SPAGO CMS DOCUMENTATION

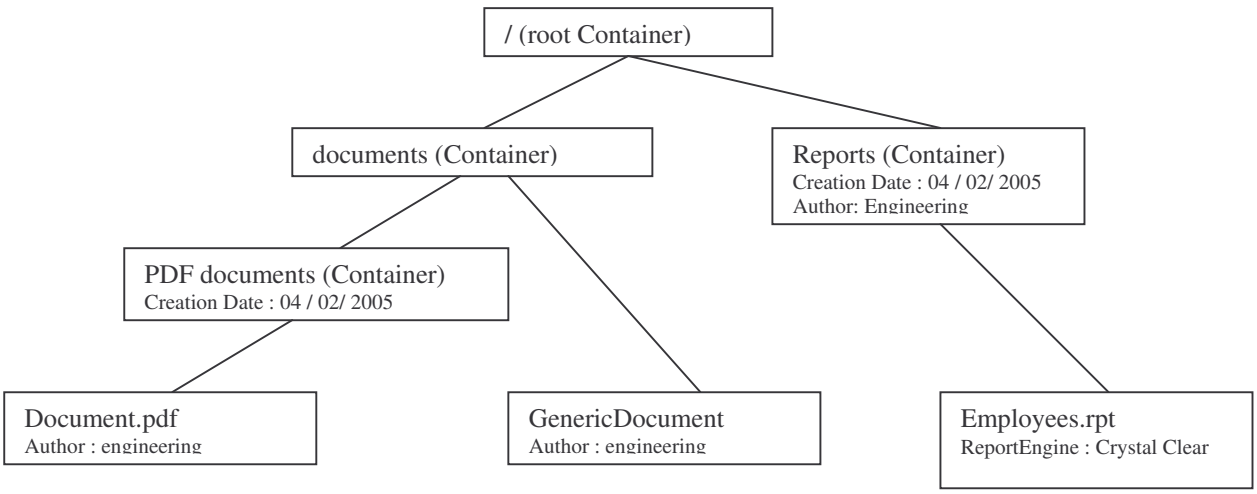
1. Introduction

Spago CMS (Content Management System) is a plug-in for the Spago Framework that allows to perform operations on a Content Management System. Spago CMS is based on JCR 170 specification (a new set of API, still in development, whose scope is to create an access model to the Content Management Systems) and so is independent from the particular system use, but obviously this system must join the JCR specification.
Spago CMS is a software layer, over JCR, that implements easy methods to manage contents: developers will be able to perform versioning operations in a simple way, using only a few calls to Spago CMS API.

A content Repository is a tree of nodes. Each node can be of two types:

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

An example of Spago CMS Repository Tree can be:



A node is identified by its path and repository can't contain different contents with the same path. All operations on the content repository are applied to a node so if one developer wants perform some operations on a node he must specify the node path. For example, if the developer wants read the properties and the contents of the GenericDocument in the above example, he must perform a GET operation on the path "/documents/GenericDocument".

The operations that a developer can perform on the repository tree are:

- SET OPERATION: this operation store a node inside the repository tree in the position identified by a path, if the node doesn't exist it will be created, otherwise will be created a new version.
- GET OPERATION: this operation reads a node and returns a Spago SourceBean containing all the required informations about the node (properties, childs, versions, content, ...)
- DELETE OPERATION: this operations permits to cancel a node with all its versions or to cancel a single version of the node.
- RESTORE OPERATION: this operations permits to restore an old version of a node. The most recent versions will not be erased but when the developer get the nodes he obtains the information about the version restored.
- SEARCH OPERATION: this operations permits to search the nodes of the tree with a query write in Xpah language or JCQRL, which is a query language specified in JSR 170.

The plug - in is distributed insede a jar file call SpagoCMS.jar, obviously to compile applications that use this module it's necessary to include the jar into the classpath. The only dependency is for the jcr-0.16.jar archive which contains all interface and class of the JSR specification, so also this file must be included in the classpath.

2. Integration with Spago

Spago CMS is a plug-in for Spago therefore it cannot be used separately. Integration happens through the Spago configuration files. The configuration files necessary to Spago CMS are the following:

- ContentConfigurations.xml: contains the configuration for all the CMS managed. (Mandatory)
- ContetPermission.xml: contains the users permissions for the operations on the CMS (Optional)
- ContentQuery.xml: contains the query expression used for search operations on the CMS (Optional)
- ContentOperation.xml: contains the operation description (an xml envelope) for operations on the CMS. (Optional)

The necessary files must be census inside the Spago master.xml file so that the Framework can read the contents. A example of the master.xml configuration file is:

```
<CONFIGURATOR path="/WEB-INF/conf/contentOperations.xml" />
<CONFIGURATOR path="/WEB-INF/conf/contentConfiguration.xml" />
<CONFIGURATOR path="/WEB-INF/conf/contentQueries.xml" />
<CONFIGURATOR path="/WEB-INF/conf/contentPermissions.xml" />
```

The start-up of the plug-in must happens with the start-up of the other components of Spago, so inside the configuration file "initializer.xml" must be present an xml envelop like this:

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

The Envelop has two parameters:

- Class: the name of the class that perform all operation start-up of the plug-in (creation of the repositories, creation of the connection pool, …);
- Config: the name of the xml configuration envelop. Note that this envelop is contained inside the "contentConfiguration.xml", so if you change the format of this file take care to control if this name is equal with the one in the file.

3. Start-Up Configuration

The xml start-up configuration is contained inside the file “contentConfiguration.xml”, this file has a fixed envelop <CONTENTCONFIGURATION> which contains one xml envelop for each repository managed, structured like this:

```
<CONTENTREPOSITORY class="it.eng.spago.CMS.factoryinstance.JackRabbitCMS"
                   name="jackRabbit">
   <INIT>
      …
   </INIT>
   <CONNECTIONDATA user="anonymous" password="" workspace="CMS" />
   <POOLCONFIGURATION maxActive="10" maxIdle="5" minIdle="3"
                          wait="2000"  initialConnection="5"/>
   <INITIALSTRUCTURE>
      <NODE path=”...” >
            <PROPERTY name=”...” value=”...” >
            ...
      </NODE>
   </INITIALSTRUCTURE>
</CONTENTREPOSITORY>
```

The attributes inside the envelop are:

- Class: class for management the specific CMS, this class perform the necessary operations for the specific CMS in order to create the JSR Repository and JSR Session objects, that are the base objects from which it’s possible to continue work with jcr api. At the writing time of this document the only CMS supports is Apache Jackrabbit, but in the future there will be other CMS, so other class will be write. The user can create a personal class for support a CMS but he must implements the “CMSIFace”.
- Name: the name assigned to the repository.

Inside there’s three mandatory xml envelopes:

- <init>: contains all parameters for the specific CMS used, so if the class that managed the CMS need some specific configuration parameters, they must be inserted here.
- <connectiondata>: contains the credential to use for the connection to the repository. JCR Specification define a method for create a session of work, but there isn’t a connection pool mechanism. The session will be open using password, userID and workspace. Spago CMS needs a connection pool so the plug-in has to open connections always with the same fixed credentials. Because of this mechanism has been necessary to develop a specific mechanism in order to manage the permissions on the repository, which is described later.
- <poolconfiguration>: Spago CMS manage a connection pool for all repositories configured, and needs some common configuration parameters of the pool tha can be specified here. This parameters are the maximun number of connection to create, the max and min number of connection unused to maintain, the time to wait for retrive a connection and the number of initial connections to create.
- <initialstructure>: into this envelop is possible to define an initial structure for the tree that will be created at start-up time. For all the nodes of the structure to create the developer must declare a <node> xml envelop, which contains the path of the node and zero or more properties to set, defined by the <property> xml envelop.

The configuration file contains also one xml envelop <defaultRepository> with an attribute name that identify the default repository of the application. The attribute name must contains one of the declared repository name, otherwise Spago CMS throws an exception at load time and doesn’t create the repository.

3.1. Apache Jackrabbit Configuration

At writing time of this document the only content management system support is Apache Jackrabbit, so this document will describe only the plug-in configuration for this CMS. Note that this document will not explain jackrabbit specific configuration because the reader can look at the apache site for this, instead the document explain the parameters necessary to integrate jackrabbit with Spago CMS.

For all repository defined in the configuration file there's an xml envelop that contains another envelop (<init>) with the specific parameter for the CMS. This envelop, for use Jackrabbit, must contains two attributes:

- RepositoryPath: jackrabbit stores information inside a file system folder, this envelop declare the path of this folder and a flag that indicates if the path is relative to the application directory or absolute to the file system, so inside there are two parameters:
  - Path: the path of the folder
  - Type: "relative" or "absolute", in first case the directory will be created starting from the folder of the web application, otherwise it's created starting from the root of the file system. Note that in the second case Spago CMS will throw an exception if the application hasn't the permission to write in the path.
- ConfigurationFile: jackrabbit needs a configuration file(look at jackrabbit site for information) and with this parameter is possible to declare where to find the file, so inside there are two parameters:
  - Path: the path of the file.
  - Type: "relative" or "absolute" , in the first case the path is interpreted starting from the web application folder otherwise from the root of the file system.

The use of jackrabbit arise the need of include into the classpath the library of jackrabbit and all libraries from which it depends. (The complete list can be found at the jackrabbit site).

4. Spago CMS Operations

All the operations on the repository needs some parameters like for example the path of the node. Different operation required different number of parameters, but they are always the same, so an operation ca be view like an xml envelop with attributes that mantain param values. This approach permits also an easy way to add new functionality to the operations.
The Spago CMS Operations are represented by an xml envelop with a fixed structure (one for each operations) that mantains all the parameter necessary and a name for identify it. This xml can declare attributes with a special char '?' that can be replace by a specific value, so the operation can be parametrized and reuse in various context, simply replacing the parameters values.

Spago CMS implements a method for retrive this operation descriptor as a SourceBean, so the developers have an easy way to manage an to perform parameters value substitution. All the Spago CMS API that perform repository operations take in input the SourceBean operation descriptor, so in the future will be easy to add new parameters.
All the xml operation descriptor are stored in the configuration file "contentOperation.xml" and each one is contained in a xml envelop like this:

```
<OPERATION name="name_of_the_operation">
      ....
</OPERATION>
```

This external envelop declares a name attribute, which must be unique in the file, in order to identify the operation. Inside there's the very xml description.
Developers can use this file for a quickly operation definition but this mechanism it's not mandatory. Developers can exec operations on the repository simply creating a SourceBean with the appropriate structure and filling up it with the necessary values.

4.1. Set Operation

Set Operation permits to create new nodes or a new version of an existing nodes. The xml structure of the operation is the following:

```
<SETOPERATION path="path_of_the_node" type="(container/content)"
              versionLabel=”label_for_a_new_version” >
      <CONTENT stream="stream_of_the_content" />
      <PROPERTIES mantainOldProperty=”(true/false)”>
           <PROPERTY name="property_name" value="property_value" />
      </PROPERTIES>
 </SETOPERATION>
```

- SETOPERATION.path = the path of the node. If the node doesn’t exist the node will be created otherwise Spago CMS creates a new version of the node. If the node doesn’t exists the plug-in create also all the ancestors not existing as container nodes. The attribute is mandatory.
- SETOPERATION.type = the type of the node to create, if container a new “folder” is create or versioned, otherwise a new content. The attribute is mandatory.
- CONTENT.stream = the stream of the content. This attribute has no sense with container nodes. The value of this attribute must be an InputStream but in the xml operation descriptor this attribute contains the special char ‘?’ for replace it with a parameter InputStream passing by the developers. The attribute is not mandatory.
- PROPERTIES = inside this envelop is possible to specify all the properties of the node. This xml envelop is not mandatory
- PROPERTIES.mantainOldProperty = the properties number for a set operation isn’t always the same because the developers can add new properties to the sourceBean or different operations can operate on the same node. If there are properties already set that doesn’t compare in the set operation the developers have to choice if maintain all the properties or to cancel those that are not present in the new set. The value of this attibute can be “true” or “false”, in the first case all the properties are maintained otherwise only the set properties will be mantained.
- PROPERTIES.PROPERTY.name = the name of the property to set
- PROPERTIES.PROPERTY.value = the value for the property. In this version of the plug-in the only type value is a String, but in the future there will be other types to set, like Date and Number.

4.2. Get Operation

Get Operation permits to retrive informations about a node. The xml structure is the following:

```
<GETOPERATION path="path_to_retrive" version="version_to_retrive"
              getVersions=”(true/false)” getChilds=”(true/false)”
              getProperties=”(true/false)” getContent=”(true/false)” />
</GETOPERATION>
```

- GETOPERATION.path = the path of the node to retrive. This param is mandatory.
- GETOPERATION.version = the version of the node to retrive. This attribute is not mandatory but obviusly if specified the informations returned are relative to the version.
- GETOPERATION.getVersions = if true the result will contain informations about the versions of the node.
- GETOPERATION.getChild = if true the result will contain informations about the childs of the node.
- GETOPERATION.getProperties = if true the result will contain informations about the properties of the node.
- GETOPERATION.getContent = if true the result will contain the stream of the content node.

The result of the operation is a SourceBean containing all request informations, the complete xml structure of this SourceBean is:

```
<NODE name="" path="" version="" uuid="" type="">
      <VERSIONS>
            <VERSION name="" />
            ...
      </VERSIONS>
      <PROPERTIES>
            <PROPERTY name="" value="" codetype="" type="" />
            ...
      </PROPERTIES>
      <CHILDS>
            <CHILD name="" path="" uuid="" type="" />
            ...
      </CHILDS>
      <CONTENT stream="" />
</NODE>
```

- NODE.name = name of the node (last element of the path)
- NODE.path = entire path of the node
- NODE.version = the current version of the node
- NODE.uuid = unique identifier for the node
- VERSIONS = contains all the version elements of the node
- VERSIONS.VERSION.name = name of the version
- PROPERTIES = contains all the property elements of the node
- PROPERTIES.PROPERTY.name = name of the property
- PROPERTIES.PROPERTY.value = value of the property.
- PROPERTIES.PROPERTY.codetype = numeric code of the property type. The JSR 170 define a set of value types for the properties and each one is associated to a code.
- PROPERTIES.PROPERTY.type = name of the property type
- CHILDS = contains all the child elements of the node
- CHILD.name = name of the child node.
- CHILD.path = path of the child node.
- CHILD.uuid = unique identifier of the child node.
- CHILD.type = type of the child node.
- CONTENT.stream = the input stream of the node content

If the method is called without a version parameter the informations returned are about the current version of the node, otherwise obviously the data returned is about the version specified. The current version of a node is the last one, if no restore operation was perform, or the last version restored. Versions, Properties, Childs and content envelops are present only if the correspondent flag of the operation descriptor is set to true.

4.3. Restore Operation

The operation permits to restore an old version of the node. The xml structure is:

```
<RESTOREOPERATION path="path_of_the_node_to_restore"
                  version="name_version_to_restore">
</RESTOREOPERATION>
```

- RESTOREOPERATION.path = the path of the node to restore, this param is mandatory.
- RESTOREOPERATION.version = the name of the version to restore (name version is set by the system), this params is mandatory.

The descriptor must contains the version parameter otherwise the system can't know which version to restore. If the version specified doesn't exist Spago CMS throws an exception.

4.4. Delete Operation

The operation permits to delete a node or one of its versions. The xml structure is:

```
<DELETEOPERATION path="path_of_the_node_to_delete"
                 version="name_version_to_delete" >
</DELETEOPERATION>
```

- DELETEOPERATION.path = the path of the node to delete. The node will be completely deleted only if no version name is specified otherwise only the version is erased. This param is mandatory.
- DELETEOPERATION.version = the name of the version to delete (name version is set by the system), this params is mandatory.

The descriptor must contain the version parameter otherwise the system can't know which version to delete. If the version specified doesn't exists SpagoCMS throws an exception.

4.5. Search Operation

The operation performs a search inside the repository using a query expression that can be write with two languages:

- Jcrql = a JSR 170 query sintax for search informations inside the repository (documentation about this language can be found inside JSR 170 specification).
- Xpath = a W3c query sintax for search informations iside an xml document and so also inside the repository that can be view like an xml document. (documentation about Xpath can be foud at W3c Site).

With these languages is possible to perform different searches like for example retrive all nodes contained inside a path, retrive all nodes with a property, and so on. The xml structure of the operation is:

```
<SEARCHOPERATION query="the_query_expression" language="(jcqrl/xpath)"
</SEARCHOPERATION>
```

- SEARCHOPERATION.query = the query expression to execute. This param is mandatory.
- SEARCHOPERATION.language = the query language sintax (jcqrl, xpath). The param is mandatory.

The result of the operation is a SourceBean containing all request information, the complete xml structure of this SourceBean is:

```
<RESULT>
      <NODES>
            <NODE name='' path='' uuid='' type='' />
            …
      </NODES>
</RESULT>
```

- NODES = the result of a query is a set of nodes, this element contains all the nodes found;
- NODE = Each node found has a name, path, uuid and type attribute , this element maintains all these attributes.
- NODE.name = name of the node found
- NODE.path = path of the node found.
- NODE.uuid = uuid (unique identifier) of the node found.
- NODE.type = type of the node found (container / content).

4.5.1. Query Params

In a real application often there's a set of query to execute that can be parametrized, so it's possible to reuse the same query with different values. To satisfy this need Spago CMS implements a mechanism for sobstitution of params value inside the query.
With the same mechanism view for operations it's possible to define the queries inside a configuration file and after to retrive them, using the assigned name, with the possibilty to sobstitute params value in place of a special sequence of chars. The result of this operation will be the string of the query with the value specified by the developers.
All the queries needed by the application can be declare in a configuration file call "contentQueries.xml", and for each query the file contains an xml envelop like this:

```
<QUERY name="name_of_the_query" value="value_of_the_query" />
```

The attribute "name" is an unique identifier of the query (inside the configuration file) and the attribute "value" is the query string. The String of the query can contain a special token "-par-" that can be replaced by the string value of a param specified from the developer. It's not possible to use the token "-par-" because this char can appear inside an xpath expression.
An example of Xpath query with param can be:

```
"//*[@author = '-par-']"
```

Spago CMS implements an Object "CMSQueries" which has methods for retrive the string of the query and to sostitute token "-par-" with the string value of parameters specified by the developers.
So, if a developer wants to perform a search operation with a parametrized query, he needs to retrive the query string with CMSQueries object and after he had to retrive the search operation descriptor passing the query like an attribute (using CMSOperation oject). This mechanism is a little bit complex so in future realeses probably the queries will be declared inside the search operation descriptor.

5. First Steps With Spago CMS

The steps necessary for execute an operation on the repository are:

- Get a connection to the repository.
- Get the SourceBean descriptor of the operation (created in the code or retrived from configuration file).
- Call the method to execute the operation passing the connection and the operation descriptor.

To obtain a connection to the repository developers must get the connection pool of the repository with the instruction:

```
CMSManager.getInstance()
```

The method getInstance can take a string param that is the name of the repository to connect (the name specified in the configuration file), if the method is call with no name it will return the connection pool of the default repository.
Now it's possible to call method "getConnection()" to retrive a connection from the repository pool, so the complete instruction to get a connection is:

```
CMSConnection connection = CMSManager.getInstance().getConnection();
```

To obtain the Operation descriptor developers can build it with the appropriate structure for the operation or they can retrive it from configuration file with object CMSOperation. This object have two methods that from the name of the operation they retrive the sourceBean descriptor. The difference is that and one of them replace the special attribute value '?' with a correspondent parameter value contained in a vector.
If the developer wants to retrive a descriptor with no parameters he can simply write:

```
SourceBean descriptor =
      CMSOperations.getOperationDescriptor("CREATE_NEW_FOLDER");
```

Otherwise if he wants to replace '?' with a sequence of parameter he must construct the vector of parameters before:

```
Vector params = new Vector();
params.add("parameter value");
SourceBean descriptor =
      CMSOperations.getOperationDescriptor("CREATE_NEW_FOLDER", params);
```

For a search operation there's another step: the developers must construct the query expression, so they can write the expression in the code or retrive the query from the configuration file, with the possibility to replace parameters. It's not possible for this version to write the query expression with param value directly inside the operation descriptor because Spago CMS is not able to correctly replace the parameters, so to perform a search operation, with query and operation descriptor defined in configuration files, the instructions are:

```
Vector paramsQuery = new Vector();
paramsQuery.add("parameter value");
String query = CMSQueries.getQuery("QUERY_NAME", paramsQuery);
Vector paramsOperation = new Vector();
paramsOperation.add("parameter_value");
paramsOperation.add(query);
SourceBean descriptor =
      CMSOperations.getOperationDescriptor("SEARCH_OPERATION", parOp);
```

To execute the operation developers must call the associated method passing the connection, the operation descriptor and a Boolean flag for close or not the connection after the execution. All the operation methods are declared in an interface so to call them developers must instantiate the Object "CMSSession" that implements this interface. Only two operations has a SourceBean return value (GET & SEARCH).
An example is:

```
CMSSession sessionCMS = new CMSSession();
sessionCMS.setObject(connection, descriptor, true);

SourceBean result = sessionCMS.getObject(connection, descriptor, true);
```

A complete example of a set and get operation:

Operations descriptor in the configuration file:

```
<OPERATION name="CREATE_NEW_FOLDER">
      <SETOPERATION path="?" type="container">
            <PROPERTIES>
                  <PROPERTY name="author" value="?" />
            </PROPERTIES>
      </SETOPERATION>
</OPERATION>

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

the code to create an new directory "/documents/myDocuments" and after retrive information about it is:

```
CMSConnection connection = CMSManager.getInstance().getConnection();
CMSSession sessionCMS = new CMSSession();

Vector params = new Vector();
params.add("/documents/myDocuments");
params.add("my");
SourceBean descriptor =
      CMSOperations.getOperationDescriptor("CREATE_NEW_FOLDER", params);
sessionCMS.setObject(connection, descriptor, true);

params = new Vector();
params.add("/documents/myDocuments");
SourceBean descriptor =
      CMSOperations.getOperationDescriptor("GET_OBJECT ", params);
SourceBean result = sessionCMS.getObject(connection, descriptor, true);
```

The result sourceBean contains all required information about the node.

6. Permission Management

Often in a real application developers must define policy permissions for operations on the repository, for example there's the need of:

- Allow operations only for a few user
- Deny write operations on a specific path
- Allow only read operations to a specific user.

JCR Specification define four types of operations that can be placed under permission control:

- Add node to a parent node
- Add property to a node
- Read a node
- Erase a node

Permission Management of Spago CMS is integrated with the authentication and authorization part of Spago. Spago define only an interface (IEngUserProfile) that defines some methods for retrive the rules of one user and all "functionalities" associated, which are logical operation. An application that use this mechanism must define an object that implements this interface which will be stored in the permanent session. In every moment Spago CMS can get this object from the session and retrive the functionalities associated with the current user.
This functionalities, that are logical operations, can be mapped to operations on the repository, so, for example, the logical Function "writeOnDocumentDirectory" can be mapped on permission to add node, write properties, read and cancel node on path "/Documents", otherwise the functionality "readDocuments" can be mapped only on permission to read nodes in path /Documents.

The developer must identify a set of logical operations on the repository and after he has to associate them to users, with the mechanism used for implementing the Spago authentication and authorization. All the logical functionalities defined have to be mapped to Repository permissions using the configuration file "contentPermission.xml."

This file contains one xml envelop <CONTENTREPOSITORY name='name_of_the_repository'> for each repository of the application that needs permission management, and inside this envelop there's the permissions configuration. An example part of this file is:

```
<CONTENTREPOSITORY name="jackRabbit">
      <FUNCTIONALITY name="CMSAllOperation">
            <ADDNODEPERMISSION>
                  <PATH value="/" subtree="true" />
```

```
        </ADDNODEPERMISSION>
        <SETPROPERTYPERMISSION>
              <PATH value="/" subtree="true" />
        </SETPROPERTYPERMISSION>
        <REMOVEITEMPERMISSION>
              <PATH value="/" subtree="true" />
        </REMOVEITEMPERMISSION>
        <READITEMPERMISSION>
              <PATH value="/" subtree="true" />
        </READITEMPERMISSION>
  </FUNCTIONALITY>
  <!--    other functionalities  -- >
```

Inside each repository xml node there's an envelop for each functionality mapping. Obviously the name of the functionality must match one defined with spago authentication and authorization mechanism. Each functionality nodes can contain one xml envelops for each of the four possible operations on the repository. At the end, each operations xml nodes can contains one or more "path" elements.
All this nested elements means that a user with a functionality can perform the inside operations on the paths specified. Obviously the paths are refered to the repository identify by the external envelop. The path element have also a "subtree" attribute that can contain a "true / false" value to indicate that the permission is not valid only for the single node path but also for all its Childs.

In the code of the application the developer can control user permission to execute an operation with a "PermissionVerifier" Object. To obtain an istance of this object the instruction is:

```
PermissionVerifier.getInstance();
```

The method return an instance of PermissionVerifier and can take in input a string param for identify the repository (the param is the name of the repository), if the method is call with no param Spago CMS returns the Verifier for the default repository.

The PermissionVerifier Object implements four methods to control permission for the four type of operations define by JSR 170. All this methods take in input the path to control and a string array of the functionalities assigned to the current user. The methods control the permission configuration file searching for a mapping from the operation on the path to at least one of the functionalities. The result is a Boolean that idicates if the user can perform the operation or not.

The methods of the object are:

```
PermissionVerifier.getInstance().canAddNode(String path, String[] functs);
PermissionVerifier.getInstance().canSetProperty(String path, String[] functs);
PermissionVerifier.getInstance().canRemove(String path, String[] functs);
PermissionVerifier.getInstance().canRead(String path, String[] functs);
```

This mechanism allows to define a fine grain control but it has a problem:

Spago IengUserProfile, in order to retrive the user set of functionalities, has a method that return a collection of functionalities, so one of them can be of various type. Spago CMS mapping mechanism require that the functionalities are of type String so it's possible to mapping with permission policy. The developers have to implement a method for transform a functionality (if it's not a string) in a String value that can be census in Spago CMS permission configuration file.