Basic Tutorial on CentraSite API for JAXR

Tutorial page for CentraSite API for JAXR with examples.

Basic Tutorial on CentraSite API for JAXR

This tutorial introduces the ***CentraSite API for JAXR*** . Therefore we will have a look on what JAXR standard is and how/where we can use it. After that I will explain the **CentraSite API for JAXR structure** for the further understanding of this tutorial. Subsequent to the introduction to the CentraSite API for JAXR structure we can already start with the first steps of implementing a client application and getting to know how to use the API.

In this document we will introduce the following concepts:

- Basic Tutorial on CentraSite API for JAXR
    - What is JAXR?
    - JAXR API structure
            - Lifecycle Manager (LCM)
            - Business Lifecycle Manager (BLM)
            - Query Manager (QM)
            - Business Query Manager (BQM)
    - Internationalization
    - Getting started
    - 1. The Connection Manager
        - 1.1 Creating a Connection using CentraSite API for JAXR
            - 1.1.1 Set System Properties
            - 1.1.2 Create a Connection
        - 1.2 Closing a Connection
    - 2. Implementing a handler class
        - 2.1 Creating a Service
            - 2.1.1 Create Lifecycle Manager
            - 2.1.2 Create Service
            - 2.1.3 Add a Slot to the Service
            - 2.1.4 Save the Service
        - 2.2 Searching for a Service
            - 2.2.1 Create a Business Query Manager (BQM)
            - 2.2.2 Search Service by Name
        - 2.3 Deleting a Service
        - 2.4 Adding further Attributes to a Service
            - 2.4.1 Add Classification Attribute to Service
            - 2.4.2 Add Association Attribute to Service
            - 2.4.3 Add File Attribute to Service
        - 2.5. Creating a User-Defined Taxonomy
            - 2.5.1 Create Taxonomy
            - 2.5.2 Create Concept

What is JAXR?

Java API for XML Registries (JAXR) defines a standard API with which Java applications can access and programmatically interact with various kinds of metadata registries. JAXR was developed under the Java Community Process as JSR 93.

JAXR provides a uniform, standardized Java API for accessing different kinds of XML-based metadata registries. Current implementations of JAXR support ebXML Registry version 2.0 and UDDI version 2.0. More such registries may be added in the future. JAXR provides an API that clients can call in order to interact with XML registries, and a service provider interface (SPI) for the registry providers so that they can plug in their registry implementations. JAXR insulates application code from the underlying registry mechanism. A JAXR-based client that browses or populates a registry does not have to be rewritten if the registry changes, for instance from UDDI to ebXML.(see Wikipedia)

CentraSite provides the CentraSite API for JAXR which we can use to create, modify and delete assets. Note that we do not use JAXR for accessing UDDI, but use UDDI with JAXR.

JAXR API structure

In this section we briefly describe the main structures of JAXR that are used in this tutorial. See the JAXR Specification for a comprehensive description of all important concepts.

Lifecycle Manager (LCM)

Provides support for creating, deleting and saving objects.

Business Lifecycle Manager (BLM)

Defines a simple business-level API for life cycle management of some important high-level interfaces in the information model.

Query Manager (QM)

Provides a common base class for all other specialized QueryManager sub-classes in the API.

Business Query Manager (BQM)

Provides a simple business-level API that provides the ability to query the most important high-level interfaces in the information model.

Internationalization

Becoming familiar with the concepts of internationalization is an important step in avoiding common errors. Internationalization deals with the problem that software often needs to be modified when it is ported to other cultures or languages. The internationalization concept provides an easy way of doing that without having to change the sourcecode of the program.
To make sure that everybody can read the string, we have to convert it into an international string, which can then be converted into any desired locale string. Also fundamental to internationalization is the concept of localization, which deals with translating the contents of the software into the local linguistic and cultural conditions. In Java there is an easy way to take care of this concept and thus make sure that the software can be used in any country. However, it is not always possible to change the system’s locale to the desired one. Nevertheless, we want to store strings that can be read in any environment, no matter which locale is used. Therefore we always store international strings when we create, for example, a new service.

In the following example we show how to convert a string into an international string in CentraSite API for JAXR. First create a Business Lifecycle Manager, which is described later in this tutorial.

```
InternationalString inString = blm.createInternationalString("inString");
LocalizedString loString = blm.createLocalizedString(Locale.getDefault(), inString);
```

Getting started

To access the CentraSite registry via Java, add all of the JAR files found in the redist subdirectory under your CentraSite home directory to the classpath. Make sure that all these JAR files are accessible in your project. On windows machines the path to the directory should resemble: `C:\SoftwareAG\CentraSite\redist`

This directory must also contain a *log4j* property file, which has the filename *log4j.properties*. This file must be located in the directory in which your Java class files are located. You can download this file from this link: log4j.properties

After completing these preparations, we can now start to implement a client application using the CentraSite API for JAXR.

1. The Connection Manager

In order to create a connection to CentraSite via JAXR, we create a `JAXRConnectionManager` class which does the job for us. This class implements the interface `CentraSiteJAXRConnector` which inherits three methods: one for opening a connection, one for closing a connection and an initialization method. The necessary interface is attached to this page. CentraSiteJAXRConnector.java

1.1 Creating a Connection using CentraSite API for JAXR

After creating the Java class `JAXRConnectionManager` and adding all unimplemented methods from the interface we can start to implement them.

1.1.1 Set System Properties

The first thing to do when trying to establish a connection to CentraSite using the CentraSite API for JAXR is to set the system property to define the class that implements the JAXR ConnectionFactory . You can do this by setting them in the initialization method of your class, as shown in *Listing 1.1*.

Listing 1.1: initialization method

```
public void init() {
    System.setProperty("javax.xml.registry.ConnectionFactoryClass",
         "com.centrasite.jaxr.ConnectionFactoryImpl");
}
```

JAXRConnectionManager.java

1.1.2 Create a Connection

In the next step we focus on connecting to `nl:CentraSite`. This is handled by the method `getCentraSiteConnection()` which returns a valid connection to the registry.

1. Declare a `ConnectionFactory`. See *m1 - m2* in *Listing 1.2*.
2. Supply the Query Manager URL to our connection. As the Life Cycle Manager URL in CentraSite is always the same as the Query manager URL, we don't have to declare it again. See *m2 - m3* in *Listing 1.2*.
   The variable `hostname` represents the host name or IP address and the port number of the server on which CentraSite is running. When running locally, this is typically `localhost` plus a port number such as `:53305`. Note that the actual port number may be different!
3. Create the connection and set the user credentials. You need a `PasswordAuthentication` object to which the username and the password are assigned. See *m3 - m4* in *Listing 1.2*.
4. Return the `Connection` object. See *m4* in *Listing 1.2*.

Listing 1.2: getConnection

```
public Connection getConnection(String hostname, String uname, String password)
throws JAXRException {
    // m1
    ConnectionFactory connFactory = ConnectionFactory.newInstance();
    // m2
    Properties p = new Properties();
    p.setProperty("javax.xml.registry.queryManagerURL", "http://"
                               + hostname + "/CentraSite/CentraSite");
    p.setProperty("com.centrasite.jaxr.BrowserBehaviour", "yes");

    connFactory.setProperties(p);
    // m3
    Connection con = connFactory.createConnection();

    HashSet<PasswordAuthentication> credentials = new
HashSet<PasswordAuthentication>(1);
    credentials.add(new PasswordAuthentication(uname, password.toCharArray()));

    con.setCredentials(credentials);
    // m4
    return con;
}// end: getCentraSiteConnection
```

JAXRConnectionManager.java
***NOTE**: We use the common JAXR Connection here. You could also use the `CentraSiteConnection`, but this is not necessary. It has some extensions for CentraSite that we do not need here.

1.2 Closing a Connection

To close the connection we use the method `closeConnection(...)`, which checks whether there is a valid connection and, if there is, closes it. See *Listing 2.1*.

Listing 1.2: closeConnection

```
public void closeConnection(Connection con) throws JAXRException {
    if (con == null) // check connection
 return;
    con.close(); // close the connection
}// end: closeConnection
```

JAXRConnectionManager.java

Make sure you close the connection at the end of your program and whenever a client failure occurs! Otherwise you may block other users from accessing resources on the CentraSite Server.

2. Implementing a handler class

For the other basic registry operations, we create a class `JAXRHandler` which implements the interface `CentraSiteJAXRHandler` and uses a `JAXRConnectionManager(see above)` for connecting to the registry. The necessary interface is also attached to this tutorial: CentraSiteJAXRHandler.java

To demonstrate how to call the methods and to do some testing, we also provide a test class which implements all tests for every method used in this tutorial: JAXRHandlerTest.java

2.1 Creating a Service

2.1.1 Create Lifecycle Manager

The next steps in this tutorial are to define a service and store it in the registry. For this purpose we need a Business Lifecycle Manager (BLM) which can do that for us. The BLM can easily be constructed with our already established connection. See *m0 - m1* in *Listing 2.1*.

2.1.2 Create Service

Having created the BLM, we can now create a new service. For this purpose we use the method `createService(...)`, which has two parameters for the name and the description of the service. The method should look like follows:

1. Declare a `Service` object. See *m1 - m2* in *Listing 2.1*.
2. Supply the service with the name and the description. The name and the description must be converted into international strings to take care of the Internationalization. See *m2 - m3* in *Listing 2.1*.
3. Create a `ServiceBinding` and supply it to the service, as well. See *m3 - m4* in *Listing 3.1*.

Listing 2.1: createService

```
public Service createService(String name, String desc) throws JAXRException {
    // m0
    RegistryService regService =  con.getRegistryService(); // get the registry
service from the connection
    BusinessLifeCycleManager blm = regService.getBusinessLifeCycleManager(); //
initialize the blm with the registry service
    // m1
    Service service = blm.createService(blm.createinternationalString(name));
    service.setDescription(blm.createinternationalString(desc));
    // m3
    ServiceBinding serviceBinding = blm.createServiceBinding(); // create service
binding with specification links

    service.addServiceBinding(serviceBinding);
    // m4
    return service;
}// end createService
```

JAXRHandler.java

NOTE: Names in CentraSite are not unique! You must decide whether you want them to be unique!

2.1.3 Add a Slot to the Service

Adding a slot to a service, which describes a property the service has, enables us to apply additional information to it. Therefore, every slot has three parameters, that have to be provided when adding a slot to a service.

1. SlotName: Every slot has a name which consists of two parts, a namespace and the name. The namespace is a URI embedded in brackets and the name stands right behind the closing bracket. Together they build a qualified name (QName) and look like follows: {NameSpace}name. This QName is represented as a string. CentraSite would also allow to use local names, i.e. without a namespace, but we do not recommend to do that to avoid mix-ups with CentraSite specific attributes.

2. SlotType: The type of the slot can either any XML schema type or a CentraSite extension type and is also represented by a string.

3. SlotValue: The value of the slot is represented as a string as well. It should fit to the slots type.

We use the method `addSlotToService(...)` as seen in *Listing 2.1.3a* to do the implementation.

Listing 2.1.3a: addSlot

```
public void addSlotToService(Service service, String slotName, String slotValue,
String slotType) throws JAXRException {
    Slot slot = blm.createSlot(slotName, slotValue, slotType);

    service.addSlot(slot);
}// end addSlotToService
```

JAXRHandler.java

In CentraSite there are two different types of properties (slots), *object-specific properties* and *type-specific properties*. To understand the differences between them, we will now go through a short example of how using them.

Object-Specific Properties: *Object-specific* slots can be added to any object instance. These slots have almost no restrictions and belong only to the objects they were added to. They have a loose typing and their names, values and types are not checked for inconsistencies. This means that you can configure this slot with any information.

Type-Specific Properties: *Type-specific* slots can be used in connection with custom asset types, defined by a user. Therefor, the user may define them as either required or optional properties. These slots have a strong typing and are checked for inconsistencies against the defined requirements. If the data within these slots does not pass the check of consistency, the object will be rejected when trying to save it to the registry. If the attributes are defined as "read only", the slots can only be added during the creation of the object. For a comprehensive overview of the creation of new types via the CentraSite GUI, see Creating a new type.

In `CentraSite API for JAXR` both types of slots are added with the same method. All checks for *type-specific* slots are first executed when saving the object to the registry. The example in *Listing 2.1.3b* shows the addition of *object-specific* slots to an object of type service. This object gets slots added for a "custom string", a "number" and a "date" when it was edited. Certainly, we use a string-type to represent the string, an integer for the number and a date-type to represent the date of editing the object.

Listing 2.1.3b: Example 1(object specific)

```
Service service = ;
addSlotToService(service, "custom string", "This is a custom string!",
CentraSiteSlotType.XS_STRING);

addSlotToService(service, "number", "1", CentraSiteSlotType.XS_INTEGER);

addSlotToService(service, "edited", "2011-08-04", CentraSiteSlotType.XS_DATE);

saveObject(service);
```

The example in *Listing 2.1.3c* shows again the addition of *object-specific* slots to an object of type service. This time we used the same slot names, but took either inconsistent datatypes or values for them. Nevertheless, these slots will be added to the object exactly with this configuration.

Listing 2.1.3c: Example 2(object specific)

```
Service service = ;
addSlotToService(service, "custom string", "This is a custom string!",
CentraSiteSlotType.XS_INTEGER);

addSlotToService(service, "number", "1", "xs:foo");

addSlotToService(service, "added", "08/04/2011", CentraSiteSlotType.XS_DATE);

saveObject(service);
```

You see that adding slots to objects is a pretty easy thing though you should be careful to add reasonable triples of name, type and value. However, this won't be checked when saving the object to the registry again. If you want to add slot to custom asset types with predefined slots, though, you have to add exactly the right name, type and value triples which were defined while creating the type. Moreover, if you declared one of the attributes as "read only", the slot can only be added during the creation of the object and before saving it to the registry. Otherwise it will also be rejected. Anyway, the adding of the slot works similar to object-specific slots.

Let's say we want to create an object of type "MyType" while "MyType" has a read only slot "number" of type Integer. For an overview how to create custom asset types see Custom Asset Types. See *Listing 2.1.3d*. If we would try to save the object without adding the slot "number" of type Integer the object would be rejected. Also trying to add another slot "number" of type "string" would be rejected by the registry.

Listing 2.1.3d: Example 2(type specific)

```
RegistryEntry re = createCustomObject("name", "description", "MyType");

Slot slot = blm.createSlot("number", "1", "CentraSiteTypes.XS_INTEGER");

re.addSlot(slot);

saveObject(re);
```

After having added slots to a service, we perhaps also want to access them again to change their values or types. Doing that is shown in *Listing 2.1.3c*. Therefore, we just have to get all the slots of the object and then iterate to the one we want to edit. Inside the slot object we can now edit all the values of it.

Listing 2.1.3c: access slot

```
Service service = ;
Collection slots = service.getSlots();

for(Slot s : slots) {
    s.getName() = "newName";
    s.getValue() = "newValue";
}
```

2.1.4 Save the Service

Finally we want to save the service into the registry. Accomplishing that is the functionality of the method `saveObject(...)` from *Listing 2.1.4*. This method can also store other registry objects to the registry. As you can see, we use the `Collections.singleton` pattern here as the method from the BLM only takes collections as its input.

Listing 2.1.4: save

```
public void saveObject(Object obj) {
    blm.saveObjects(Collections.singleton(obj));
}// end saveObject
```

JAXRHandler.java

When saving the service it is automatically associated with your organization.

2.2 Searching for a Service

2.2.1 Create a Business Query Manager (BQM)

To search for a service you need a *Business Query Manager* to query the CentraSite registry. The creation of this BQM is similar to the creation of the BLM. See *m0 - m1* in *Listing 2.4*. In our case we use a `CentraSiteQueryManager`, which has some additional functionality.

2.2.2 Search Service by Name

Using the BQM that we have created, we can now query the registry. To demonstrate the principle of searching a service, we focus on searching a service by its name. For this purpose we use the method `findServiceByName(...)`. *Listing 4.2* shows the steps necessary to create this method.

1. Create a service object in which the queried service can be stored. See *m1* in *listing 2.2.2*.
2. Specify the search criteria. In our case, we want an `EXACT_NAME_MATCH` for the `name` of the service. See *m1 - m2* in *Listing 2.2.2*. Again we use `Collections.singleton` as we only have one argument in each case.
3. Supply the criteria to the query and store the response in a new Collection. See *m2 - m3* in *Listing 2.2.2*.
4. Inside this new Collection '`serviceCol`' are now all services that satisfy our criteria. We can now iterate through the collection and get the services. In our case we assume that the names of our services are unique and hence there is only one service in the collection. See *m3 - m4* in *Listing 2.2.2*.

Listing 2.2.2: search service

```
public Service findServiceByName(String name) throws JAXRException {
    // m0
    RegistryService regService = con.getRegistryService();
    CentraSiteQueryManager cqm = (CentraSiteQueryManager)
regService.getBusinessQueryManager();
    // m1
    Collection qualifiers = Collections.singleton(FindQualifier.EXACT_NAME_MATCH);
    Collection namePatterns = Collections.singleton(name);
    // m2
    BulkResponse response = cqm.findServices(null, qualifiers, namePatterns, null,
null);
    Collection serviceCol = response.getCollection();
    // m3
    Iterator iter = serviceCol.iterator();
    Service service = null;
    if(iter.hasNext()) {
 service = (Service) iter.next();
    } else {
 System.out.println("No matches!");
    }
    // m4
    return service;
}// end findService
```

JAXRHandler.java

Remember that you are responsible for the uniqueness of the names! If there is more than one service with a given name, only the first one is returned. If the names in your registry are not unique, you could also return the whole collection and then iterate through it.

2.3 Deleting a Service

The delete method in this tutorial just takes the registry key of the service to find and delete it. See *Listing 2.3*. If the service is by any other service, an exception will be thrown. If this occurs, you should remove the association and then try to delete the service again.

Listing 2.3: delete service

```
public void deleteObject(Key key) throws JAXRException {
    BulkResponse response = lcManager.deleteObjects(Collections.singleton(key));
    System.out.println(response.getExceptions());
}// end deleteService
```

JAXRHandler.java

2.4 Adding further Attributes to a Service

After having created a basic service object in section 2.1 of this tutorial, we can now add some more attributes to it. We demonstrate how to create classification attributes, association or relationship attributes and file attributes. Since each of these attributes can have multiple values, we create two methods for the creation of each attribute: one that gives you the option of deleting all values that are currently assigned to the attribute, and one that retains all the current values by default.

2.4.1 Add Classification Attribute to Service

A classification attribute is a specific kind of attribute whose value is a classification according to a particular taxonomy. It is identified by an attribute-key that instance classifications will use as a reference in order to be recognized as classification attribute. (CentraSite API for JAXR Documentation) To create a classification attribute we use the method `createClassificationAttribute(...)`, which not only creates the attribute but also assigns it to the service.

1. Declare a `CentraSiteTypeDescription` and initialize it with the desired `type`. See *m1 - m2* in *Listing 2.4.1a*.
2. Declare a `CentraSiteClassificationAttribute` and initialize it with the attribute you desire. This requires iterating through all the classification attributes listed in the type description and comparing the `classificationAttributename`. See *m2 - m3* in Listing 2.4.1a_.

3. Define the classifying concept you want to use and add it to the list of attribute values. See *m3 - m4* in *Listing 2.4.1a*.
4. Assign the attribute and its values to the service. See *m4 - m5* in *Listing 2.4.1a*.

Listing 2.4.1a: add classification attribute

```
public CentraSiteRegistryObject
createClassificationAttribute(CentraSiteRegistryObject ro, String type, String
conceptKey, String classificationAttributeName,
                                                              Boolean keepexisting)
throws JAXRException{
    // m0
    RegistryService regService = con.getRegistryService();
    CentraSiteQueryManager cqm = (CentraSiteQueryManager)
regService.getBusinessQueryManager();
    // m1
    CentraSiteTypeDescription td = cqm.getTypeDescription(type);
    // m2
    CentraSiteClassificationAttribute attribute = null;
    for (CentraSiteClassificationAttribute ca : td.getClassificationAttributes()) {
// search for the attribute with the right name
 if (ca.getName().equals(classificationAttributeName)) {
            attribute = ca; // initialize the attribute
            break;
 }
    }
    // m3
    Concept classifingConcept = (Concept) cqm.getRegistryObject(conceptKey); //
define the concept to be used
    ArrayList<Object> values = new ArrayList<Object>();
    if (keepExisting){ // if there already is an attribute with that name => keep
all existing values
 values.addAll(ro.getClassificationValue(attribute));
    }
    values.add(classifingConcept); // add the new value "classifyingConcept" to the
list
    // m4
    ro.setClassificationValue(attribute ,values); // assign the attribute and its
values to the service
    // m5
    return ro;
}// end createClassificationAttribute
```

Usually, we want to retain all the existing values of an attribute. In this case we use the method shown in *Listing 2.4.1b*, which uses the method shown in *Listing 2.4.1a* but sets the `keepExisting` parameter to 'true' by default.

Listing 2.4.1b: add file attribute

```
public CentraSiteRegistryObject
createClassificationAttribute(CentraSiteRegistryObject ro, String type, String
conceptKey, String classificationAttributeName) throws JAXRException {
    return createClassificationAttribute(ro, type, conceptKey,
classificationAttributeName, true);
}// end createClassificationAttribute
```

JAXRHandler.java

2.4.2 Add Association Attribute to Service

A RelationShip attribute is a specific kind of attribute whose value is represented by an Association. It is identified by an attribute-key that instances will use as a reference in order to be recognized as RelationShip attribute. (CentraSite API for JAXR Documentation) Adding an association attribute to a service means that we want to build a relationship between this service and one or more other services, i.e. we want the service to be able to use the functionality of other services. Creation is quite similar to the creation of the classification attribute.

1. Declare a `CentraSiteTypeDescription` and initialize it with the desired `type`. See *m1 - m2* in *Listing 2.4.2a*.
2. Search for the relationship attribute with the desired name and initialize a `CentraSiteRelationshipAttribute` with it. See *m2 - m3* in *Listing 2.4.2a*.
3. Declare a collection of `RegistryObjects` and add all related services to it. See *m3 - m4* in *Listing 2.4.2a*.
4. Assign the attribute with all its values to the service. See *m4 - m5* in *Listing 2.4.2a*.

Listing 2.4.2a: add association attribute

```
public CentraSiteRegistryObject
createRelationshipAttribute(CentraSiteRegistryObject ro, String type, String
relationshipName,Collection<RegistryObject> targetROs,
                                                            Boolean keepExisting)
throws JAXRException{
    // m0
    RegistryService regService = con.getRegistryService();
    CentraSiteQueryManager cqm = (CentraSiteQueryManager)
regService.getBusinessQueryManager();
    // m1
    CentraSiteTypeDescription td = cqm.getTypeDescription(type);// type description
of desired type
    // m2
    CentraSiteRelationShipAttribute attribute = null;
    for (CentraSiteRelationShipAttribute ca : td.getRelationShipAttributes()) {//
search for the attribute with the right name
  if (ca.getName().equals(relationshipName)) {
      attribute = ca;// initialize the attribute
      break;
  }
    }
    // m3
    Collection<RegistryObject> tros = new ArrayList<RegistryObject>(); //
collection for all related services
    tros.addAll(targetROs);
    if (keepExisting) { // if service is already related to other services => keep
them
 tros.addAll(ro.getRelationShipValue(attribute));
    }
    // m4
    ro.setRelationShipValue(attribute,targetROs); // set the attribute and its
values
    // m5
    return ro;
}// end createRelationshipAttribute
```

Usually, we want to retain all the existing values of an attribute. In this case we use the method shown in *Listing 2.4.2b*, which uses the method from *Listing 2.4.2a* but sets the `keepExisting` parameter to ‘true’ by default.

Listing 2.4.2b: add association attribute

```
public CentraSiteRegistryObject
createRelationshipAttribute(CentraSiteRegistryObject ro, String type, String
relationshipName, Collection<RegistryObject> targetROs) {
    return createRelationshipAttribute(ro,type,relationshipName, targetROs, true);
}// end createRelationshipAttribute
```

JAXRHandler.java

2.4.3 Add File Attribute to Service

A file attribute is a specific kind of attribute whose value is represented by an ExternalLink. It is identified by an attribute-key that instances will use as a reference in order to be recognized as file attribute. (CentraSite API for JAXR Documentation)
The creation of a file attribute is quite similar to the creation of classification and relationship attributes. *Listing 2.8* shows the creation of such an attribute using the method `createFileAttribute(...)`.

Listing 2.4.3a: add file attribute

```
public CentraSiteRegistryObject createFileAttribute(CentraSiteRegistryObject ro,
String type, String fileName, String fileAttrName,
                                                    Boolean keepExisting) throws
JAXRException {
    // m0
    RegistryService regService = con.getRegistryService();
    CentraSiteQueryManager cqm = (CentraSiteQueryManager)
regService.getBusinessQueryManager();
    // m1
    CentraSiteTypeDescription td = cqm.getTypeDescription(type);
    // m2
    CentraSiteFileAttribute attribute = null;
    for (CentraSiteFileAttribute fa : td.getFileAttributes()) {
  if (fa.getName().equals(fileAttrName)) {
      attribute = fa;
      break;
  }
    }
    // m3
    Collection values = new ArrayList<RegistryObject>();
    if(keepExisting) {
       values.addAll(ro.getFileValue(attribute));
    }
    values.add(fileName);
    // m4
    ro.setFileValue(attribute, values);
    // m5
    return ro;
}// end createFileAttribute
```

Usually, we want to retain all the existing values of an attribute. In this case we use the method in *Listing 2.4.3b*, which uses the method from *Listing 2.4.3a* but sets the `keepExisting` parameter to 'true' by default.

Listing 2.4.3b: add file attribute

```
public CentraSiteRegistryObject createFileAttribute(CentraSiteRegistryObject ro,
String type, String fileName, String fileAttrName) {
    return createFileAttribute(ro, type, fileName, fileAttrName, true);
}// end createFileAttribute
```

JAXRHandler.java

2.5. Creating a User-Defined Taxonomy

The last section in this basic tutorial deals with the creation of a taxonomy. A taxonomy is a classification scheme for the classification of registry objects. It is divided into several concepts that represent a finer classification scheme for the objects belonging to this taxonomy. We show you how to build two methods, one for the creation of the taxonomy and another one for adding concepts to that taxonomy.

2.5.1 Create Taxonomy

The method `createTaxonomy()` creates us a basic taxonomy with a name and a description. The comments in the sourcecode should be self-explanatory. See *Listing 2.5.1*.

Listing 2.5.1: createTaxonomy

```
public ClassificationScheme createTaxonomy(String name, String desc) throws
JAXRException {
    BusinessLifeCycleManager blm = regService.getBusinessLifeCycleManager(); //
create BusinessLifeCycleManager
    InternationalString inName = blm.createInternationalString(name); // convert
name into InternationalString
    InternationalString inDesc = blm.createInternationalString(desc); // convert
description into InternationalString

    ClassificationScheme taxonomy = blm.createClassificationScheme(inName, inDesc);
// create the taxonomy

    return taxonomy;
}// end createTaxonomy
```

2.5.2 Create Concept

A concept can be assigned to any kind of registry object. For our purposes we use this feature simply for assigning concepts to taxonomies and other concepts. The first parameter in the method `createConcept(...)` represents the parent object to which the concept is assigned. The concept is assigned to the object right after creating it. However, we also return it, in case we want to do something else with it. See *Listing 2.10* for the implementation of the method.

Listing 2.5.2: createConcept

```
public Concept createConcept(RegistryObject obj, String name, String desc) throws
JAXRException {
    BusinessLifeCycleManager blm = regService.getBusinessLifeCycleManager(); //
create BusinessLifeCycleManager
    InternationalString inName = blm.createInternationalString(name);

    Concept concept = blm.createConcept(obj, inName, desc);

    return concept;
} // end createConcept
```