Spago Cms

Authors Luca Fiscato

INDEX

1 Versions History

Version/Release n° :	1.0	Data Version/Release :	23 May 2006
Description:	First release (english version)		

2 Document Goal

The goal of this document is to introduce the Spago Cms plugin and explain its usage.

3 References

You can find more information about Spago framework downloading the documentation from:

http://forge.objectweb.org/project/showfiles.php?group_id=195.

You can find more information about Content Management System (CMS) and the jsr 170 specification downloading the documentation from:

http://www.jcp.org/en/jsr/detail?id=170

4 Prerequisites

Spago Cms plugin is compatible with Spago 2.1.0.

5 Introduction

Spago CMS (Content Management System) is a plug-in for the Spago Framework that allows to exec operations on a Content Management System in a simple way. For instance, it can be used, by Spago developers, to perform versioning and restoring operations using only a few calls to its API.

Since it is based on JCR 170 specification (a new set of API, whose scope is to define an access model to the Content Management Systems) it can be used with different cms implementations compliant to the jsr 170 specification (Independence from the product).

A content repository is modelled as tree of nodes and each node can have properties, contents and other child nodes. Each node has a unique path, which is the path through the tree to get to it, and every time an developer want to perform an operation on a node, he must provide the path of the node. Te Spago Cms plugin divides nodes into two main category:

- Container Nodes: nodes that can contain other nodes, can have properties but can't have contents. Is the same concepts of file system folders.
- Content nodes: leaf nodes that can have properties, contents but can't contain other nodes. Is the same concepts of file system file.

The operations that a developer can do into the repository tree are:

- Set Operation: stores a node inside the repository tree in the position identified by a path. If the node doesn't exist it will be created, otherwise the plugin creates a new version of the node.
- Get Operation: reads a node from the repository and returns all the required node information (properties, childs, versions, content, ...)
- Delete Operation: deletes an entire node (with all its versions) or deletes a single version of the node.
- Restore Operation: restores a node version.
- Search Operation: exec a repository inquiry and returns the list of nodes that satisfy the search criteria. The query is expressed with Xpah language or JCQRL (a query language specified in JSR 170).

6 Installation

6.1 Distribution

The Spago cms module is composed by a core library, which contains base objects and definition of the operations, and from two other libraries (repository providers and execution service) which contain some default classes that can be replaced with your own implementation. This composition is due to the architecture of the plugin:

- The Spago cms module have to be independent from the cms product and so it must use only the api defined by the specification jsr 170. The repository root object defined by the jsr 170 api is called 'Repository' and to get it it's necessary to use some product specific api. In order to keep the independence of the plugin, the core library must not contain any reference to a specific cms product class. To achieve this requirements the classes, which retrieves the jcr 'Repository' objects, implement a core interface and are external to the Spago Cms core. In this manner the core part is able to create a new instance of these classes and to use them calling the interface methods.
- It's possible and likely to have a unique application that manages the repository and some other applications that want to use it. In this case it's necessary a communication layer between these applications. Spago Cms core define a generic interface which defines the possible requests towards a repository and use an external implementation to exec them. In this way is possible to

configure different implementation which can exec the request using inprocess api, Soap, Rmi, Webdav ....

6.2 How to install

To install Spago Cms you need to add the following libraries to the classpath of your application (for example, if your application is a web application, it's possible to put them into the WEB-INF/lib folder):

- spagocms-core-2.1.0.jar: contains the core classes like beans and interfaces
- spagocms-repositoryproviders-2.1.0.jar: contains classes useful to get a jcr 170 Repository object in different ways and for different products
- spagocms-inprocessexecutionservice-2.1.0.jar: contains the implementation for an inprocess operation execution assuming that that the application that uses the repository and the one that handles it are the same.

6.3 Integration with Spago

To integrate the plug-in with the Spago framework you need to add to the Spago master.xml file the link towards the configuration file of the Spago Cms (which contains initialization parameters and has a default name 'cms.xml') as below:

```
<MASTER>
      ...
      <CONFIGURATOR path="/WEB-INF/conf/cms.xml" />
      ...
</MASTER>
```

and to add the Spago Cms Initializer into the initializes.xml file as below:

```
<INITIALIZERS>
      ...
      <INITIALIZER class="it.eng.spago.cms.init.CMSInitializer"
                        config="CONTENTCONFIGURATION" />
      ...
</INITIALIZERS>
```

The first step allows the Spago Framework to read the cms configuration whereas the second one execs the inizialization of the cms environment during the application start-up. Obviusly after these changes you need to restart the application.

6.4 Configuration

The Spago Cms configuration contains some initialization parameters and is structured like below:

```
<CONTENTCONFIGURATION>

      <EXECUTIONSERVICE
            class="it.eng.spago.cms.exec.InProcessOperationExecutor"/>

      <CONTENTREPOSITORY
            class="it.eng.spago.cms.repositoryproviders.JackrabbitRepositoryProvider"
            name="jack">
        <PARAMETERS>
```

```
            <PARAMETER name="repository_path" value="/SpagoCmsExampleJackRep"/>
            <PARAMETER name="conf_file_path"
                       value="/Servers/tomcat-5.0.28/webapps/SpagoCms/repository.xml"/>
        </PARAMETERS>
        <CONNECTIONDATA user="system" password="" workspace="default" />


        <POOLCONFIGURATION maxActive="10" maxIdle="5" minIdle="3"
                           wait="2000"  initialConnection="5"/>
        <INITIALSTRUCTURE>
            <NODE path="/Documents" />
            <NODE path="/Documents/MyDocuments" />
            <NODE path="/Reports" />
            <NODE path="/Reports/MyReports" />
            <NODE path="/Tests" />
        </INITIALSTRUCTURE>
        <NAMESPACES>
            <SYSTEMNAMESPACE prefix="appSys"
                             uri="http://it.eng.spago.contentRepository.system" />
            <USERNAMESPACE prefix="appUsr"
                           uri="http://it.eng.spago.contentRepository.user" />
        </NAMESPACES>
      </CONTENTREPOSITORY>


      <DEFAULTREPOSITORY name="jack" />

</CONTENTCONFIGURATION>
```

The file is composed from three main different tags:

- <EXECUTIONSERVICE>: its attribute 'class' contains a complete class name of a class which implement the operation execution core Interface. For the current release the only possible class is it.eng.spago.cms.exec.InProcessOperationExecutor but you can write your own implementation and configure it.
- <CONTENTREPOSITORY>: contains the intial configuration for one repository and gives it a unique logical name.The cms configuration file can contains more than one repository configuration. The tag will be explain in more detail later.
- <DEFAULTREPOSITORY>: its attribute name defines which is the default repository within the ones defined. The value of the attribute must be equals to a logical name of one of the repository configured.

The tag <CONTENTREPOSITORY> configure a single repository and contains other tags and attributes explained below:

- Class attribute: the complete name of a class that returns a jsr 170 repository object. The possible class name are listed in the next paragraph.
- Name attribute: the logical name assigned to the repository configuration
- <PARAMETERS>: contains other tags useful for the repository provider class. It's content is not standard and changes based on the repository provider configured. In the next paragraph you find the possible configuration for the current release.
- <CONNECTIONDATA>: defines the deafult repository access credentials (user name, password and workspace name) used by the plugin to open repository session
- <POOLCONFIGURATION>: defines the configuration for repository session pool (Spago Cms manages a pool of repository session to reduce the overhead due to the session opening). The

parameter defines the total maximun number of sessions, the maximun and minum number of sessions not used, the time of session inactivity before close it and the number of initial sessions

- <INITIALSTRUCTURE>: defines the set of nodes to create automatically when the cms is initialized. Each node to create is defined by a NODE tag which contains its path. If the node already exists the plugin will ignore it. The order of the path is important because is not possible to insert a longer path after the insertion of less-long paths.
- <NAMESPACES>: defines the prefixs and uris of the jcr namespaces used by the plugin to store the jcr properties. The namespace to configure are for system properties and for used defined properties

The current release contains four different repository provider classes and each one of them as its own configuration parameters:

- it.eng.spago.cms.repositoryproviders.JackrabbitRepositoryProvider: builds a Jackrabbit repository. The tag <PARAMETERS> must contain two parametes: the first one defines the root folder of the repository where it stores its content whereas the second one defines the path of the jackrabbit configuration file (look at jackrabbit documentation for more detail)

```
<PARAMETER name="repository_path" value="/SpagoCmsExampleJackRep"/>
<PARAMETER name="conf_file_path"
           value="/tomcat/webapps/SpagoCms/repository.xml"/>
```

- it.eng.spago.cms.repositoryproviders.ExoRepositoryProvider: gets an eXo ECM repository. It must be used only if the application is installed on a eXo ECM server. The tag <PARAMETERS> must contain a parameter that defines the name of the portal application.

```
<CONTEXTNAME name="portal" />
```

- it.eng.spago.cms.repositoryproviders.JndiRepositoryProvider: gets the repository from a jndi context. Obviously the repository must have been configured as a jdni resource. The tag <PARAMETERS> must contain two parameters: the first one defines the name of the jndi initial context whereas the second one defines the jndi name of the resource

```
<JNDICONTEXTNAME name="java:comp/env" />
<JNDIOBJECTNAME name="cms/spagobicms" />
```

- it.eng.spago.cms.repositoryproviders.RmiRepositoryProvider: gets the repository from an rmi server. Obviously the repository must have been configured as a rmi resource. The tag <PARAMETERS> must contain a parameters that defines the name of the rmi resource.

```
<RMINAME name="//address:port/jackrabbit.repository"/>
```

7 Usage

Once configured, Spago Cms should be quite easy to use. To exec an operation on the repository you have to:

- Create an instace of the CmsManager object
- Create the bean of the operation that you want to perform into the repository
- Set the attributes of the operation bean
- Call the right method of the CmsManager passing it the operation bean

7.1 Operation Beans

Each repository operation can be configured with different parameters and attributes in order to behave in different manner. For example the get operation can be configured to get only the node content or only the list of node properties. For this reason Spago Cms contains a bean object for each operation that models it. To exec an operation you must create the relative bean, set its attribute and after pass it to a CmsManager object. The operation bean keep her configuration inside a SourceBean object.

Set Operation

Set Operation allows to create new nodes or a new version of an existing nodes. The internal configuration SourceBean has this structure:

```
<OPERATION name="">
      <SETOPERATION path="" type="(container/content)"
                    cancelOldProperties="(true/flase)">
            <CONTENT stream="" /> (optional)
            <PROPERTIES> (optional)
                  <PROPERTY name=""
                        type="STRING/DATE/LONG/DOUBLE/BOOLEAN/BINARY" >
                        <PROPERTYVALUE value=""/>
                        <PROPERTYVALUE value=""/>
                        ....
                  </PROPERTY>
                  ....
            </PROPERTIES>
      </SETOPERATION>
</OPERATION>
```

The method sof the bean allow to set these attributes:

- path: the path of the node to store
- type: the type of the node (container or content)
- cancelOldProperties: cancel the properties of the node in case it exists
- CONTENT.stream: Input stream of the content to store into the node
- List of properties: properties of the node to store. Each property must have a name, a type and value

Get Operation

Get Operation allows to retrive information about a node or about a specific version of the node. The internal configuration SourceBean has this structure:

```
<OPERATION name="">
      <GETOPERATION path="?"
                    version="?"
                    getVersions="(true/false)"
                    getChilds="(true/false)"
                    getContent="(true/false)"
                    getProperties="(true/false)" />
</OPERATION>
```

The methods of the bean allow to set these attributes:

- path: the path of node
- version: the version of the node. If the version is not defined the system will return information relative to the current node version
- getVersions: if true retrives information about the node versions
- getChilds: if true retrives information about the node childs
- getContent: if true retrives the content of the node
- getProperties: if true retrives information about node properties

Restore Operation

The operation allows to restore an old version of the node. The internal configuration SourceBean has this structure:

```
<OPERATION name="">
    <RESTOREOPERATION path="?" version="?" />
</OPERATION>
```

The methods of the bean allow to set these attributes:

- path: the path of the node
- version: the name of the version to restore

Delete Operation

The operation allows to delete an entire node or to delete only a version of the node. The internal configuration SourceBean has this structure:

```
<OPERATION name="">
      <DELETEOPERATION path="?" version="?" />
</OPERATION>
```

The methods of the bean allow to set these attributes:

- path: the path of the node
- version: the name of the version to delete. If it is not setted the plugin deletes the entire node, otherwise it deletes only the version of the node

Search Operation

The operation allows to inquiry the repository. The internal configuration SourceBean has this structure:

```
<OPERATION name="">
      <SEARCHOPERATION query="?" language="(xpath/sql)" />
</OPERATION>
```

The methods of the bean allow to set these attributes:

- query: the query to exec in order to extract data (the query should be written in xPath sintax or using the jsr 170 query sintax)
- language: the language used to express the query (xPath or jsr 170 query language)

7.2 Examples

Example 1: Stores a content node (leaf) taking it's content from a file system file. The node is stored at the path '/root/path' and contains a property with name 'nameProperty' and one value 'valueProperty'

```
CmsManager manager = new CmsManager();
FileInputStream fis = new FileInputStream(“/file”);
SetOperation setOp = new SetOperation();
      setOp.setEraseOldProperties(false);
      String path = “/root/path”
setOp.setPath(path);
      setOp.setType(SetOperation.TYPE_CONTENT);
      setOp.setContent(fis);
      List properties = new ArrayList();
String nameProp =”nameProperty”;
String valueProp = “valueProperty”;
String [] valuePropArr = { valueProp };
CmsProperty prop = new CmsProperty(nameProp, valuePropArr);
properties.add(prop);
setOp.setProperties(properties);
       manager.execSetOperation(setOp);
```

Example 2: Retrives information about a node having path '/root/path'. The information required are relative to versions and properties. taking it's content from a file system file. The result of the operation is a CmsNode object which contains all the information

```
CmsManager manager = new CmsManager();
GetOperation getOp = new GetOperation();
getOp.setPath(“/root/path”);
getOp.setRetriveChildsInformation("false");
getOp.setRetriveContentInformation("false");
getOp.setRetrivePropertiesInformation("true");
getOp.setRetriveVersionsInformation("true");
CmsNode cmsnode = manager.execGetOperation(getOp);
```

Example 3: Deletes the version named “1.0” of the node at path “/root/path”

```
CmsManager manager = new CmsManager();
DeleteOperation delOp = new DeleteOperation(“/root/path”, “1.0”);
manager.execDeleteOperation(delOp);
```

Example 4: Restores the version named “1.0” of the node at path “/root/path”

```
RestoreOperation resOp = new RestoreOperation(“/root/path”, “1.0”);
CmsManager manager = new CmsManager();
manager.execRestoreOperation(resOp);
```

Example 5: execs an inquiry on the repository using an xpath query. The operation return a List of CmsNode objects which model the cms node that satisfy the search criteria

```
CmsManager manager = new CmsManager();
SearchOperation searchOp = new SearchOperation();
searchOp.setLanguage("xpath");
searchOp.setQuery("//*[@appUsr:"+nameProp+"='"+valueProp+"']");
List nodes = manager.execSearchOperation(searchOp);
```