OCOPOMO
Open Collaboration in Policy Modelling

D4.2 System and User Documentation

SD-1: System Documentation of OCOPOMO Alfresco Tools

Document Full Name	**OCOPOMO_D4.2-SD1_Alfresco-SysDoc.doc**
Date	**16/04/2013**
Work Package	**WP4: Integration of components**
Lead Partner	**Intersoft**
Authors	**Peter Bednár, Carsten Hartenfels**
Document status	**v1.00 FINAL**
Dissemination level	**PUBLIC (PU)**

TABLE OF CONTENTS

1. INTRODUCTION 3

1.1. SPRING WEB SCRIPTS 3
1.2. SPRING SURF 3

2. GENERIC ARCHITECTURE OF OCOPOMO ALFRESCO COMPONENTS 5

2.1. FILE LOCATIONS 6
2.2. SETUP OF DEVELOPMENT ENVIRONMENT AND DEVELOPMENT CYCLE 7

3. STRUCTURE OF SOFTWARE PACKAGES 8

3.1. COMMON PACKAGE 8
3.1.1. Traceability extensions 9
3.1.2. CCD Explorer 9
3.1.2.1. File Index 9
3.1.2.2. Program Arguments 10
3.1.2.3. Java Web Start 11
3.1.2.4. Applet 11
3.1.2.5. Comments 13
3.1.3. Wiki Page Formatting extensions 13
3.1.4. Localization 14
3.2. POLLING PACKAGE 14
3.3. CHAT PACKAGE 18
3.4. VISUALIZATION PACKAGE 23

4. CONCLUSION 25

1. INTRODUCTION

The main implementation technology for developing the suite of OCOPOMO Alfresco tools is Spring Web Scripts, http://wiki.alfresco.com/wiki/Web_Scripts, and Spring Surf framework, http://wiki.alfresco.com/wiki/Spring_Surf. These technologies will be detailed in the following sections.

1.1. SPRING WEB SCRIPTS

Spring Web Scripts can be divided to data web scripts and presentation layer web scripts used in the Spring Surf framework. Data web scripts provide access to the content stored in the repository through REST-like API bounded to the URI that responds to HTTP methods such as GET, POST, PUT, and DELETE.

Web Script is defined using the following components:

- ***XML configuration descriptor file*** - defines identification name and documentation description, URL pattern on which the script will be bounded and configuration options for authentication and transactions. A URI template is a URI containing tokens that may be substituted with actual values. Tokens may represent values to query parameters or values within the URI path where the syntax for expressing a token is {<token name>}. For example, to specify URI with two query parameters - one named "a" and the other named "b", the pattern should be defined like: /add?a={a}&amp;b={b}.
- ***Controller*** - implements application logic programmed for the script. Script controller can be programmed using the server-side JavaScript or Groovy or it can be implemented as the Java class. The role of the controller is to process data received as the API parameters and/or create a data model, which will be rendered in the API response using the Web Script template.
- ***Response template*** - render output in the correct format for specific needs, such as HTML, Atom, XML, RSS, JSON, CSV, or any combination of these. Templates can be implemented in FreeMarker or PHP.

All scripts developed for OCOPOMO Alfresco tools are implemented using the server-side JavaScript and FreeMarker templates. Communication protocols are based on JSON format.

More information about the Spring Web Scripts can be found at:

http://docs.alfresco.com/4.0/index.jsp?topic=%2Fcom.alfresco.enterprise.doc%2Fconcepts%2Fws-architecture.html

1.2. SPRING SURF

The user interface of OCOPOMO Alfresco Tools is implemented as the extensions or modules for Alfresco Share (http://wiki.alfresco.com/wiki/Alfresco_Share), which is implemented using the Spring Surf framework.

Surf is a Spring framework extension for building new Spring framework applications or plugging into existing Spring web MVC (Model, View, Controller) applications. Spring Web MVC provides separation between the application Model, View, and Controller (known as MVC). You can use Surf with other popular Spring Web MVC technologies including Tiles, Grails, and Web Flow.

The main part of the Surf is the View composition framework, which defines the following user interface items:

- ***Page*** - defines user interface view bounded to the specified URL of the Alfresco Share application. User can navigate to the page with browser. Pages are defined using the XML descriptor file, which specifies documentation title and description, reference to the template instance used for page rendering and authentication options.

- ***Page Template* -** defines the layout for pages. HTML markup of the layout is defined in the FreeMarker template type, which divides page to regions. Region is a placeholder in the page layout, which is rendered by specified component. Particular page specifies template instance - XML descriptor file, which bounds particular components (presentation web scripts) to particular region specified for the selected template type. It means that the same layout defined by one template type can be reused on many pages and rendered in different way depending on which components were mapped to the template regions in template instance descriptor.
- ***Page Component* -** usually associates a region with a presentation web script, which generate the HTML markup. The same Web script can be reused on multiple pages or in different regions on the same page, i.e. each view can be effectively decomposed to the reusable parts.

More information about the Spring Surf can be found at:

http://docs.alfresco.com/4.0/index.jsp?topic=%2Fcom.alfresco.enterprise.doc%2Fconcepts%2Fws-architecture.html

2. GENERIC ARCHITECTURE OF OCOPOMO ALFRESCO COMPONENTS

The scheme depicted in Figure 1 presents the generic architecture used to implement OCOPOMO Alfresco Components. It summarizes interaction of all components in Spring Web Scripts or Spring Surf framework The following sections contain descriptions and instructions on how to perform the most common operations of the *Collaboration* and *Scenario Generation* tools. Operations are grouped according to the site components.

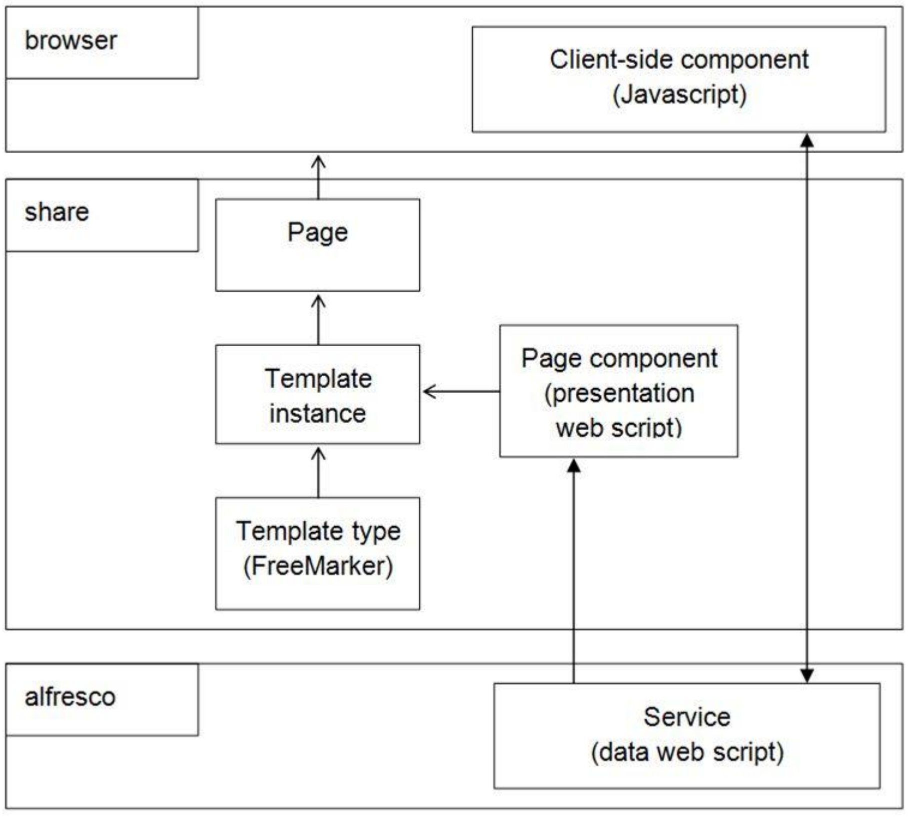

browser
Client-side component (Javascript)
share
Page
Template instance
Page component (presentation web script)
Template type (FreeMarker)
alfresco
Service (data web script)

Figure 1: Architecture of OCOPOMO Alfresco components.

When the user navigates to the Alfresco Share page, the request is process by the Spring Surf framework using the following procedure:

1. At first, Surf matches the requested URL to the page descriptor. Page descriptor points to the template instance with the associated template type and mappings of page regions to page components.
2. For each page region defined in the template instance, Surf framework invokes associated component web script, which generates HTML markup for the region. Component web script fetches data model from the services. Service APIs are usually implemented using the data web scripts generating the data model encoded in JSON.
3. Surf will invoke FreeMarker template engine to parse and render page template type. Page template type generates final HTML mark-up for the page view, which consists of the main layout specified in the template type and embedded mark-ups for all regions.

Alternative way how the view page is generated/modified is based on the dynamic HTML approach, which can be combined with the content generated by the Surf request processing. In dynamic HTML approach, view is modified using the JavaScript framework such as Yahoo YUI library, which is directly modifying browser HTML representation of the page. Data model is fetched directly from the service by the client-side JavaScript component. Both approaches can be arbitrary combined.

Data are updated directly by invoking of the service APIs by client-side JavaScript component, which encode updated data into the JSON format and invoke POST or DELETE HTTP request to the service URL.

2.1. FILE LOCATIONS

Web script component file names adhere to the following naming conventions:

- Descriptor: *<script id>.<HTTP method>.desc.xml*
- Controller: *<script id>.<HTTP method>.js*
- Template: *<script id>.<HTTP method>.<format>.<status code>.ftl*

where:

- *<script id>* identifies the web script and must be the same for all components of the same script;
- *<HTTP method>* specifies which HTTP method will initiate the web script;
- *<format>* specifies a format generated in the response (usually JSON for data scripts or HTML for presentation scripts);
- *<status code>* [optional] specifies that template will be used to render output only when the controller set specified HTTP response status code.

The Alfresco platform provides an extension mechanism, which allows overwriting of the settings or customization of the implemented pages and web scripts without the overwriting the original files. The extensions are stored in the %ALFRESCO\tomcat\shared\classes directory, where %ALFRESCO is the root directory of the local installation of the Alfresco software. Table 1 provides the list of directories for OCOPOMO Alfresco Tools (all directories are relative to the %ALFRESCO\tomcat\shared\classes directory).

Table 1: Directories for the installation of OCOPOMO Alfresco tools.

File locations for web scripts and pages
alfresco/extension/templates/webscripts/org/alfresco/slingshot/<page component> Data web scripts for the *<page component>* (i.e. for example, pollings or wiki).
alfresco/web-extensions/site-data/pages Page descriptors for user interface views.
alfresco/web-extensions/site-data/template-instance Template instances XML descriptor files, which associate components with the regions in layout template for the specified page.
alfresco/web-extensions/site-webscripts/org/alfresco/components/<page component> Presentation web scripts for the *<page component>* (i.e. for example, pollings or wiki). Web scripts for page components.
alfresco/web-extensions/templates/org/alfresco FreeMarker templates for layout template types.

Additionally to files for Spring Web Scripts and Spring Surf frameworks, Table 2 summarizes directories for web resources (such as CSS, icon picture files, etc.) and client-site JavaScript. All web resource files are deployed directly in the %ALFRESCO\tomcat\ webapps\share directory.

Table 2: Directories for web resources of OCOPOMO Alfresco tools.

File locations for web resources
components/<page component> web resources and client-site JavaScript files for the <*page component*> (i.e. for example, pollings or wiki).
modules/<module> web resources and client-site JavaScript files for the reusable <*module*> (i.e. online text editor, form engine, etc.)
themes/defaults shared web resources (for example picture files with the application logo etc.)

2.2. SETUP OF DEVELOPMENT ENVIRONMENT AND DEVELOPMENT CYCLE

To start developing new components or to customize OCOPOMO Alfresco components, the developer should install local installation of the Alfresco platform. The installation instructions for selected platform available on:

http://docs.alfresco.com/4.0/index.jsp?topic=%2Fcom.alfresco.enterprise.doc%2Fconcepts%2Fsimpleinstalls-community-intro.html

During the development, please follow these steps:

1. If you are developing new web script or page, copy descriptors, scripts and template files for the new component into the extension directories described in the previous chapter File locations.
2. Restart your Alfresco installation to register new scripts and pages.
3. Implement any changes in the templates or JavaScript files to modify application logic or generated content.
4. To update data web scripts deployed on the Alfresco CSM; navigate to http://localhost:8080/alfresco/service/index page. On this administration page you can list all available web scripts. Click on *Refresh web scripts* to refresh the running installation with the edited changes. To update presentation web scripts or pages, navigate to http://localhost:8080/share/service/index page to refresh Alfresco Share scripts.
5. Refresh modified page in the browser.

The refreshing of the web scripts is not necessary when you are modifying client-site JavaScript files located in the Alfresco Share deployment directory %ALFRESCO\tomcat\webapps\share. In this case, you can directly refresh modified page in the browser.

3. STRUCTURE OF SOFTWARE PACKAGES

The distribution of the OCOPOMO Alfresco components is divided to the following installation packages:

- ***common package*** - contains common OCOPOMO extensions for the base Alfresco installations. Most extensions are related to the Wiki page component and Document Library component, which is used for the scenario editing and extensions for the traceability. This package also contains Slovak localization for the Alfresco platform.
- ***polling package*** - contains files implementing Polling manager.
- ***chat package*** - contains files implementing Chat manager.
- ***visualization package*** - contains extensions of the Wiki page component and internal HTML editor for embedding of the data visualization charts. This package also contains data proxy required for the data visualization and CCD Explorer.

Each package can be installed independently from others, i.e. administrator can decide to use only the selected tools and developers can install only the customized component. The system documentation described in the following chapters is organized according to the installation packages.

3.1. COMMON PACKAGE

The ***common*** package contains a set of basic configuration files, web scripts and resources for OCOPOMO extensions of components, which were reused from the Alfresco Share collaboration platform (such as Wiki page and Document Library used for scenario editing and presentation of simulation results). It also contains localization files for Slovak and Italian language (English language is supported by default). Scheme depicted in Figure 2 summarizes these extensions and their interaction with the components reused from the Alfresco platform.

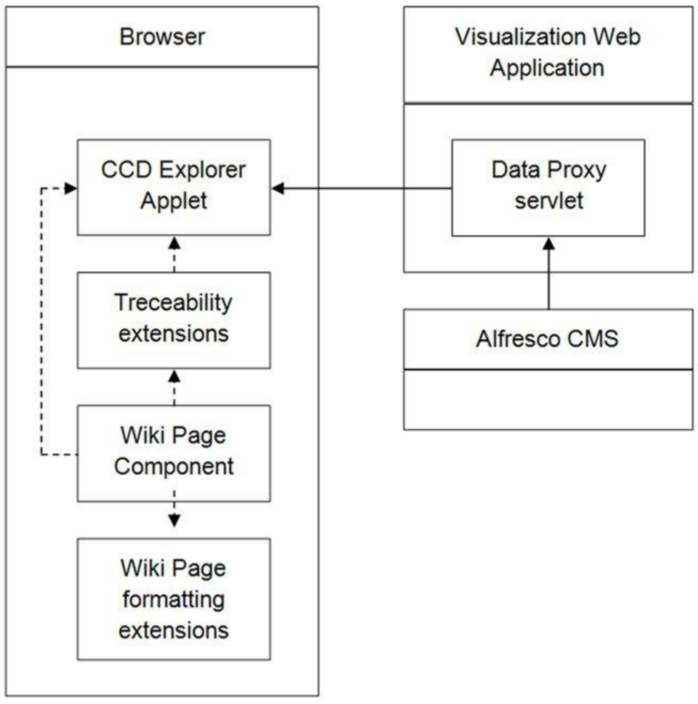


Figure 2: Extensions provided within the *common* package.

3.1.1. Traceability extensions

The suite of *Traceability tools* is the main extension of OCOPOMO Alfresco tools, which consist of the extensions for Wiki page HTML formatting that enables a presentation of traceability annotations as well as an integration of the CCD Explorer applet.

The Wiki page visualization is extended with a new content parser, which extracts annotation elements from the Wiki HTML content. Annotations elements have the following structure:

```
<span class="annotation" id="<element id>" data ="<JSON annotation object>">
annotated HTML content
</span>
```

where <element id> is the ID of the span HTML element used for referencing and <JSON annotation object> is the object with references to the relevant CCD concepts assigned in this annotation and array of links to the evidence-based scenarios and background documents used for the annotation. More information about the embedded JSON annotation object can be found in the documentation of Simulation Analysis tool.

Traceability content parser registers for each annotation element mouse click handler, which displays popup dialog window with the list of relevant links to the evidence-based scenarios and background document. In this dialog, user can also invoke CCD Explorer, which is lunched with the filtering options displaying only the part of the CCD diagrams with the concepts relevant for the selected annotation.

The CCD Explorer applet is deployed as the resource in %ALFRESCO\tomcat\webapps \share\applets. Besides of the annotation dialog, the Wiki page view was also extended with the direct link to CCD Explorer, i.e. user can directly start to brows unfiltered diagrams.

3.1.2. CCD Explorer

The CCD Explorer, also referenced as CCD Model Explorer applet, is available both as a desktop application, provided via Java Web Start of JRE (version 6 or higher is required) , as well as an applet built into a web site. In case of the OCOPOMO ICT toolkit, the CCD Explorer is embedded into the Alfresco web application.

CCDExplorerApplet_JavaDoc.zip **-** see the accompanying zip package that contains the documented source code in JavaDoc and implementation code examples for the CCD Model Explorer applet.

3.1.2.1. File Index

To have any functions available, the CCD Web Interface needs to be provided with a file index. That file index comes as an XML file called index.xml. It should be created using the index file generator pointing to the folders created by the Eclipse CCD tool. All file paths are relative to the folder the index.xml resides in. See http://userpages.uni-koblenz.de/~ocopomo/ccdexplorer/ for an example file. In the next paragraphs, we explain the structure of the index.xml file used for CCD Explorer.

The index node is the root node of the index.xml. It needs to have a name attribute specifying the name of the CCD project.

Example:

```
<index name="KosiceExtended"> ... </index>
```

Files referenced in the index.xml represent source and data files used in the related OCOPOMO Java project. A file node needs to have a name attribute specifying the file name and a path attribute specifying its path relative to the folder the index.xml resides in. File nodes should be closed immediately, since they cannot contain other nodes. Currently supported file types are *.ccd, *.ccd_actions, *.ccd_diagram, *.ccd_instances, and *.txt.

Example:

```
<file name="KosiceExtended:ccd" path="KosiceExtended:ccd"/>
```

Folders in the index.xml represent folders in the related OCOPOMO Java project. A documents folder is usually necessary for CCD files. A folder node needs to have a name attribute specifying its name and can contain more folders and files.

Example:

```
<folder name="documents"> ... </folder>
```

3.1.2.2. Program Arguments

The CCD Explorer needs to be provided with a handful of arguments to function accordingly. Depending on the way the program is implemented, there are different ways to specify these arguments, as it is explained in Table 3.

Table 3: Program arguments for the CCD Explorer tool.

Argument	**Description**
path	The path argument must be a URI that points to the folder where the index.xml resides. Do not directly point the path to the index.xml, merely point it to the folder it is in. A path argument must be given. The prefix for this argument is path. Example: if the index.xml's location is http://www.ocopomo.eu/project/index.xml, then the correct path for this project would be http://www.ocopomo.eu/project/.
user	The user name of the current user. Will be used for comments. If a user name is not provided, the user will not be able to comment on anything. The prefix for this argument is *user*.
url	The full URL to the server where comment data should be uploaded. When synchronizing comments, the CCD Web Interface will attempt to connect to that exact URL and upload the synchronized comment.xml data there. If a URL is not provided or if the given URL is invalid, it will not be possible to upload user comments to the server. The prefix for this argument is *url*.
com	Sets the upload type for comments. Possible values are *everything* or *all* for upload of the entire comments.xml and *changes* or *dif* for cumulative upload of the changes only. For testing purposes comments can be written to a local file using *tofile* or *file*. If the upload type is not provided or if the upload type is invalid, comment upload will be disabled and the user will not be able to comment on anything.
lang	Sets the language of the CCD Web Interface. Currently available languages are *en* (English) and *de* (German). If no or an invalid locale is given, the language will default to English. The prefix for this argument is *lang*.

3.1.2.3. Java Web Start

To deploy a Java Web Start program, a JNLP file needs to be created. See the official JNLP documentation for a structure that should be applied for the respective JNLP file at http://docs.oracle.com/javase/tutorial/deployment/deploymentInDepth/jnlpFileSyntax.html, as well as the http://userpages.uni-koblenz.de/~ocopomo/ccdexplorer/ for an example JNLP file.

Arguments for the JavaWeb Start are encompassed in <argument> tags and are located within the <application-desc> tag. Only one argument can be in every <argument> tag. They can be provided in any order and must start with their prefix and an equal sign '='.

Example:

```
<application-desc main-class="eu.ocopomo.web.WebInterface">
  <argument>path=http://wwwocopomo.eu/project/</argument>
  <argument>user=TestUser</argument>
  <argument>url=http://wwwocopomo.eu/project/comments.xml</argument>
  <argument>lang=en</argument>
</application-desc>
```

3.1.2.4. Applet

The CCD Explorer can be implemented into a HTML file using Java's applet functionality. For an example HTML file, see http://userpages.uni-koblenz.de/~ocopomo/ccdexplorer/.

It is recommended to use <applet> tags (instead of <object> tags) to implement the applet because it enables pushing commands via JavaScript.

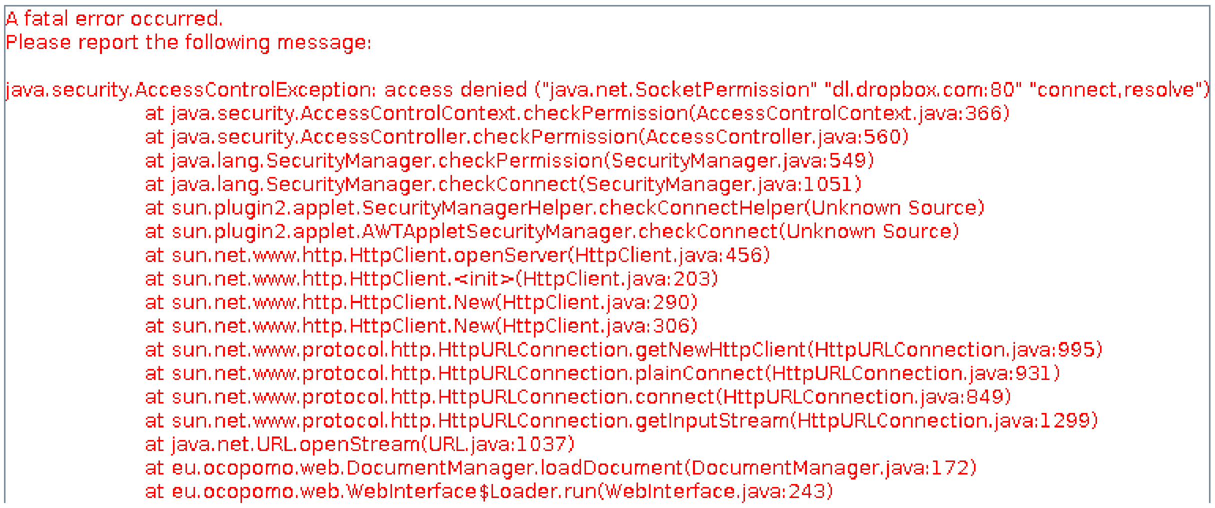
Arguments are passed to an applet using the <param> tag within the <applet> tags. The name attribute of the parameters are the prefixes presented above in Table 3.

Example:

```
<applet id='ccdapplet'
  archive='WebInterface.jar'
  code='eu.ocopomo.applet.CCDApplet.class'
  width=800
  height=600>
   <param name="path" value="http://www.ocopomo.eu/project/">
   <param name="user" value="TestUser">
   <param name="url" value="http://www.ocopomo.eu/project/comments.xml">
   <param name="lang" value="en">
</applet >
```

As opposed to the other ways of running the CCD Explorer tool, there are some security restrictions on the file paths of the applet. As it is described at http://docs.oracle.com/javase/tutorial/deployment/applet/security.html, applets can only access URLs from the host they came from. Practically, this means the applet cannot access paths "above" its own path. For example, if the JAR file of the applet is located at http://www.ocopomo.eu/project/CCDWebInterface.jar, it can only access files and folders starting with http://www.ocopomo.eu/project/. Trying to pass a URL that does not start with this host will not work and cause an error.

Hence both the path and url argument must not point anywhere the applet cannot access, otherwise upon starting the applet an error message will be shown, similar to the one in Figure 3. This also means an applet that is run from a local file cannot access the internet.



```
A fatal error occurred.
Please report the following message:

java.security.AccessControlException: access denied ("java.net.SocketPermission" "dl.dropbox.com:80" "connect,resolve")
	at java.security.AccessControlContext.checkPermission(AccessControlContext.java:366)
	at java.security.AccessController.checkPermission(AccessController.java:560)
	at java.lang.SecurityManager.checkPermission(SecurityManager.java:549)
	at java.lang.SecurityManager.checkConnect(SecurityManager.java:1051)
	at sun.plugin2.applet.SecurityManagerHelper.checkConnectHelper(Unknown Source)
	at sun.plugin2.applet.AWTAppletSecurityManager.checkConnect(Unknown Source)
	at sun.net.www.http.HttpClient.openServer(HttpClient.java:456)
	at sun.net.www.http.HttpClient.<init>(HttpClient.java:203)
	at sun.net.www.http.HttpClient.New(HttpClient.java:290)
	at sun.net.www.http.HttpClient.New(HttpClient.java:306)
	at sun.net.www.protocol.http.HttpURLConnection.getNewHttpClient(HttpURLConnection.java:995)
	at sun.net.www.protocol.http.HttpURLConnection.plainConnect(HttpURLConnection.java:931)
	at sun.net.www.protocol.http.HttpURLConnection.connect(HttpURLConnection.java:849)
	at sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:1299)
	at java.net.URL.openStream(URL.java:1037)
	at eu.ocopomo.web.DocumentManager.loadDocument(DocumentManager.java:172)
	at eu.ocopomo.web.WebInterface$Loader.run(WebInterface.java:243)
```

Figure 3: Example of the applet error message.

It is possible to push a string of commands to the CCD Explorer applet using JavaScript. The sample HTML file located at http://userpages.uni-koblenz.de/~ocopomo/ccdexplorer/ demonstrates some examples for these functions.

In the JavaScript commands outlined below, the parts in [brackets] are optional. All commands are separated by a semicolon ';'. Avoid whitespace.

push

The push(String) method is used to open any specified file. If a diagram is opened, it can be filtered by passing UUIDs from the CCD file.

A command is formatted as follows: *uri;type;fileName;[full;][filter;][uuid;uuid;...]*

pushIds

The pushIds(String) method is used to open a diagram and filter it without needing a URL.

A command is formatted as follows (note that this method needs at least one valid UUID): *[full;]filter;uuid;[;uuid;uuid;...]*

uri

The URI of the file relative to the applet's path (see in Table 3).

type

The file type. Possible file types are *actions*, *diagram*, *instances*, *tree* and *txt*.

Filtering is possible with the first three file types.

FileName

The name of the file as it should appear on the tab inside the applet. This is purely cosmetic.

full

If this command is given, the applet will be put into full-view mode after the file is opened. Otherwise, it will be put into normal view, even if the applet was in full view previously.

filter

Sets the filter for the opened diagram. Available filters are *none* for no filtering, *lazy* for only showing selected shapes and the shapes they are connected to and *strict* for only showing selected shapes. If an invalid filter is given, it will default to *none*.

If the opened file is not a diagram, this command will be ignored.

uuid

The *xmi:id* of an element that should be selected, as found in the .ccd file for the diagram. Invalid or empty UUIDs will be ignored.

If the opened file is not a diagram, this command will be ignored.

3.1.2.5. Comments

The CCD Explorer supports adding comments to items in the CCD tree and shapes in diagrams, and commenting other comments in a web-forum-like manner.

Comments are saved in an XML file called comments.xml. It must reside in the same folder as the index.xml file. The comments.xml is structured as follows:

root **-** The outmost node of the XML document is usually called root, but it may have a different name. Attributes of this node are retained, but ignored. The only children a *root* node can have are *comment* nodes.

comment - This node represents a single comment. It must be a direct child of the *root* node. A comment node must not have any child nodes. A comment node has the following attributes:

id - The unique ID of the comment. This ID will remain the same even if the comment is edited. It is generated as follows:

[author].[current system time in milliseconds].[MD5 hash of the heading]

parent - The unique ID of the parent of the comment. May be another comment or an *xmi:id* from the CCD file.

author - The user name of the author of the comment.

head - The caption of the comment.

body - The actual comment text.

Upload - To upload comments, the CCD Explorer will connect to the given URL (see the *url* argument in Table 3) and push XML data via HTTP.

Everything - With this upload type, the comments.xml from the server is synchronized with the local comments. Then, the whole comments.xml will be uploaded, including all previous comments already existent in the comments.xml on the server. That means, all the server needs to do is replace the old comments.xml with the new one.

Changes Only - With this upload type, only the XML data of added or changed comments is uploaded as an XML document. The server will need to add new comments into the existing comments.xml and replace edited comments.

To File **-** For testing purposes, comments can be written to a local file system. This upload type will only work if the *path* argument (see in Table 3) points to a local file.

Disabled **-** If comment upload is disabled, the user will not even be able to comment on anything or edit their comments in the first place. No upload attempts will be made.

Comment upload is disabled if a user name is not provided (the *user* argument in Table 3) or if no or an invalid upload type was given (the *com* argument in Table 3).

3.1.3. Wiki Page Formatting extensions

Besides of the traceability annotations, visualization of the Wiki pages was extended with the various extensions for the formatting of the HTML content that improve visual appearance of Wiki pages and usability. The extensions are implemented as the add-ons for the Wiki page parser, which parses HTML content and replaces links typed in Wiki notation, for example:

```
[[<page name>|<link label>]]
```

by simple HTML `<a>` elements.

The extensions includes:

- Support for dynamic table of contents generated from headings;
- Support for embedded preview of documents stored in the Document Library;
- Support for extended tables implemented using the YUI DataTable component. It supports for example sorting of values by clicking on the column header and can be better customized than regular HTML tables.

All these extensions dynamically modify source HTML content and add new styles and embedded elements into the code of the visualized page (content is modified on the client side internally in the browser and the source Wiki page stored in the repository is not modified).

3.1.4. Localization

The localization and internalization is implemented using the Java resource bundles. Bundles are stored in the property files for each supported locale. Names of the property files adhere to the standard naming convention <bundle name> _<locale>.properties.

Property files are stored in the same directory like the web scripts of the corresponding components, i.e. alfresco/extension/templates/webscripts/org/alfresco/slingshot/<page compo-nent> for data scripts and alfresco/web-extensions/site-webscripts/org/alfresco /components/<page component> for presentation scripts. All resource directories are relative to the Alfresco extension directory, i.e. %ALFRESCO\tomcat\shared\classes.

For client-side components, localized messages are injected as the initialization parameter in the JavaScript code generated by the corresponding presentation web script. For example, the `page.get` presentation web script generates `include` elements for the client-side page JavaScript component together with an initialization code for this component. To access a localized message for the current locale in the client-side component, it is possible to use the construct:

```
this.msg("<message property key>");
```

which is a built-in JavaScript function, where `<message property key>` is the key in the message bundle for the localized label/message. The locale for the current user is detected according to the preferred language set in the browser configuration.

Besides of the page components and web scripts, localization property files for data models (types, properties, enumerated values) and workflow elements (actions, forms, etc.) are stored in the alfresco/messages directory.

3.2. POLLING PACKAGE

The ***polling*** package contains files implementing the Polling manager component. It consists of the data and presentation web scripts managing the pollings, polling posts and results. The following scheme, presented in Figure 4, summarizes the internal architecture of the Polling manager and interactions between the data web scripts, presentation web scripts and pages.

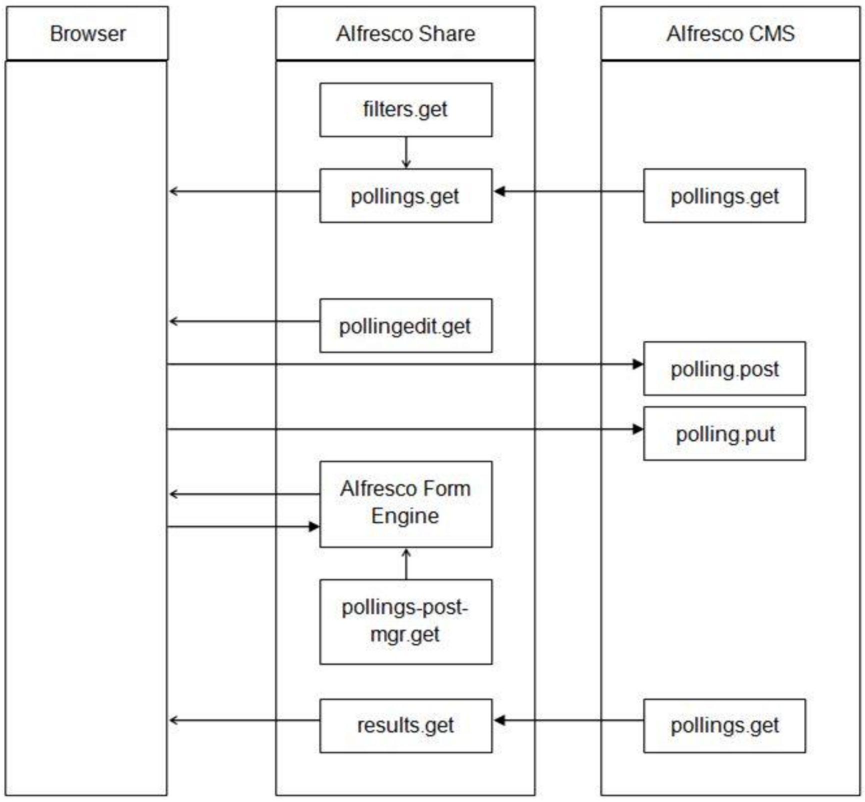


Figure 4: Inner structure of the *polling* package.

The web scripts implemented in the *polling* package, which serve as application interfaces for provided functionality, are presented in the following Table 4.

Table 4: Data and presentation web scripts for the *polling* package.

Data web script org/alfresco/slingshot/pollings/polling.post		
URL	/api/pollings/site/{site}/{container}	
Parameters	site	identifier of the collaboration site
	container	identifier of the content node container where data will be stored (pollings)
Request	POST JSON object	
	title	String title for new polling
	description	String description for new polling
	pollingItemType	String with the reference to the type used for the polling post. This type defines properties, which corresponds to polling questions
	tags	Tags used for classification of new polling
Response	Status code with error message in the case of error	
Description: Creates new polling with the specified item type. Item type defines properties, which corresponds to polling questions.		

Data web script org/alfresco/slingshot/pollings/polling.put		
URL	/api/pollings/site/{site}/{container}/{path}	
Parameters	site	identifier of the collaboration site
	container	identifier of the content node container where data will be stored (pollings)
	path	path to the modified polling (node reference)
Request	PUT JSON object	
	title	String new title for the polling
	description	String new description for the polling
	pollingItemType	String with the reference to the type used for the polling post. This type defines properties, which corresponds to polling questions
Response	Status code with error message in the case of error	
Description: Updates existing polling with the new title, description and specified item type. Item type defines properties, which corresponds to polling questions.		

Data web script org/alfresco/slingshot/pollings/pollings.get		
URL	/api/pollings/site/{site}/{container}	
Parameters	site	identifier of the collaboration site
	container	identifier of the content node container where data will be stored (pollings)
Request	GET (empty)	
Response	JSON response object with the polling entities, each entity has the following properties:	
	nodeRef	String content node reference of the polling
	name	String name of the content node
	title	String title of the polling
	description	String short description of the polling
	pollingItemType	String with the reference to the type used for the polling post. This type defines properties, which corresponds to polling questions
	createdOn	String with the creation date
	creator	String reference to the person object of the creator who created this polling
	author	String reference to the author name (i.e. name of the creator)
	permissions	Object with read/write permissions for the current user invoking the request
	tags	Array of tags used for classification of this polling.
Description: Lists all existing pollings for the specified collaboration site.		

Data web script org/alfresco/slingshot/pollings/results.get		
URL	/api/pollings/results/site/{site}/{container}	
Parameters	site	identifier of the collaboration site
	container	identifier of the content node container where data will be stored (pollings)
	path	path to the polling (node reference)
Request	GET (empty)	
Response	JSON response object with the polling results	
	polling	String content node reference of the polling
	title	String title of the polling
	pollingItemType	String with the reference to the type used for the polling post. This type defines properties, which corresponds to polling questions
	totalCount	Number with the total count of posts send for this polling
	fields	Array with statistics objects for each field defined for the item type (i.e. for each polling question)
	fields/field	String name of the field
	fields/{field}:values	Object with label for possible answer and count of posts with this answer
Description: Lists actual results for the specified polling.		

Presentation web script org/alfresco/components/pollings/filters.get	
URL	/components/pollings/filters
Parameters	None
Request	GET (empty)
Response	HTML Markup
Description: Filter page component used for filtering of the pollings. Available options for filtering are stored in the XML configuration file. Default configuration includes All, User and Recent options.	

Presentation web script org/alfresco/components/pollings/polingedit.get	
URL	/components/pollings/pollingedit
Parameters	None
Request	GET (empty)
Response	HTML Markup
Description: Generates HTML markup for the main region of the polling-edit page. It includes Javascript and CSS resource files for the PollingEdit client-side component.	

Presentation web script org/alfresco/components/pollings/polling-post-mgr.get	
URL	/components/pollings/polling-post-mgr
Parameters	None
Request	GET (empty)
Response	HTML Markup
Description: Customizes HTML markup for the form generated for the polling posts. Default implementation includes post title to form header. The form is generated by the Alfresco Form Engine, which inspects definition of properties for the specified post item type and generate input controls for each field (question).	

Presentation web script org/alfresco/components/pollings/pollings.get	
URL	/components/pollings/pollings.get
Parameters	None
Request	GET (empty)
Response	HTML Markup
Description: Generates HTML content for the main region of the pollings page, which lists all available pollings. The list is filtered according to the options specified by the filters component.	

Presentation web script org/alfresco/components/pollings/results.get	
URL	/components/pollings/results.get
Parameters	None
Request	GET (empty)
Response	HTML Markup
Description: Generates HTML content for the main region of the results page. Controller of this component invokes /api/pollings/results data service to fetch result statistics for the specified polling. Results are rendered as the HTML tables.	

3.3. CHAT PACKAGE

The ***chat*** package contains files implementing the Chat manager component. It consists of the server managing the communication and exchange of messages and client implemented as the Alfresco dashlet. The scheme presented in Figure 5 summarizes the internal architecture of the Chat manager and interactions between the data web scripts, presentation web scripts and chat server servlet.

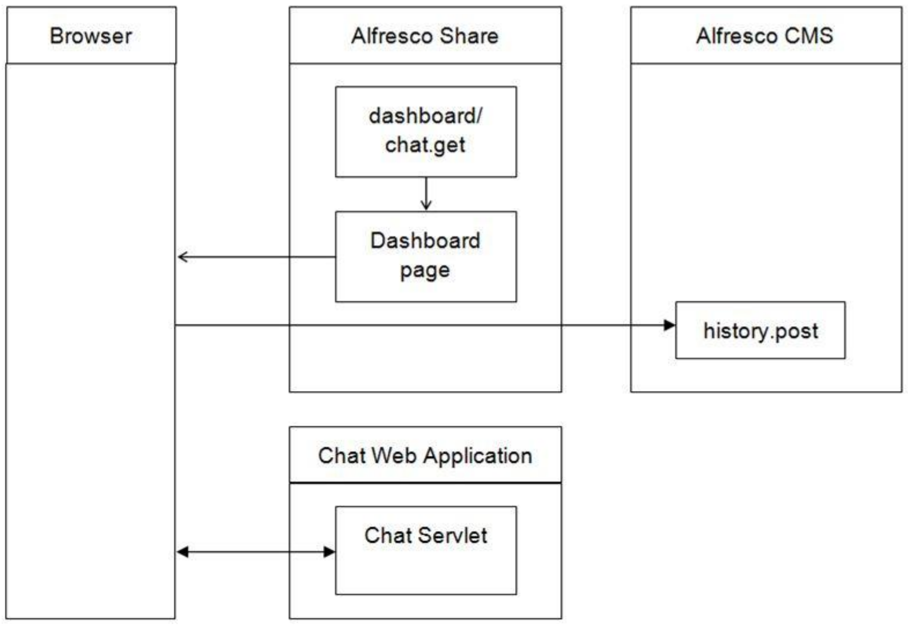


Figure 5: Inner structure of the *chat* package.

The web scripts implemented in the *chat* package, which serve as application interfaces for provided functionality, are presented in Table 5.

Table 5: Data and presentation web scripts for the *chat* package.

Data web script org/alfresco/slingshot/chat/history.post		
URL	/api/chat/site/{site}/{container}/history	
Parameters	site	identifier of the collaboration site
	container	identifier of the content node container where data will be stored (documentLibrary)
Request	POST JSON object	
	description	String description used for the generated document
	history	Array with the chat messages published as the document in Document Library
	name	String name of the generated document
	occupants	Array with the user IDs participating on this chat
	subject	String with the subject of the chat.
	title	String title used for the generated document
Response	Status code with error message in the case of error	
Description: Publishes history of the chat as the new document stored in the Document Library. The content of the new document is HTML formatted chat messages.		

Presentation web script org/alfresco/components/dashlets/chat.get	
URL	/components/dashlet/chats
Parameters	None
Request	GET (empty)
Response	HTML markup
Description: Generates HTML markup for the chat dashlet. Generate HTML code is combined together with the regions of other dashlets to form site dashboad page. It includes chat client-side Javascript component and chat CSS resource file in the page header.	

Next in Table 6 and in the subsequent list of communication patterns we present the details about the message exchange between the Chat client-side component and Chat servlet. The communication protocol is based on the XMPP protocol for real-time asynchronous communication; however the messages are encoded in JSON instead of XML.

Table 6: Parameters used in JSON objects during the *chat* communication.

Parameter (field)	**Description**
rid	Request id (type: integer) - used for synchronization of requests between client and server
sid	Session id (type: string)- unique identifier of the chat session
body (in main level of JSON)	Main structure for encapsulating other parts of the request / response (could contain more messages from one session)
type	Type of some specific message (type: string) - can be "presence" (for presentation of users), "message" (for chat messages and subject changes), "iq" (for query on rooms), "result" (for query result)
from	Id of user which initiates some communication or query (type:string) - id of some user (jid)
to	Id of communication element which is part of the communication (type:string) - id of some user (jid), id of some room (roomid), or id of site (siteid)
body (inside chat message type JSON)	Chat message string (type: string)
subject	Subject of chat (type:string) - chat room subject (in result of query) or new subject for change (in change subject request)
stamp	Timestamp of message (type:string) – timestamp in standard format (e.g., "2013-01-01T00:00:00Z"), indicates creation time for message
items	Container for results of query – contains JSON objects
roomid	Id of room in query result (type:string)
newmsg	Number of new messages (type:integer) – number of new messages in current particular room (used in result to query)

occupants	Container for occupants of current room – contains JSON objects with two fields: jid (type:string) – id of occupant, and name(type:string) - full name of user
close	Flag for closing of room (type:string) – if “close” is available in presence type of message (we are using “true” as value, but only presence of the field is enough), the room with roomid in “to” field is closed
history	Flag for sending the history after presence request (type:string) – if “history” is presented in response (we are using “true” as value, but only presence of the field is enough), client application knows that all messages in it (serialized as JSON objects one by one) are not new messages, but only old messages from current room
flush	Flag that is used in order to flush session queue of current user (type:string), if “flush” is presented in request (we are using “true” as value, but only presence of the field is enough) – used from the client for new open of chat – chat rooms are refreshed and all old messages in queue are flushed in order to avoid redundancy
openedRoom	Id of currently opened room of chat user (type:string) – roomid of such room, or NOT_OPENED_ROOM flag.

Patterns for Chat communication (with examples of requests and responses):

1. ***Connection to server*** – starting the communication from client with server – requires empty body from client, returns session ID in “sid” field.

   Client request: {"rid": 0, "from": "some_jid"}

   Server response: {"rid": 1, "sid":"some_sid "}

2. ***Empty body request*** – used for creation of loop, where client is waiting for changes, if nothing is done, empty response is send. Opened room field is send from the client (for synchronization of messages in currently opened room on client side – timestamp synchronization) with the room_id. This is used also for chat user object to setup same flag. If this field is not presented in request, user has not opened room currently and flag is setup to “NOT_OPENED_ROOM” flag’s value.

   Client request: {"sid":"some_sid","rid": 126, "body":"[]"}

   Server response: {"sid":"some_sid","rid": 126, "body":"[]"}

3. ***Query for rooms*** – request for rooms of current user – requires type “iq” in main body container with from field containing user’s id – some_jid (flush is used, if query is at the start of new chat communication from client application, i.e., the page with chat plugin is re-opened again), “to” contains site id.

   Client request: { "rid": 126, "sid": "some_sid", "body": [{"type": "iq", "from": "some_jid", "to": "some_siteid", "flush": "true"}] }

   Server response: { "rid": 126, "sid": "some_sid", "body": [{ "type": "result", "items": [{ "roomid": "room1356@site.org", "subject": "room 1156 topic", "newmsg": 3, "occupants": [{ "jid": "occupant1_jid", "name": "First User" }, { "jid" : "occupant2_jid", "name": "Second User" }] }] }

4. ***Query room for chat history*** – presence type of message to particular room with "to" setup for room id ("from" contains user's jid), response contains history of room, with "history" flag added for client. If user is not the occupant of the room, he/she is added to the room and chat history is send to the client too. All other participants of rooms are informed by the presence message with "from" field of added user and room id. If user is currently occupant of the room and "close" flag is added (i.e., "close": "true") to request, participant is removed from the chat room. All other participants (their sessions) are informed by presence message with "from" field of removed user, room id and "close" flag. If user was last participant, room is deleted.

   Client request (for standard history): {"sid":"some_sid", "rid": 126, "body": [{"type": "presence", "from":"some_jid ", "to":"room://1568@site.org"}]}

   Server response → JSON messages (history of current room – all chat messages and subject changes) are serialized in main body part of response, with additional "history" flag after "sid" and "rid") for easy identification of client that these JSON objects are history of chat room.

5. ***Send message to new chat*** – used in form "from" "to", where both field are users (from_jid is creator and to_jid is some other user contacted by creator) – it is actually the creation of new chat room, it means that new room object is created, presence message of both users to room with new roomid is send, and then chat message is send to both users (containing the first message from creator).

   Client request: {"sid":"some_sid", "rid": 126, "body": [{"type": "message", "from": "from_jid", "to":"to_jid", "body":"start message"}

   Server response: server prepares two presence responses and two message responses to newly created room with room_id, i.e., it can be schematically written as: message (from_jid, to_jid) -> presence (from_jid, room_id), presence (to_jid, room_id), message (from_jid, room_id), message (to_jid, room_id)

6. ***Send message to an existing chat*** – similar to previous one, but "to" flag contains room_id of the existing room.

   Client request: {"sid":"some_sid", "rid": 126, "body": [{"type": "message", "from": "from_jid", "to":"room_id", "body":"some message to existing room "}

   Server response: server prepares chat message response for every occupant within the current room, i.e., schematically it is: message (from_jid, room_id) -> [for each occupant of room] message (from_jid, room_id)

7. ***Add a new user to existing chat*** – it is presence type of message with "from" field containing current user's jid (from_jid) and "to" field containing room's id (room_id), for which from_jid is not the occupant of this chat room. Then it is necessary to update room and send necessary presence messages to other participants, and also provide the chat history to this new participant.

   Client request: {"sid":"some_sid", "rid": 126, "body": [{"type": "message", "from": "from_jid", "to":"room_id", "body":"some message to existing room "}

   Server response: server prepares presence message responses for every occupant within the current room and room history to new user (see number 4 for "chat_history" method), i.e., schematically it is: presence (from_jid, room_id) -> [for each occupant of room] presence (from_jid, room_id), chat_history (room_id , from_jid)

8. ***Change chat topic*** – subject of the topic of the particular room (with room_id) is changed by the user (with from_jid). If "subject" field is in the message, then its value is used as a new subject. Then message with subject is send to every occupant of this room, i.e., it is forwarded to sessions of other participants.

   Client request: {"sid":"some_sid", "rid": 126, "body": [{"type": "message", "from": "from_jid", "to":"room_id", "subject": "New subject of this chat"}] }

Server response: same message is forwarded to all participants (but through their sessions, i.e., with their session parameters)

9. ***Send site presence message*** – there is possibility to forward any (optional for client) field from one client through presence message (currently) to all users on server. It can be used to inform all clients about availability or status, “to” contains site_id. In example, status field is added to request. Then to everybody on chat server (with site_id) same message is forwarded.

   Client request: {"sid":"some_sid", "rid": 126, "body": [{"type": "presence", "from": "from_jid", "to":"site_id", "status": "Unavailable "}] }

   Server response: same message is forwarded to all participants (but trough their sessions, i.e., with their session parameters)

3.4. VISUALIZATION PACKAGE

The ***visualization*** package contains client-site JavaScript and resource files implementing the extensions for the data visualization. Besides of the client JavaScript components, it contains servlet, which implements Google Visualization Data Source API and proxy servlet, which provides secured access to the content stored in the Alfresco repository. The scheme presented in Figure 6 summarizes interactions between the OCOPOMO client-site components, Google Visualization framework and data source and proxy servlets.

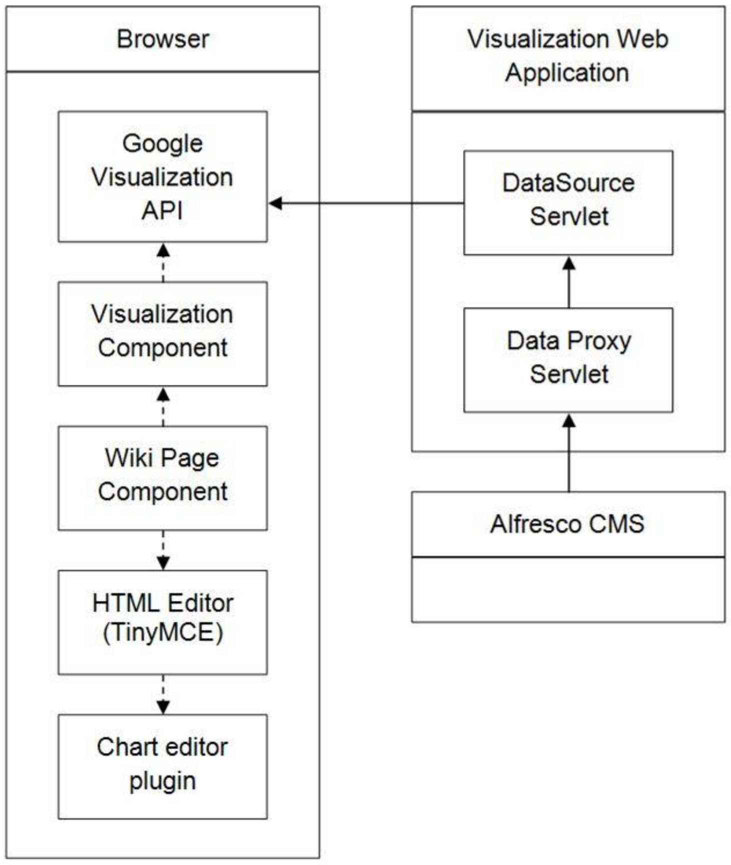


Figure 6: Inner structure of the *visualization* package.

The data visualization is integrated with the Wiki component, which is primarily used for the presentation of the simulation scenarios. It consists of the rendering components and editing components.

During the rendering, Wiki page component invokes Visualization component, which parses the Wiki page HTML content and extracts elements with the embedded charts. Elements contain JSON settings for the Google Visualization API. Google Visualization API fetches the visualized data from the Data Source servlet, which converts XML files with simulation traces into the Google Visualization data set encoded in JSON. XML simulation traces are stored in the Alfresco CMS repository, and are loaded from the repository through Data proxy servlet.

Inserting of charts is integrated into the online HTML editor used to enter HTML content of Wiki pages and other Alfresco components (Blog, HTML documents stored in the Document Library). Online editor is implemented using the TinyMCE client-side JavaScript framework, which has its own extension mechanism. Charts are integrated using the Chart editor plug-in, which configures additional command button for inserting new charts into the HTML content. Chart button opens the configuration dialog where the user can specify properties for the visualized data and generated charts. Chart plug-in code then encodes specified settings and inserts into the edited HTML code special markup with the embedded chart.

The chart markup has the following format:

```
<div class="chart">
    <span data="<JSON configuration object>"/>
</div>
```

JSON configuration object combines settings for data source, chart and additional controls associated with the chart. Properties for JSON objects are listed below in Table 7.

Table 7: Parameters used in JSON objects for data visualization.

JSON property	**Description**
dataSourceURL	String URL of the data source for visualized data. For visualization of simulation traces, it points to the Data Source servlet.
query	String query used to select visualized data (for example select * fetches all data).
wrapper	Object with the chart wrapper. Chart wrapper is the JSON object with the settings specified for chart, for example type of the chart (LineChar, BarChart, etc.), title, properties for axis and legend, etc.
controls	Array of objects with the control wrappers. Control wrapper is the JSON object with the settings specified for the controls associated with the chart. User can use these controls to interact with the chart and customize visualization (for example additionally filter visualized data).

For more information about the configuration options, please read the documentation about Google Visualization API on https://developers.google.com/chart/interactive/docs/reference.

4. CONCLUSION

This document provides the implementation and technology details of OCOPOMO Alfresco tools, i.e., OCOPOMO extensions of the Alfresco framework.

The suite of OCOPOMO Alfresco tools is available for download at:

http://ocopomo.ekf.tuke.sk/trac/ocopomoprj/wiki/Alfresco,

the mirror of installation packages and configuration bundles is available at

http://www.ocopomo.eu/workspace/wp-04-integration-of-components-1/d4.1-integrated-platform/integrated-ict-toolkit/alfresco

Installation instructions and usage guidelines are provided in the main text of D4.2 deliverable, as well as in the accompanying documents D4.2.-A *User Manual on Collaboration and Scenario Generation Tools* and D4.2-E: *User Manual on Simulation Output Visualisation and Traceability Tools*.