ICAT4 API

Introduction

The ICAT4 API is a rather thin layer on top of a relational DBMS. The database is wrapped as a web service and the actual tables are not exposed directly. Each table in the database is mapped onto a data structure exposed by the web service. When the web service interface definition (WSDL) is processed for Java then each data structure results in a class definition. The examples shown here all assume the use of Java.

Setting Up

The web service is accessed via a proxy (conventionally known as a port). The proxy may be obtained by the following:

```
URL url = null;
url = new URL("http://localhost:8080" + "/ICATService/ICAT?wsdl");
QName qName = new QName("client.icat3.uk", "ICATService");
ICATService service = new ICATService(icatUrl, qName);
ICAT proxy = service.getICATPort();
```

The first part of the icatURL should be modified to reference the ICAT installation you are using. A sessionId can then be obtained by logging in to ICAT:

```
String sessionId = proxy.login("userName", "password");
```

This sessionId will be valid for a period of two hours. Configuring userName, password pairs is dependent upon the chosen authentication module and is not discussed here. The sessionId is used as an argument to all subsequent calls.

Data Manipulation

Creating an Object

To create an object in ICAT first instantiate the object of interest for example a Dataset and then call the setters to set its attributes and finally make a call to create the object in ICAT.

So typical code in Java might look like:

```
Dataset ds = new Dataset();
ds.setName("Name of dataset");
ds.set …
Long dsid = proxy.create(sessionId, ds);
```

You will see that no convenient constructors are generated, rather each field of the object must be set individually. Some fields are optional and may be left with null values, however some are compulsory and the call to create will fail if they are not set. There is currently no way of knowing what is the smallest set of fields that need be set - however the number is generally quite small.[1] Each object has a primary key that identifies it in the database - this may be of three kinds.

1. In the simple case, you provide a string that has not been used before as then name you give when creating your object. This may be recognised by the presence of a field called *name* of type *String.* An example is DatasetType.
2. Alternatively an object may be identified by multiple fields. In this case the fields must be defined and if the combination is unique a “surrogate key” which is a “small integer” (actually implemented as a Long) is generated by the database. You can remember this key value to access the object quickly via a *get* call. This case may be recognised by the presence of a variable with a name of *id* and a type of *Long.* An example is Dataset.
3. Finally when an object is identified by multiple fields an alternative strategy is sometimes used. Here the key is a separate object: you set the fields of the key and then set the key value in the object to be created. To look this up quickly you need to create the key object and set all its fields then pass it to a *get* call. An example is Parameter.

Some fields represent attributes of the object but others are used to represent relationships. The relationships are typically represented in the class definitions as by a variable which either holds a reference to a single object or a collection of objects. Take a look at a Dataset class definition. This has a variable investigationId which must be set to the (surrogate) key of the investigation. This implies that the Investigation must be created first. You will see that there are two Collections: datafileCollection and datasetParameterCollection implemnted as lists of Datafiles and of DatasetParameters. These collections may be populated before calling create on the Dataset and the whole structure will be created in the database[2].

The call to create returns the key of the created object. If you choose to write:

```
ds.setId((Long)proxy.create(sessionId, ds));
```

then the client copy of the dataset will be updated to have the correct key value - however the keys in any other objects below the dataset if you had included a list of datafiles will still be null on the client side.

Retrieving an object when you know its key

If dsid is the key of a Dataset then it may be retrieved by the call:

```
Dataset ds = (Dataset) proxy.get(sessionId, “Dataset”, dsid);
```

[1]Those pieces of information that cannot be derived by inspection of the code - will be accessible by a “reflection” call which can be made available from the command line.
[2]Currently, because of the database structure, this does not work for DatasetParameters nor for DatafileParameters.

The second parameter is a string holding the name of the type of object to retrieve and some other optional information. By default only the requested object is returned and not related objects. If you want the Dataset along with its related Datafiles, DatasetParameters and DatafileParameters then replace

```
"Dataset"
```

with

```
"Dataset INCLUDE Datafile,DatasetParameter,DatafileParameter"
```

Where the object types to include must be collections representing one-to-many relationships from the starting object.[3]

Updating an object

To update an object simply update the fields you want to change and call update. For example:

```
Dataset ds = (Dataset) proxy.get(sessionId, "Dataset", dsid);
ds.setInvestigationId(93L);
proxy.update(sessionId, ds);
```

Changes to primary key fields will be ignored as will the contents of any collections. However as indicated in the example above many-to-one relationships, such as the investigation relationship to the dataset, may be updated.

Deleting an object

The following code will get a dataset and delete it.

```
Dataset ds = (Dataset) proxy.get(sessionId, "Dataset", dsid);
proxy.delete(sessionId, ds);
```

or, as only the primary key matters,:

```
Dataset ds = new Dataset();
ds.setId(dsid);
proxy.delete(sessionId, ds);
```

Searching for an object

A rather powerful search mechanism is provided. It will be introduced by means of examples:

```
List<?> results = proxy.search(sessionId, "Dataset");
```

[3]Currently there are some restrictions on what can be returned.

will return all Datasets. If the query is:

```
“Dataset.name”
```

this will return all Dataset names. Multiple datasets with the same name are permitted and this call will include duplicates.

```
"DISTINCT Dataset.name”
```

will avoid duplicates. To get related objects returned, then the same INCLUDE syntax that was described for the *get* call may be used:

```
“Dataset INCLUDE Datafile,DatasetParameter,DatafileParameter”
```

You can specify an order:

```
“Dataset.id ORDER BY id”
```
[4]

Restrictions can be placed on the data returned. For example:

```
“Dataset.id [datasetType IN ('GS', 'GQ')]”
```

which could also written:

```
“Dataset.id [datasetType = 'GS' OR  datasetType = 'GQ']”
```

The restriction in the square brackets can be as complex as required - but must only refer to to attributes of the object being restricted - in this case the Dataset. Expressions may use parentheses, AND, OR, <, <=, >, >=, =, <>, !=, NOT, IN, LIKE and BETWEEN[5].

Functions may also be used such as:

```
“MAX (Dataset.id)”
```
[6]

Selection may involve more than one related object. To show the relationship a “<->” token is used. For example:

```
Dataset.id <-> DatasetParameter[datasetParameterPK.name = 'TIMESTAMP'
```

Note also here the use of the JPQL style path: `“datasetParameterPK.name”`.

---

[4]If you have both INCLUDE and ORDER BY then the include must currently be first. This restriction will be removed.
[5]BETWEEN does not currently work for strings. This appears to be a bug in the JPA implementation.
[6]Functions are not yet implemented

This expressions means ids of Datasets which have a DatasetParameter which has a datasetParameterPK with a name of TIMESTAMP. Multiple <-> may appear but all the objects involved, including the first one, must be connectable in only one way.

Authorization

The mechanism is rule based. Rules allow groups of users to do things. There are currently four things that can be done: Create, Read, Update and Delete. This will probably be extended to add Download as a separate action. This will be required because of the push towards open metadata where most of the ICAT data will have Read access but the access to the catalogued data needs to be separately controlled.

Groups

The first task is to add users to groups. This is done by:

```
proxy.addUserGroupMember(sessionId, “operator”, “fred”);
proxy.addUserGroupMember(sessionId, “operator”, “bill”);
```

which adds users fred and bill to the group operator (creating it if necessary). The sessionId must be that of a user called “root”. [7]

By default access is denied to all objects, rules allow access to be allowed. It is only necessary to be permitted by one rule where that rule is only applied to the object referenced directly in the API call.

Rules

The rule:

```
proxy.addRule(sessionId, "user-office", "Investigation", "CRUD",
null);
```

allows members of the user-office group full access to Investigation. To save space only the four interesting arguments will be shown in the examples. So the rule above would be shown as:

```
"user-office", "Investigation", "CRUD", null
```

The rule:

```
null, "Parameter", "R", null
```

allows any authenticated user (with a sessionId) to read Parameters. Consider a group of users “FredReaders” To allow FredReaders to read a datafile with a name of “fred” we could

[7]Currently rules and the relationship between users and groups are handled differently from other things in the API. This will be tidied up shortly.

have:

```
    “FredReaders”, “Datafile”, “R”, ”[name = ‘fred’]”
```

More complex restrictions can be added using other related objects. For example to allow read access to Datasets belonging to an Investigation which includes an Investigator which has a facilityUser with a facilityUserId matching the current user (from the sessionId) we can have:

```
null, "Dataset", "R", "<-> Investigation <-> Investigator
[facilityUser.facilityUserId = :user]"
```

where the :user is the current user. You will note that the syntax is very similar to that user by the search.[8]

[8]The second and fourth parameters will be merged so that this example will read: `null, "R", "Dataset <-> Investigation <-> Investigator [facilityUser.facilityUserId = :user]"` to make the syntax closer to that of the search.