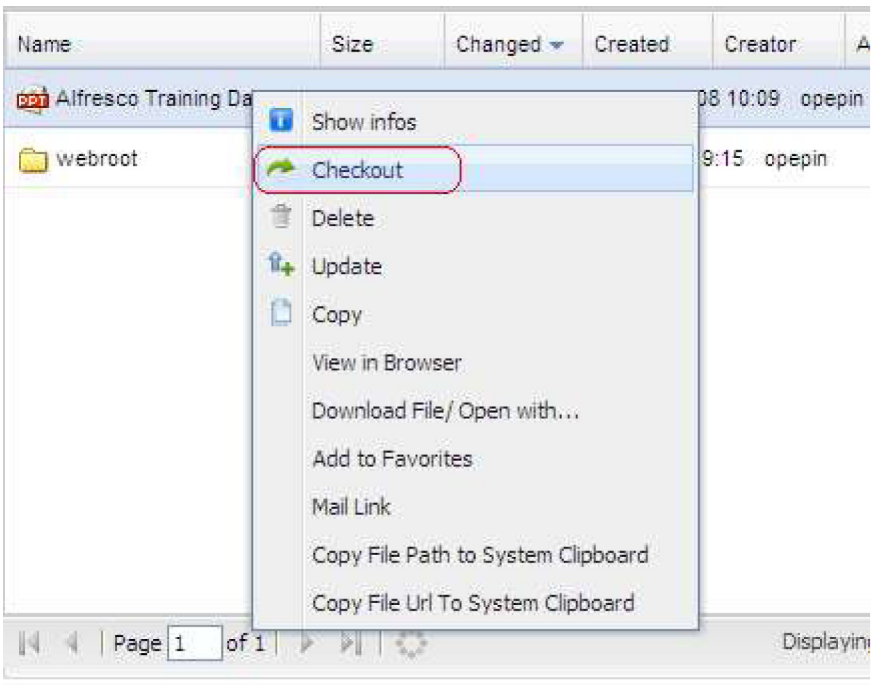

Figure 8: Checkout contextual menu

In the previous code snippet, you can see that the menu-item registers the checkoutFile(nodeId) event handler and passes the current record's node-id as a parameter. This method handles the interaction with the RESTful API and creates an AJAX request using the resource URL, method, and parameters.

As a reminder, the DoCASU API to check-out a document is a PUT to the following URL:
/docasu/ui/node/checkout/{nodeId}

The web script returns JSON. From a user interface perspective, if the checkout is successful, the file is checked-out and the working copy appears in the file list, otherwise an error alert box gets displayed. This is implemented with the following JavaScript code:

```
function checkoutFile(nodeId) {
      Ext.Ajax.request({
            url: 'ui/node/checkout/'+nodeId,
            method: 'PUT',
            success: function(result, request){
                  // reload the folder
          gridStore.load();
            },
            failure: function(result, request){
                  Ext.MessageBox.alert('Failed', 'Failed on checkout file. ' +
result.responseText);
            }
      });
}
```

The check-in and undo-checkout are fairly similar. The same mechanism also applies to other contextual actions on files and folders that need to interact with the Alfresco repository.

Building the UI to Handle File Uploads

DoCASU would be useless without the ability to upload and download files into and out of the repository. First, there has to be an upload window with a file upload control that browses the files on the local file system. File upload involves launching a popup window, providing a form to let the user specify the file, and writing a handler that posts the form to the REST API.

Upload Dialog

The upload form is displayed in an inline popup which gets called by this code. This component relies on the extJS multiple file uploader Ext.ux.UploadPanel from http://filetree.extjs.eu/

```
function showUploadFile(folder) {

      var win = new Ext.Window({
            width:280
            ,minWidth:265
            ,id:'winid'
            ,height:220
            ,minHeight:200
            ,layout:'fit'
            ,border:false
```

```
            ,closable:true
            ,title:'Upload Content'
            ,iconCls:'icon-upload'
            ,items:[{
                    xtype:'uploadpanel'
                   ,buttonsAt:'tbar'
                   ,id:'uppanel'
                   ,url:'ui/node/content/upload'
                   ,path:folder // nodeId of the parent folder
                   ,maxFileSize:1048576
                   ,enableProgress:false // not implemented cf. progress.get.js
                   ,progressUrl:'ui/node/content/progress'
                   ,singleUpload:false // upload a file at a time
            }]
      });
      win.show();
            var uploadPanel = win.items.map.uppanel;
      //debugger;
            uploadPanel.on('allfinished', function (obj) {
                  reloadView(true);
            });
}
```

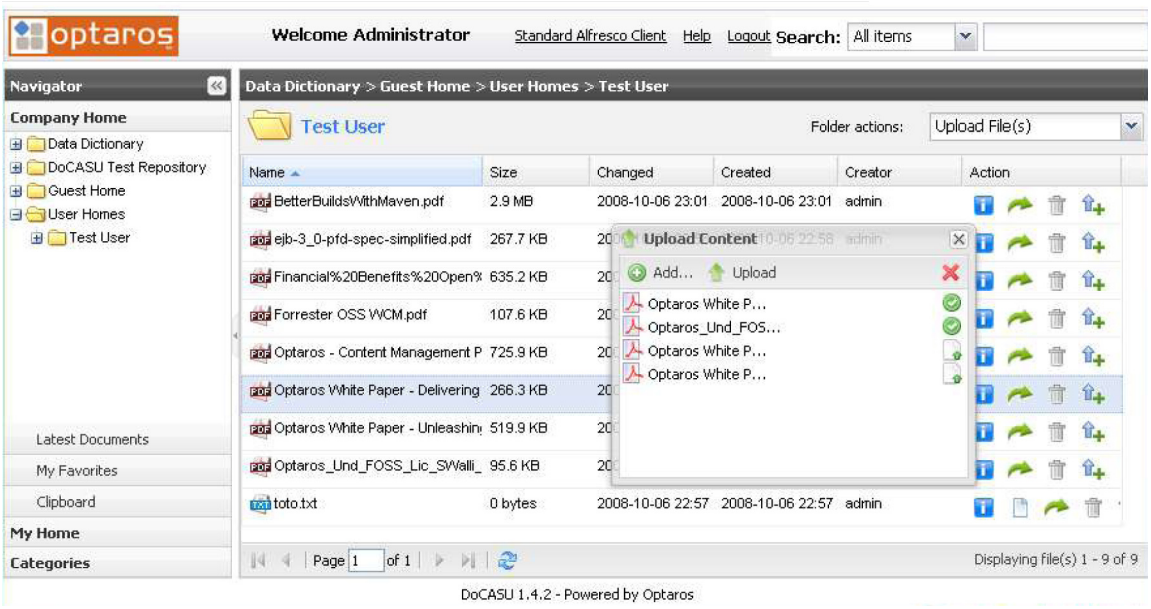

Figure 9: Multiple file upload

File Downloads

The file download is done using Alfresco's standard downloadUrl mechanism and does not require any additional web scripts.

Building and deploying the application

The DoCASU application is deployed as an AMP (Alfresco Module Package) file. It is freely available for you to download, to use, and to contribute fixes and enhancements.

Step-by-Step: Downloading and Building DoCASU

To get the DoCASU source code, build, and deploy the application, follow these steps:

1. Check-out the code from the Subversion repository at code.optaros.com:
   svn -co http://code.optaros.com/svn/docasu/trunk
2. Build the code using Apache Ant:
   ant package-amp
3. Copy the docasu.amp file to your $ALFRESCO_HOME/amps directory.
4. Stop the running Alfresco server.
5. Either patch the alfresco.war archive with the AMP files using Alfresco Module Management Tool:
   ${ALFRESCO_HOME}/apply_amps.sh
6. OR, if the apply_amps script is not part of your Alfresco distribution, you can use Alfresco's command line AMP tool:
   java -jar alfresco-mmt.jar install <path-to-amp> <path-to-alfresco.war>
7. Either unzip the patched WAR on top of your existing installation or remove the tomcat/webapps/alfresco folder and re-deploy the WAR.
8. Start Tomcat.
9. Watch the logs for:
   INFO [ModuleServiceImpl] Found 1 module(s).
   INFO [ModuleServiceImpl] Starting module 'com.optaros.alfresco.docasu' version 1.x.y.
10. Test the successful deployment by pointing your favorite browser to: http://localhost:8080/alfresco/wcs/docasu/ui
11. Login and explore.

Roadmap

Optaros expects DoCASU users and the community to guide us through selecting the most important requirements—we will adapt the roadmap as we learn. So feel free to get involved and provide suggestions or contribute code. The community workspace can be found at http://code.optaros.com/trac/docasu/

The key features planned at the time of this writing are:

- Providing a "drop zone" for dragging and dropping files from/onto the desktop.
- Creating configurable skins and better customization capability.
- Usability improvements.
- Complete localization and internationalization.
- Investigate how CMIS standard can be leveraged for DoCASU

Please refer to the online Roadmap for up-to-date information at http://code.optaros.com/trac/docasu/wiki/Roadmap

Summary

This white paper showed an example of a complete web application built with a Rich Internet Application framework (Ext JS) in the presentation tier making AJAX calls to a custom REST API hosted by the Alfresco repository. The application, DoCASU, is a real-world application Optaros built for a client and then made freely available as an open source project on www.docasu.org .

Specifically, you have seen:

- How Optaros designed and built the DoCASU user interface
- Some of the major architectural and design drivers
- The main parts of the DoCASU codebase, including details of how Ext JS was leveraged to build the components
- How to download and install the application on top of your own Alfresco repository

For further information about building applications with Alfresco, you can refer to the Alfresco Developer Book (http://www.packtpub.com/alfresco-developer-guide/book)

Obviously, Optaros wants to grow and support a community around solutions like DoCASU, but hopefully the deconstruction of DoCASU in this paper showed you what's possible with web scripts, illustrated why REST together with frameworks like Ext JS can be so powerful, and sparked some ideas for your own custom web applications that need to leverage content as a service.

About Optaros

Optaros, an international consulting and systems integration firm, designs and assembles fully supported, Next Generation Internet (NGI) solutions that deliver superior business performance. The unique Optaros Assembly Methodology (OptAM) leverages open source, open standards, service oriented architectures, rich interface design, and a global collaborative delivery model to assemble solutions faster and more flexibly than traditional build, buy, or rent options. Optaros serves over 90 clients including ABB, Biogen, Gtech, Merck/Serono, Movielink, Swisscom, Swiss Supreme Court, The New York Times, and many others. For more about Optaros, go to www.optaros.com

About the Authors

Olivier Pépin is Technical Director for Optaros Europe. Olivier Pépin is a technical architect and has 10 years of experience in delivering technology solutions in diverse enterprise environments for professional services firms. Mr Pépin's areas of expertise includes enterprise architectures, web architectures, SOA, JEE, and ECM.

Jeff Potts is the Director of the ECM Practice at Optaros, Inc. and has over 15 years of IT and technology implementation experience in IT departments and professional services organizations. Mr. Potts' areas of expertise include document management, content management, workflow, collaboration, portals, and search.

Optaros Contacts

- DoCASU Team
  E-Mail: docasu@optaros.com

- USA: Brian Otis
  E-Mail: botis@optaros.com
  Tel: +1 (617) 227-1855 x8110

- Switzerland (Geneva): Kay Flieger
  E-Mail: kflieger@optaros.com
  Tel: +41 (0)22 731 84 20

- Schweiz (Zürich): Sebastian Wohlrapp
  E-Mail: swohlrapp@optaros.com
  Tel: +41 (0)44 362 11 11

- Deutschland: Sascha Rowold
  E-Mail: srowold@optaros.com
  Tel: +49 (0)89 54 80 48 98

- UK: Peter Short
  E-Mail: pshort@optaros.com
  Tel: + 44 207 953 4054

Disclaimer:

Optaros makes no representations or warranties with respect to the contents or use of this document, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose.