SCIDIP-ES

SCIENCE DATA INFRASTRUCTURE FOR PRESERVATION - EARTH SCIENCE

Registry Framework

Developer's Overview

Current Version: M30 Release – Based on SVN ver. 6787
Last Updated: 3 March 2014
Original Author: Simon Berriman

Table of Contents

Document Control

Date	Version	Description	Author
21/09/2013	M24 Draft – Based on SVN ver. 5970	Initial version, based on 0.0.2-SNAPSHOT	Simon Berriman
28/10/2013	M24 Release – Based on SVN ver. 6350	Updated for release version 1.1.2	Simon Berriman
03/03/2014	M30 Release – Based on SVN ver. 6787	Updated for release version 2.0.0	Simon Berriman

Introduction

The Registry Framework is designed to make life easier when interacting with one or more Registries within the SCIDIP-ES model. The Framework makes working with RepInfo Labels and Manifests completely agnostic of where they originated and indeed whether or not the RIN crosses over multiple Registries. Only when storing new or amended RILs or Manifests is it necessary to specify which of the discovered registries to store it in.

Including in an Application

The Framework is a library in the form of a Java JAR, which is to be included in the classpath of an implementing application. This can be done either by downloading and compiling the source from SVN, or by using the latest version from Nexus, which is the recommended approach. This document covers the current release version which is version 2.0.0. The development snapshot in SVN is not covered here.

To include the Framework the following should be added to the application's `pom.xml` file (correct at time of writing):

```
        <dependency>
            <groupId>eu.scidipes.common</groupId>
            <artifactId>scidipes-framework</artifactId>
            <version>2.0.0</version>
        </dependency>
```

Interacting with the Framework

All of the Registry Framework's core functions are fronted by a single class of static methods. This design was chosen in an attempt to ensure 'under the hood' singletons are properly controlled, and that any other framework which is being used by the implementing application (e.g. Spring) does not interfere in any way.

The entry point class for an application to interact with the framework is `eu.scidipes.common.framework.FrameworkWrapper`. The methods in this class return only core Java classes or SCIDIP-ES Model interfaces. There are a few convenience methods, but for the most part the methods are designed to be included in an application's own control logic – i.e. looped over where necessary. The Framework does make use of several caches (implemented with Ehcache), which cuts down significantly on network traffic. Thus no specific requirement is placed on the implementing application to cache returned objects itself.

In addition to interaction methods, there are two other method in the FrameworkWrapper which are worthy of special note. `shutdown()` must be called as an application is exiting to ensure the proper disposal of the caches. Failure to do so could result in unexpected behaviour when the application is next started. It could also prevent the JVM from exiting, as the cache manager is not in a daemon thread. Once the `shutdown()` method has been called, the Framework will throw cache exceptions if any further interactions are attempted. The `restart()` method reinitialises the Framework, including the cache manager if it is necessary to use it again, however with good application design this should not be necessary.

The following sections give a little more detail on the expectations and outputs from the various static methods. For the most up-to-date information, see the current JavaDoc, which us located at http://registry2.scidip-es.eu/javadoc/framework/

allocateNewPID()
Allocates a newly generated CPID. At present this simply returns a new type 4 (pseudo randomly generated) UUID. See http://www.ietf.org/rfc/rfc4122.txt for details on UUIDs.

allocateNewPIDWithPrefix(String)
Allocates a newly generated CPID using allocateNewPID() with the passed prefix prepended to it. This could be of use if, for example, it is wished to use a UUID as a qualified URN, or if an application wishes to group its created IDs together in some way.

getAllPredefinedCategories()
Returns a set of all the known predefined, non-reserved categories, as loaded at startup from the APA switchboard. Categories are used by RILs and Manifests to provide some indication as to the intended use or purpose of a given piece of RI. A 'non-reserved' category is any category available for general use, and not reserved for a specific programmatic function or purpose. An example of reserved categories are those for the RILs and Manifests themselves which are used by the framework to identify known registry response types.

getAllRILsOnRegistry(Registry)
Queries the passed Registry for **all** the most recent versions of the Rep Info Labels it contains, and returns a set of identifiers. Caution should be exercised in using this method, as the resultant set has the potential to get very large from a well populated registry.

getAllRegistryObjectsFor(Set<CurationPersistentIdentifier>)
This is a convenience method which takes a set of independent CPIDs, and returns them as keys in a Map, with each one either mapping to the most recent versions of the Registry object (RIL or Manifest) which it represents, or if it is of an unknown type, mapped to null.

getCPID(String)
One of the core methods. Given a CPID in the form of a string object of a unique identifier, this will return an object conforming to the model `CurationPersistent-Identifier` interface. Nothing is fetched remotely at that point in time, however the local cache is checked to return an existing instance if that CPID had been requested already.

getLocationHolding(Identifier<D>)
This is a convenience method. It will return the Registry from which the passed identifier was retrieved, unless it is new in which case a null is returned. This method simply calls `getLastKnownGoodLocation()` on the passed identifier itself and type checks the result to ensure it is a Registry.

getKnownRegistries()
On startup, the Framework uses Java's Service Provider Framework (SPF) to discover all the included implementations of the `DigitalObjectLocation` model interface. The returned set is checked and guaranteed to only contain those `DigitalObjectLocation`s which are also Registries – i.e. which conform to the model `Registry` interface. No check is made as to whether or not the Registry is enabled or available.

getEnabledRegistries()
Similar to getKnownRegistries(), this method returns a set of all the discovered Registries, which are also marked as 'enabled'. The discovered Registry objects carry this as a boolean property.

getManifest(Identifier<D>)
One of the core methods. The identifier passed is expected to be a CPID, such as that returned by getCPID(String). It should be reasonably expected that the CPID is one that represents a Manifest before calling this method, as if the Framework is unsuccessful in finding a Manifest for the passed identifier, an RIException will be thrown. Assuming success, an object conforming to the model `Manifest` interface is returned. The Manifest returned will be the most recent version found on any Registry.

getManifest(Identifier<D>, int)
One of the core methods. The identifier passed is expected to be a CPID, such as that returned by getCPID(String). It should be reasonably expected that the CPID is one that represents a Manifest before calling this method, as if the Framework is unsuccessful in finding a Manifest for the passed identifier, an RIException will be thrown. Assuming success, an object conforming to the model `Manifest` interface is returned. The Manifest returned will be that of the version requested. An RIException will be thrown if the requested version can not be found.

getRepInfoLabel(Identifier<D>)
One of the core methods. The identifier passed is currently expected to be a CPID, such as that returned by getCPID(String). It should be reasonably expected that the CPID is one that represents a RepInfo Label before calling this method, as if the Framework is unsuccessful in finding a RepInfoLabel for the passed identifier, an RIException will be thrown. Assuming success, an object conforming to the model `RepInfoLabel` interface is returned. The RepInfoLabel returned will be the most recent version found on any Registry.

getRepInfoLabel(Identifier<D>, int)
One of the core methods. The identifier passed is currently expected to be a CPID, such as that returned by getCPID(String). It should be reasonably expected that the CPID is one that represents a RepInfo Label before calling this method, as if the Framework is unsuccessful in finding a RepInfoLabel for the passed identifier, an RIException will be thrown. Assuming success, an object conforming to the model `RepInfoLabel` interface is returned. The RepInfoLabel returned will be that of the version requested. An RIException will be thrown if the requested version can not be found.

getRepInfoCategoryByName(String)
This will return a RepInfoCategory model object, corresponding to the textual name passed. If the passed name represents one of the predefined categories, that fixed category object will be returned. For any other category name, a new model object is returned; this is to allow for new categories outside the context of the predefined list to be created as used. See 'Appendix A – Predefined RepInfo Categories' for the current list of predefined categories. As all categories are cached, so this method should be the primary mechanism for obtaining an instance of a RI Category.

getRegistryTypeFor(Identifier<D>)
Determines whether a CPID represents a RIL or a Manifest. The returned RegistryObjectType object is an enum from the model, representing one of these two types of objects a Registry can return. If the passed CPID does not correspond to either a Manifest of RIL (e.g. it simply does not exist) then null is returned.

getRepInfoLabelFromManifestID(Identifier<D>)
This is a convenience method. It is a simple daisy-chain of getManifest(Identifier<D>) followed by getRepInfoLabel(Identifier<D>), allowing a straight RIL to RIL chain to be followed (of the most recent versions) where the application does not care for the intermediate Manifest.

searchForManifestsByRILCPID(CurationPersistentIdentifier)
Given the CPID of a RIL, this method will return a set of Manifest CPIDs from across all available registries which are of that RILs type; i.e. Manifests which name that RIL in their 'rilcpid' field.

searchForRILsReferencingManifest(CurationPersistentIdentifier)
Given the CPID of a Manifest, this method will return a set of RIL CPIDs from across all available registries which name the passed Manifest in one of their RI lists.

searchForRILsMatching(String)
Given a textual keyword or phrase as a string, this method will return a set of RIL CPIDs where the RIL contains that string. No heuristics are used by the Framework, however individual registry implementations are free to perform the search in the best manner they see fit. As a minimum it should be expected for the keyword to be searched for as part of either a RILs '`displayname`' or '`description`' fields.

searchForManifestsInCategory(RepInfoCategory)
Given a category object, this method will return a set of Manifest CPIDs from across all available registries which advertise themselves as being descriptive of the passed category.

searchForRILsInCategory(RepInfoCategory)
Given a category object, this method will return a set of RIL CPIDs from across all available registries where a Manifest which uses this RIL has advertised itself as being descriptive of the passed category. *NB. RILs themselves do not describe categories.*

storeManifest(Manifest, Registry)
One of the core methods. This is for storing a new or amended Manifest object in the specified Registry. The CPID is expected to be embedded in the Manifest object already, so for new Manifests an application should first call allocateNewPID() whilst constructing the object. Versioning is handled automatically, so no attempt should be made to set or alter the version number property. It is also advisable to check that the Registry is writeable. Any failure to complete the storage operation will result in an RIException being thrown.

storeRepInfoLabel(RepInfoLabel, Registry)
One of the core methods. This is for storing a new or amended RepInfoLabel object in the specified Registry. The CPID is expected to be embedded in the RepInfoLabel object already, so for new RepInfoLabels an application should first call allocateNewPID() whilst constructing the object. Versioning is handled automatically, so no attempt should be made to set or alter the version number property. It is also advisable to check that the Registry is writeable. Any failure to complete the storage operation will result in an RIException being thrown.

The following methods are not concerned with direct Registry object manipulation, but are for obtaining Java classes from remote repositories where specified by a manifest. The exact operation of this mechanism is yet to be fully worked out, so the current implementation should be seen as alpha quality code.

getJavaConstructorFrom(Manifest)
Used to create a new instance of a Java class, as pointed to by the 'location' field in the passed Manifest. The resultant class must implement the `JavaConstructor` model interface. This method comes with some caveats and will throw an `IllegalStateException` if it is unsuccessful for any reason. (The exact reason for a failure will be logged.) It makes use of a custom class loader, which can load bytecode from any stream, and give it a custom package and classname. It is currently limited to loading single classes without dependencies and which are uncompressed, i.e not in a JAR file. This is something which will be addressed as priorities dictate.

getInformationObject(JavaConstructor, DigitalObject)
This method attempts to apply the passed DigitalObject to the given class implementing the `JavaConstructor` model interface, as returned by getJavaConstructorFrom(Manifest). The result *should* be a new object implementing the `DigitalInformationObject` model interface.

This is something of a demonstration method, as the approach taken to exception handling is very broad-brush, and may not be acceptable within the implementing application. For the resultant object to be accepted it must implement the `DigitalInformationObject` model interface directly, and not through inheritance. This is due to the fact that verification must be done by interface name string comparison on account of the class loader hierarchy that will be in place at the point of execution.

Registry Authentication and Authorisation

The Registry interface describes a digital object location which holds only Manifest and RepInfoLabel objects as all or part of a Rep Info Network (RIN). Such locations may require authentication to access. The superinterface of Registry, DigitalObjectLocation, defines three accessors whose purpose is explained below.

isEnabled()

By default all discovered Registries are enabled. This is an override flag that will take any given Registry object out of use, and is achieved by manually calling `setEnabled(false)`. Once disabled, a Registry can be found again in the set returned by FrameworkWrapper's `getKnownRegistries()`.

isAvailable()

Until determined otherwise, this will be false. A Registry is considered to the 'available' is it is enabled, and has been tested to have **at least read** access.

isWritable()

Until determined otherwise, this will be false. A Registry is considered to the 'writable' is it is enabled, and has been tested to have **read *and* write** access.

authoriseForReadOnly() and authoriseForReadWrite()

A Registry can be tested to read and/or write access by calling one of these two methods. Calling `authoriseForReadWrite()` implies a call to `authoriseForReadOnly()`, meaning that there is no write access to a Registry without there also being read access. These methods are assertions, in that they do not return a result, but will throw an RIException in the event of the assertion failing for any reason. This can be tested for the subclass `RIHTTPException`, which will contain further details if the cause of the assertion's failure was server side – e.g. as a result of rejected credentials.

setCredentialsProvider(CredentialsProvider) and HTTPAuthCredentialsProvider

One Registry can have at most one set of credentials at any one time. These are provided through a model CredentialsProvider. As the current implementation of the Framework only understands HTTP as a communications protocol to Registries, the only CredentialsProvider implementation included in `HTTPAuthCredentialsProvider`. These are immutable holders for username and password pairs, which must be set into a Registry prior to attempting to call one of the 'authorise' methods above. If none is set, then it is assumed that the Registry is 'open' and any attempt to 'authorise' will therefore be ignored.

Code Recipes

To find out whether a CPID, passed as a String, is a RIL or Manifest:

```
final CurationPersistentIdentifier cpid = FrameworkWrapper.getCPID(cpidString);

try {
     final RegistryObjectType rrt = FrameworkWrapper.getRegistryTypeFor(cpid);

     switch (rrt) {
     case RIL:
            // Do RIL things
     case MANIFEST:
            // Do Manifest things
     }

} catch (final RIException rie) {
     // Handle unknown type condition
}
```

If you know you have the identifier for a RepInfoLabel as a String:

```
RepInfoLabel ril = null;
try {
    CurationPersistentIdentifier cpid = FrameworkWrapper.getCPID(cpidString);
    ril = FrameworkWrapper.getRepInfoLabel(cpid);

} catch (RIException e) {
    LOG.warn(e);
}
```

To retrieve the each Manifest which is using a given RepInfoLabel:

```
final Set<Identifier<DigitalObjectLocation>> manifestIDs =
         FrameworkWrapper.searchForManifestsByRILCPID(ril.getCpid());

for (final Identifier<DigitalObjectLocation> manifestID : manifestIDs) {

    final Manifest manifest = FrameworkWrapper.getManifest(manifestID);

    // Do something with the Manifest
}
```

Creating and storing a new Manifest:

```
// Create a new object conforming to the Manifest interface (fictional)
final MyManifest manifest = new MyManifest();
// Create a new identifier for it
final CurationPersistentIdentifier cpid = FrameworkWrapper.allocateNewPID();
// Set the new identifier into the manifest
manifest.setManifestCpid(cpid);
// Populate the rest of the manifest's fields...

// Find a registry to save it to (fictional private method)
final Registry registry =
         getChosenRegistryFrom(FrameworkWrapper.getEnabledRegistries());

// [Optional] Set a (fictional) credentials provider.
//   This is usually only done once – not for every call.
registry.setCredentialsProvider(new MyCredentialsProvider(user, pass));

try {
    // Assert that we have write access to Registry
    registry.authoriseForReadWrite();

    // Attempt to store the Manifest if the chose Registry is not read-only.
    if (registry.isWritable()) {
         FrameworkWrapper.storeManifest(manifest, registry);
    }

} catch (final RIException rie) {
    // Handle store failure
    LOG.error(rie);
}
```

To exclude a Registry from being searched or used:

```
for (final Registry reg : FrameworkWrapper.getAvailableRegistries()) {
    if (reg.getLocationUID().equals("ESA")) {
         reg.setEnabled(false);
    }
}
```

Appendix A – Predefined RepInfo Categories

The APA switchboard holds the central list of all the predefined RepInfo Categories, and the most up-to-date list can be seen on that server itself, which is currently located at: http://switchboard.digitalpreserve.info/categories.txt

This list should not, however, be considered exhaustive, and indeed is expected to alter over time. Rep Info Labels and Manifests may carry any text as a 'category' and can be used to supplement or extend this list as required by end-users. That said, where possible a predefined entry should be used in order to maximise the discoverability and potential reuse for any given registry object.

Here is straight list of the current RepInfo Categories, as defined at the time of writing:

```
Other
Other/Registry
Other/Registry/RepInfoLabel
Other/Registry/Manifest
Other/AccessSoftware
Other/Algorithms
Other/CommonFileTypes
Other/ComputerHardware
Other/ComputerHardware/BIOS
Other/ComputerHardware/CPU
Other/ComputerHardware/Graphics
Other/ComputerHardware/HardDiskController
Other/ComputerHardware/Interfaces
Other/ComputerHardware/Network
Other/Media
Other/Physical
Other/ProcessingSoftware
Other/RepresentationRenderingSoftware
Other/Software
Other/Software/Binary
Other/Software/Data
Other/Software/Documentation
Other/Software/SourceCode
Other/Software/JavaClassConstructor
Other/Software/OperatingSystem
Semantic
Semantic/Data
Semantic/DictionarySpecification
Semantic/DictionarySpecification/Dictionary
Semantic/Document
Semantic/Document/XMLDocument
Semantic/Language
Semantic/Language/ComputerProgramming
Semantic/Language/ComputerProgramming/VendorExtensions
Semantic/Language/HumanWritten
Semantic/Models
Semantic/Standards
Semantic/Standards/DevelopingOrganisation
Semantic/Standards/DevelopingOrganisation/Standard
Structure
Structure/Container
Structure/Formats
Structure/Formats/DescriptionLanguageSpecification
Structure/Formats/DescriptionLanguageSpecification/FileDescription
Structure/Formats/DataFileType
Structure/Formats/Specification
```

Appendix B – Known Issues and Planned Development

The following list is by no means exhaustive, and is not necessarily in priority order, however it is here to highlight currently foreseen additional development which may be necessary over the remainder of the project:

1. Registry adapters to be separated out of the core codebase. This will allow registry implementations to be more easily excluded/included by an application.
2. Review of Java object construction, including adding remote JAR loading.
3. Define Provenance objects and their retrieval.