CentraSite

CentraSite API for JAXR

Version 9.5 SP1

November 2013

This document applies to CentraSite Version 9.5 SP1.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: IINM-DG-JAXR-95SP1-20140410

Table of Contents

Preface

This document explains how to use the CentraSite API for JAXR (Java Application Program Interface for eXtensible Markup Language Repositories). Here you can find reference information that describes the additions that we have made to the JAXR standard as published by the Java Community Process (JCP), and also sections that demonstrate some common use cases for the CentraSite API for JAXR and provide examples.

The reader of this document should be an experienced Java programmer, with knowledge of XML and the concepts of enterprise repositories.

Messages and Codes

- CentraSite API for JAXR Messages

1 Introduction

The CentraSite API for JAXR is based on the Java API for XML Registries (JAXR) standard. CentraSite supports JAXR capability level 1; in addition, it has some extensions that enable you to exploit specific functions of CentraSite.

Online API reference documentation for the JAXR API is available in the Java EE documentation. It is split into JAXR and JAXR infomodel.

A *Java Web Services Tutorial* with a section about JAXR is also available.

Software AG's extensions to the specification are described in the chapter **CentraSite API for JAXR Reference Information**. They can be briefly summarized as follows:

- You can create user-defined object types.

  CentraSite extends the JAXR object model by user-defined types, which may have triggers and operations attached. Correspondingly, the CentraSite JAXR-based extensions interface extends the JAXR query interface and allows you to search user-defined objects.

- XQuery access to the stored data.

  CentraSite allows a client to access the stored data directly using XQuery via the XQJ-based (XQuery API for Java) interface. For more information about XQJ, please refer to the specification (JSR 225), which you can download from this URL: **http://jcp.org/en/jsr/detail?id=225**.

I Using the CentraSite API for JAXR

2 Using the CentraSite API for JAXR

This part of the CentraSite documentation describes common use cases. It also includes a number of examples.

The following topics are discussed:

3 Creating and Closing a JAXR-based Connection

The following topics are discussed in this chapter:

Creating a JAXR-based Connection

To create a JAXR-based connection

1 Ensure that the *CLASSPATH* includes directories that contain the following files:

activation.jar
CentraSiteCommons.jar
CentraSiteDynLoader.jar
CentraSiteJAXR-API.jar
CentraSiteLCM.jar
CentraSiteLCM-api.jar
CentraSiteLCM-L10N.jar
CentraSitePolicy-API.jar
CentraSiteResourceAccess-API.jar
CentraSiteUtils.jar
CentraSiteUtils-L10N.jar
CentraSiteVMS.jar
CentraSiteVMS-L10N.jar
commons-codec.jar
commons-httpclient.jar
commons-lang.jar
commons-logging.jar
cstUtils.jar
groovy-all*.jar
inmUtil.jar
inmUtilConf.jar
jaxen.jar
jaxr-api.jar
jaxrpc.jar
jdom.jar
log4j.jar
PolicyLogBindings.jar
saaj.jar
saxpath.jar
script-api.jar
sin-common.jar
sin-misc.jar
sin-ssx.jar
sin-xmlserver.jar
stax-api.jar

TaminoAPI4J.jar
TaminoAPI4J-l10n.jar
uddiKeyConverter.jar
wstx-asl.jar
wvcm.jar
xmlbeans.jar
xqjapi.jar
xqj-ino-api.jar

Note: You can find these files in the CentraSite *redist* folder.

Note: If you have activated an e-mail policy, the *CLASSPATH* must additionally include the file *mail.jar*, which you can find in the *rts/bin* folder.

2 Start your client program with the following parameter:

```
-Djavax.xml.registry.ConnectionFactoryClass=com.centrasite.jaxr.ConnectionFactoryImpl
```

Or:

Set this property during program startup:

```
System.setProperty("javax.xml.registry.ConnectionFactoryClass",
                                                    ↩
"com.centrasite.jaxr.ConnectionFactoryImpl");
```

3 Create a factory:

```
ConnectionFactory connFactory = ConnectionFactory.newInstance();
```

4 Supply the `queryManagerURL` to the connection:

```
Properties p = new Properties();
p.setProperty("javax.xml.registry.queryManagerURL",
                                                  ↩
"http://localhost:53307/CentraSite/CentraSite");
```

Note: In CentraSite, the `lifeCycleManagerURL` is always the same as the `queryManagerURL`, hence it need not be specified.

Note: The port number, in the example above specified as 53307, may need to be changed to suit your local configuration.

5 Set the `BrowserBehaviour` option:

```
p.setProperty("com.centrasite.jaxr.BrowserBehaviour", "yes");
connFactory.setProperties(p);
```

Enabling BrowserBehaviour mode is the preferred way of creating a JAXR-based connection. This is beneficial for several reasons. The BrowserBehaviour mode uses a less strict locking pattern, and this can result in an increased number of parallel read and update operations. For example in the CentraSite Control UI, while one user is looking at some asset, another user can update the same asset in parallel. In the same scenario without BrowserBehaviour, the update would fail as the necessary lock cannot not be granted.

Moreover, with BrowserBehaviour mode, the assets cached on the client side are refreshed more often. After an asset is read, it will be refreshed in the cache if it is returned as the result of a subsequent query with a newer timestamp.

6 Create the connection and set the user credentials. The `setCredentials()` method expects a `Set` containing a `java.net.PasswordAuthentication` object.

```
Connection connection = connFactory.createConnection();

HashSet credentials = new HashSet(1);
credentials.add(new PasswordAuthentication("userid", "password".toCharArray()));


connection.setCredentials(credentials);
```

7 With the connection given, the other environment objects can easily be constructed:

```
RegistryService regService = connection.getRegistryService();
BusinessLifeCycleManager lcManager = regService.getBusinessLifeCycleManager();
BusinessQueryManager     bqManager = regService.getBusinessQueryManager();
```

Closing a JAXR-based Connection

A JAXR-based connection uses some resources in the CentraSite XML Server. We therefore strongly recommend making sure that a connection is closed in case of a JAXR-based client failure. Otherwise the resources are released only after a non-activity timeout; this might hinder parallel users.

To close a JAXR-based connection

- ```
  connection.close();
  ```

  where `connection` is as specified in the example above.
  ```

4 Defining a Service

A service is provided by an organization. It should have a name and a description, and the details are specified by service bindings which are further detailed by specification links. The following code snippet, which assumes that the providing organization is known, shows how to create a new service:

```
Organization providingOrganization = ...;

Service service = m_lcManager.createService("service name");
service.setProvidingOrganization(providingOrganization);
InternationalString description =
                        lcManager.createInternationalString("service description");
service.setDescription(description);

ServiceBinding serviceBinding = ...;  // create service binding with specification ↩
links

service.addServiceBinding(serviceBinding);

ArrayList serviceList = new ArrayList();
serviceList.add(service);
lcManager.saveServices(serviceList);  // save service and related modified objects
```

5 A Service that Uses Another Service

If a service calls another service, this should be modeled with the pre-defined "Uses" association.

```
Service callingService = ...;
Service calledService = ...;

// find the "Uses" concept
ClassificationScheme associationType
      = ↩
bqManager.findClassificationSchemeByName(Collections.singleton(FindQualifier.EXACT_NAME_MATCH), ↩
"AssociationType");
Concept usesConcept
     = bqManager.findConceptByPath("/" + associationType.getKey().getId() + "/Uses");

// create association of type "Uses"
Association usesAssociation = lcManager.createAssociation(calledService, usesConcept);

callingService.addAssociation(usesAssociation);  // callingService is now the
                                                 // source object of the association

ArrayList associationList = new ArrayList();
associationList.add(usesAssociation);
lcManager.saveAssociations(associationList, false);  // save association and
                                                     // related modified objects
```

6 A Service with Additional Information

Each JAXR-based object instance may be supplied with arbitrary additional information. JAXR uses the “slot” mechanism to provide this kind of extensibility.

Note: JAXR allows arbitrary strings as slot names. The CentraSite implementation stores a slot by creating an XML element whose tag name is the slot name. Consequently, a slot name should be a valid XML QName. If a QName has a non-null URI, the lexical representation of the slotname is the URI enclosed in curly braces, followed by the local-name, for example `{myUri}mySlotname`.

The following code snippet shows how to add a slot to a service object:

```
Service service = ...;

Slot slot = lcManager.createSlot("{myUri}mySlotName", "slotValue", null);
service.addSlot(slot);

ArrayList serviceList = new ArrayList();
serviceList.add(service);
lcManager.saveServices(serviceList);
```

7 Pre-Defined Classification Schemes (Taxonomies)

The CentraSite registry comes with several pre-defined classification schemes:

- All the classification schemes that are defined in the JAXR standard.
- A classification scheme for the products using CentraSite. Thus, each registry object can be classified with its product. This makes it easy to find all registry objects originating from a particular product.

  The name of this classification scheme is "Products", and its member concepts are CentraSite itself and products that use CentraSite.
- A classification scheme for database management systems: This can be used to classify data sources by the type of the database management system they represent.

  The name of this classification scheme is "Databases", and its member concepts are:

  Adabas;
  Tamino;
  DB2;
  Enabler;
  MSSQL;
  Oracle.
- A classification scheme for content types: This can be used to classify external links with their content type/MIME type.

  The name of this classification scheme is "ContentType". This is an external classification scheme.
- A classification scheme for the types of objects in the CentraSite repository: This can be used to classify external links with their repository object type.

  The name of this classification scheme is "RepositoryObjectType" , and its member concepts are:

  BPEL;
  BPELObject;

CustomComponent;
Documentation;
DTD;
E-mailEvent;
Emerger;
FileEvent;
HTML;
Icon;
JAR;
JMSEvent;
Layout;
Ontology;
Payload;
ProjectFolder;
ReportDefinition;
ScheduledTask;
Sequence;
SOAP;
Template;
TypeIcon;
WSDD;
WSDL;
XML;
XSD;
XSLT;

- Some external classification schemes used for UDDI mapping:

  ClassificationGroup;
  Object;
  UseType;
  uddi-org:wsdl:categorization:protocol;
  uddi-org:wsdl:categorization:transport;
  uddi-org:wsdl:portTypeReference;
  uddi-org:wsdl:types;
  uddi-org:xml:localName;
  uddi-org:xml:namespace.

8 Impact Analysis

Impact analysis means finding dependencies between objects: which object depends on which other object, or vice-versa: if one object is modified or deleted, which other objects are affected? For example, if a web service interface changes, which callers must be adapted?

In JAXR, dependencies between objects are established via associations. There are a variety of predefined association types, and moreover a JAXR-based client can create its own association types. Although the names of the association types - for example `HasChild` or `HasMember` - suggest a certain semantic, JAXR itself does not imply any semantics with the association types. CentraSite supports the following conventions for associations.

If there is a dependency between two objects, each of which can exist on its own, then this dependency should be expressed by a `Uses` association. Example: one web service calls another web service. Remember that JAXR-based associations are directed: the association's source object should be the caller/user (in general, the object that depends on another object), and the association's target is the called/used object.

If there is an object `C` that cannot exist without another object `P`, then `C` should have a `HasParent` association to `P`. Example: A table object cannot exist without a database object, hence there is a `HasParent` association from each table to the corresponding database.

The reason for preferring `HasParent` over the “inverse” `HasChild` association is as follows: CentraSite tries to maintain referential integrity; this means, among other things, that is not possible to delete an object that is still the target of an association. Hence associations should be directed in such a way that an object cannot be deleted if someone else still depends on it: an object should not be deleted if it still has children, or if it is still in use by someone else.

II CentraSite API for JAXR Reference Information

9 CentraSite API for JAXR Reference Information

This part of the CentraSite documentation describes the differences between the JAXR standard and our API. In particular, the CentraSite-specific extensions to the JAXR standard are explained here.

The following topics are discussed:

10 User-Defined Objects

In addition to the pre-defined object types such as organizations, services and associations, CentraSite allows you to define your own object types. Once such a type has been created using the CentraSite Control GUI, a corresponding concept exists in the `ObjectType` classification scheme.

To create an instance of a user-defined object type

1 Create a `RegistryEntry` object.

2 Classify it with the type concept.

The following code example assumes that a user-defined type "{User-Uri}UserType" exists:

```
RegistryEntry userTypeObject
           = (RegistryEntry)lcManager.createObject(LifeCycleManager.REGISTRY_ENTRY);

// find the "{User-Uri}UserType" concept
ClassificationScheme objectType
                    = bqManager.findClassificationSchemeByName(null, "ObjectType");
Concept userTypeConcept
                        = bqManager.findConceptByPath("/" + ↩
objectType.getKey().getId()
                                                              + "/{User-Uri}UserType");

// create classification
Classification userTypeClassification
                                  = lcManager.createClassification(userTypeConcept);
userTypeObject.addClassification(userTypeClassification);

/*
 * from now on the userTypeObject is of type "UserType", and
 * userTypeObject.getObjectType() will return a concept equal to userTypeConcept
 */
```

```
// save object
ArrayList objectList = new ArrayList();
objectList.add(userTypeObject);
lcManager.saveObjects(objectList);
```

11 Direct XQuery Access to the Stored Data

A CentraSite JAXR client can call XQJ (XQuery API for Java technology) functionality directly in order to access the registry data. JAXR itself also uses XQJ to access the registry. For more information about XQJ, please refer to the specification (JSR 225), which you can download from this URL: **http://jcp.org/en/jsr/detail?id=225**.

The CentraSite `CentraSiteConnection` maintains an `XQConnection` object which it uses for its own purposes as well as for direct client access. The client can get this object as follows, assuming he already has a JAXR-based connection:

```
Connection jaxrCon = ...;
XQConnection xqjCon = ((CentraSiteConnection)jaxrCon).getXQConnection();
```

As both the client and JAXR use the same XQJ connection, the following restrictions apply (assuming the client uses JAXR and XQJ in parallel):

- The client must not call any JAXR-based `save...` method if he has an open transaction, because JAXR performs the `save...` methods as one atomic operation based on an XQJ transaction.
- The client should never close the XQJ connection. Instead, he must close the JAXR-based connection. This action cleans up anything else.

12 Unique Keys

This implementation does not support client supplied keys. The method `RegistryObject.setKey()` throws an `UnsupportedCapabilityException`. CentraSite rejects client-supplied keys.

13 Simultaneous Database Access and Locking

The CentraSite implementation stores all RegistryObjects in a common repository, which is a database. If multiple JAXR-based clients (or, to be more precise, multiple JAXR-based connections) are active simultaneously, it is possible that they might read and update the data in the common database concurrently.

Multiple clients that update a RegistryObject must be synchronized in order to prevent lost updates. Usually, this is handled by the underlying database's locking mechanism. However, since it is likely that many JAXR-based clients would be browsing or searching the repository and only a few JAXR-based clients would be modifying data, the CentraSite implementation has been optimized to allow maximum concurrent access. In particular, if one or more JAXR-based clients are reading a RegistryObject, another JAXR-based client may update it concurrently.

For example, if a user has opened CentraSite Control to look for a particular object and then keeps his or her UI open for a protracted period – maybe even for several days – this should not prevent other users from updating that object.

Locks for read access are therefore relatively permissive, but of course it must be ensured that two JAXR-based clients cannot modify the same object at the same time. This is achieved as follows:

When a JAXR-based client starts to modify a RegistryObject, JAXR acquires an exclusive lock for this object from the database management system. This prevents any other client from updating the same object at the same time. When the JAXR-based client saves the modified object, the lock is released as a side-effect of calling `LifeCyclemanager.saveObjects()`. Alternatively, if the JAXR-based client decides to discard the changes, it should release the lock by calling `CentraSiteConnection.rollback()`.

With this locking behavior, there are two principal scenarios when two JAXR-based clients attempt to modify the same object at the same time. Bear in mind that in order to modify an object, the JAXR-based client always has to read it first, then modify the Java instance, then call `saveObjects()` in order to write the modified object back to the database.

Scenario A

JAXR-based Client A	JAXR-based Client B
1. Read a RegistryObject.	
	2. Read the same RegistryObject.
3. Start to modify the object. This automatically locks the object.	
	4. Start to modify the object. The attempt to lock the object fails and a `LockNotAvailableException` is thrown.

As long as client "A" holds the exclusive lock for the object, client "B" is unable to modify it.

Scenario B

JAXR-based Client A	JAXR-based Client B
1. Read a RegistryObject.	
	2. Read the same RegistryObject.
3. Start to modify the object. This automatically locks the object.	
4. Save the object. This releases the lock.	
	5. Start to modify the object. The attempt to lock the object fails and an `ObjectOutdatedException` is thrown.

In scenario "B", client "A" has finished making its changes and has released the lock, so the lock is now available for acquisition by another client, for example client "B". However, client B's local copy of the object does not reflect the current database status of the object, which has been modified in the meantime by client A. If client B were allowed to save object, client A's modifications would be overwritten.

To avoid this, each RegistryObject has a last-modification date. When a lock is acquired, the API checks whether the last-modification date of the object in the database is the same as the last-modification date of the client's local copy of the object. If the dates are not the same, an `ObjectOutdatedException` is thrown. This ensures that updates are not lost and that all modifications are based on the latest state of the object.

Immediately before the `ObjectOutdatedException` is thrown, the API cleans up its internal structures. When the client catches the exception, it should release all references to the RegistryObject and then re-read it. This should return the latest copy of the object from the database; the client can now continue to make the necessary modifications to this clean copy.

14 The Caller

The caller identifies himself by issuing `Connection.setCredentials()`. The corresponding `User` object is retrieved from the registry using the name given in the credentials. If the user record does not yet exist, it is created. This new user object is not added to any organization.

Here, the user name is the name attribute as inherited from the `RegistryObject` interface. It should not be confused with the user's `PersonName`.

The caller must be known before a connection can be used. In other words, `setCredentials()` is required, otherwise a security error occurs.

Note: The user name must be unique in the registry.

15 Semantics of Remove Operations

There are several methods that allow an object to be removed from its parent. Depending on the kind of object, the remove operation has different effects:

- **Associations, Classifications, External Identifiers, Service Bindings, Specification Links**

  If such an object is removed from its parent and the parent is then saved, the object is automatically deleted, since it cannot exist as a standalone object. These objects can be removed using the following methods:

  RegistryObject.removeClassification()
  RegistryObject.setClassifications()
  RegistryObject.removeAssociation()
  RegistryObject.setAssociations()
  RegistryObject.removeExternalIdentifier()
  RegistryObject.setExternalIdentifiers()
  ServiceBinding.removeSpecificationLink()
  Service.removeServiceBinding()

- **Other Objects**

  Other objects are delinked from their parents during the remove operation. They continue to exist as separate objects. If the parent object is saved, the removed objects are also automatically saved.

  The remove operations for these objects are:

  ClassificationScheme.removeChildConcept()
  Concept.removeChildConcept()
  Organization.removeUser()
  Organization.removeService()
  Organization.removeChildOrganization()
  RegistryObject.removeExternalLink()

RegistryObject.setExternalLinks()
RegistryPackage.removeRegistryObject()

16 Delete Operation

Deleting an object means deleting it from the persistent store. Optionally, the delete operation can be called with an `objectType` parameter, which is one of the pre-defined `LifeCycleManager` interface names. If this parameter is specified, only objects of that type are accepted for delete. The interface names shown in the following list are allowed for a deletion; all others are rejected with an `InvalidRequestException`.

```
LifeCycleManager.ASSOCIATION
LifeCycleManager.CLASSIFICATION
LifeCycleManager.CLASSIFICATION_SCHEME
LifeCycleManager.CONCEPT
LifeCycleManager.EXTERNAL_IDENTIFIER
LifeCycleManager.EXTERNAL_LINK
LifeCycleManager.ORGANIZATION
LifeCycleManager.REGISTRY_ENTRY
LifeCycleManager.REGISTRY_PACKAGE
LifeCycleManager.SERVICE
LifeCycleManager.SERVICE_BINDING
LifeCycleManager.SPECIFICATION_LINK
LifeCycleManager.USER
```

Objects have relationships to each other: some relationships prohibit object deletion, while other relationships are automatically cleaned up during deletion. The following sections describe for each object type how it is treated during deletion.

RegistryObject

In general, an attempt to delete a registry object is rejected if:

- it is a new object, i.e., it has not yet been saved, or
- it is the target of an association.

Deleting a registry object has the following side-effects:

1. Remove the object from all its packages; update the packages.
2. Delink the object from all its external links; update the external links.
3. Delete all associations whose source object is the object to be deleted.
4. Delete all classifications whose classified object is the object to be deleted.
5. Delete all external identifiers whose registry object is the object to be deleted.

Association

1. Remove the association from its source object.
2. Update the source object. This automatically deletes the association.

AuditableEvent

It is not possible to delete an auditable event explicitly.

Classification

1. Remove the classification from its classified object.
2. Update the classified object. This automatically deletes the classification.

ClassificationScheme

1. Reject deletion if there are child concepts; otherwise:
2. Delete the classification scheme.

Concept

1. Reject deletion if there are child concepts; otherwise:
2. Remove the concept from its parent object.
3. Update the parent object.
4. Delete the concept.

ExternalIdentifier

1. Remove the external identifier from its registry object.
2. Update the registry object. This automatically deletes the external identifier.

ExternalLink

1. Reject deletion if there are linked objects; otherwise:
2. Delete the external link.

Organization

1. Reject deletion if there are child organizations, services, or users; otherwise:
2. Remove the organization from its parent organization.
3. Update the parent organization.
4. Delete the organization.

RegistryEntry

1. Delete the registry entry.

RegistryPackage

1. Reject deletion if there are member objects; otherwise:
2. Delete the registry package.

Service

1. Remove the service from its organization.
2. Update the organization.
3. Delete all service bindings whose service is the service to be deleted.
4. Delete the service.

ServiceBinding

1. Remove the service binding from its service.
2. Delete all specification links whose service binding is the service binding to be deleted.
3. Update the service. This automatically deletes the service binding.

SpecificationLink

1. Remove the specification link from its service binding.
2. Update the service binding's enclosing service. This automatically deletes the specification link.

User

1. Remove the user from its organization.
2. Update the organization.
3. Delete the user.

17 Unsupported Methods

The following methods are not supported and throw an `UnsupportedCapabilityException` exception:

- `RegistryService.getDeclarativeQueryManager()`
- `RegistryService.makeRegistrySpecificRequest()`

18 Unsupported FindQualifiers

The following FindQualifiers are not supported:

- COMBINE_CLASSIFICATIONS
- SERVICE_SUBSET
- SOUNDEX

19 Using Wildcards

The wildcard character, which is the percent (“%”) character, represents zero or more characters. Thus, for example, the search string “ABC%DEF” finds all strings that begin with “ABC” and end with “DEF”, with any number of characters in between. The search string “ABC%DEF%” finds all strings that begin with “ABC” and include “DEF” anywhere else. If you do not include a wildcard character in the search string, the search assumes that there is a wildcard character at the end of the search string, unless the find qualifier "EXACT_NAME_MATCH" is specified. Thus, for example, if you specify “ABC” as the search string, the search in fact looks for and finds strings that match the pattern “ABC%”, i.e. all strings that begin with the characters “ABC”.

20 Using Namespaces

Some names, for example type names and slot names, comprise a namespace and a name. When programming a JAXR-based client, these names must be represented in the following format:

```
{namespace}name
```

In other words, the namespace is enclosed in curly braces and is used as a prefix for the name.

Strings in this format are used in the following methods:

for objectType in CentraSiteQueryManager.findObjects()
for typeName in CentraSiteQueryManager.getTypeDescription()
for name in LifeCycleManager.createSlot()
for slotName in ExtensibleObject.getSlot()

21 The Method createSlot

The method `createSlot` in the interface `LifeCycleManager` takes 3 parameters; its signatures are as follows:

```
Slot createSlot (String name, String value, String slotType)
Slot createSlot (String name, Collection values, String slotType)
```

The CentraSite implementation accepts any value of type `String`, or a null reference, for the third parameter, `slotType`. This parameter is stored with the slot, but it is not interpreted in any way. Note, however, that the JAXR standard does not indicate how this parameter should be interpreted; it might, for example, be interpreted as indicating the data type of the slot in some future implementation. We recommend specifying the `slotType` as an "xs:string".

22 Caching Considerations

This chapter describes the following aspects of caching behavior as it affects the API:

JAXR-based Caching Strategy

Objects that are retrieved from the registry by means of the CentraSite API for JAXR are stored in a cache by the JAXR-based connection. All objects stored in the cache are inspected from time to time by the Java garbage collector, which may delete them if there are no references to them from the application.

Any object reference that results from a call to `getRegistryObject()`, `getRegistryObjects()` or any of the `find` methods is, if possible, resolved from the cache. If an application already holds a reference to an object that resulted from any of these calls, the reference will also be in the cache, and the call will return the same Java reference.

There are situations, however, where the cache is cleared completely. This occurs, for example, after executing `saveObjects` or `deleteObjects`. Any Java reference that is retrieved after the cache is cleared will be different from a reference that was retrieved before the cache is cleared.

Note: This does not affect data integrity, since objects read cannot be concurrently updated.

Caching in CentraSite User Interfaces

The CentraSite user interfaces, i.e. Control and Eclipse, browse JAXR-based data; this means that they make use of the JAXR-based caching mechanism, but they do not block concurrent updates. Control and Eclipse users should be aware that, in general, the data display does not immediately reflect changes that another user may make.

Note: This does not affect data integrity in the sense that outdated data may be the source of any updates.

You can see the current data at any time by choosing the **Refresh** button.

Dynamically Loaded JAR Files

The system locally caches dynamically-loaded JAR files. You should be aware that the date and time of the cached files are compared with the date and time of the library files whenever a new connection is created; the JAR files in the cache are refreshed if they are found to be out of date. This could mean that processing continues with a newer version of a JAR file after a connection has been created.

Note also that problems may arise if a custom security manager has been implemented, because the connection to the database will be refused.

Cache Location

The system uses the following strategy to determine the location of the cache store:

- If the system property `com.softwareag.centrasite.dynloader.cache-dir` is defined, then its value is used as the location of the cache store.
- Otherwise, the location of the cache store is derived from:
  1. A directory whose name is taken from the system property `java.io.tmpdir`;
  2. A sub-directory whose name is constructed from the string "CentraSite", a package name, and the string "Jars".