Java Server Documentation

Version alpha

(Work in progress)

Table of Contents

Chapter 1. Introduction

In addition to the reduced developer load, the choice of alternative existing O/R implementations such as Hibernate or Java Data Objects (JDO), the compiled language and more extensive feature list bring significant performance gains.

1.1. Project Information

```
LICENSE

    OME is distributed with the GNU Lesser General Public License
    (LGPL). You should be aware that any code submitted to the OME
    project will also fall under this license. If you have issues
    relating to licensing, want to sub license parts of the OME source
    code, or wish to discuss the licensing scheme please contact the
    OME core developers (ome-devel@lists.openmicroscopy.org.uk).

COPYRIGHT

    The OME source code, specification, documentation and database schema are,
    in their entirety Copyright (C) 2003 Open Microscopy Environment,
    Massachusetts Institute of Technology, National Institutes of Health,
    University of Dundee. As with the OME license, any questions related to the
    copyright, copyright holders or copyright scheme should be directed to the
    OME core developers (ome-devel@lists.openmicroscopy.org.uk).

DEVELOPER LOCATIONS

    The OME core developers are split between three main locations: OME Boston
    (MIT and Harvard; managed by Peter Sorger and Erik Brauner), OME Dundee
    (University of Dundee; managed by Jason Swedlow) and OME Baltimore (NIH;
    managed by Ilya Goldberg).

RESOURCES

    Most of the OME developer related documentation is either in the
    source tree (/doc and pod in the source files) or located on the
    CVS webserver (http://cvs.openmicroscopy.org.uk/).  There is also
    a documentation website available at
    http://docs.openmicroscopy.org.uk/.

LINKS

        The OME Project:  http://www.openmicroscopy.org/
        OME developers:   http://cvs.openmicroscopy.org.uk/
        Web based CVS:    http://cvs.openmicroscopy.org.uk/cvsweb/
        Documentation:    http://docs.openmicroscopy.org.uk
        Bug Tracker:      http://bugs.openmicroscopy.org.uk

MAILING LISTS

        OME developers:   ome-devel@lists.openmicroscopy.org.uk
```

Chapter 2. Getting Started

As with any new framework, it takes a while to work through the various layers. We will assume that due to the development quality of the project that anyone interested will first begin by examining the source code. Let's start with building.

2.1. Building

The OMERO build system is currently based on Maven [http://maven.apache.org]. In addtion, to work with the Omero source base, you will need to have a fully working OME database, not necessarily Postgres, a Java Development Kit, and a Servlet container, as well as various environment variables.

- Install Java [http://java.sun.com]. *The Omero server code requires Java 5.* Also, set the JAVA_HOME environment variable to your JDK installation
- Install a Servlet Container (Tomcat, Jetty, Resin, or JBoss et al.). Most testing is done on Tomcat [http://jakarta.apache.org/tomcat].
- Install Maven. Set MAVEN_HOME to your Maven installation. Alternatively you can use the included Maven installation and set MAVEN_HOME to OMERO_HOME/lib/maven/. If you do this, also run the command: `OMERO_HOME/lib/maven/bin/install_repo.sh HOME_DIR/.maven/repository` install_repo.bat is available for Windows users. Also, place MAVEN_HOME/bin/maven(.bat) on your PATH.
- Copy docs/examples/build.properties.example to OMERO_HOME or HOME; edit the properties for your site.

  Note: This file should *not* be put under revision control!

Now you are ready to *build and install* Omero. Run `maven bootstrap` to prepare the installation. Then run `maven install`to place all jars and the war file in your local maven repository. Copy the war file under `OMERO_HOME/components/server/target` to your servlet container. Enjoy!

Alternatively, follow the container-specific instructions below.

2.2. Installing

If you are not building from source, but have downloaded the war (web-application resource) file then your work is a bit simpler. Simply copy the war (web application resource) file to your servlet container. Once it is unpacked, edit: `/WEB-INF/classes/spring.properties` to connect to the database.

Once you start your servlet container, you can run the test suite against it.

Note: These instructions are for releases only. If you are working from subversion, please see "Building" above to get things running.

2.2.1. Installing on Tomcat

There are several methods to make working with a Tomcat instance simpler.

Edit the Tomcat section of your build.properties file, mentioned under "Building".

- Run "maven" from OMERO_HOME If maven is not on your path, alternatively run: `OMERO_HOME/lib/maven/bin/maven`
- cd to OMERO_HOME/components/server
- Run `maven tomcat:install` If you have already installed once, you'll first need to `maven tomcat:remove` .

2.3. Using

~~DEPRECATED:Building clients that access the Omero service is as easy as having omero-client.jar and omero-model.jar on your classpath.~~

If you are working from the source distribution, a file will be created under `client/target/` named `Classpath.sh` which properly defines your CLASSPATH environment variable to include all the necessary jars. (TODO: working from binary distribution). In your classpath when using maven is also a file, `spring.properties` which will contain your connection information if you properly defined build.properties. The values in `spring.properties` are used by Spring to create your proxy to the server.

If your path is correctly set (as above), simply create a ServiceFactory and use it to obtain a Service.

```
        ServiceFactory services = new ServiceFactory();
        HierarchyBrowsing proxy = services.getHierarchyBrowsingService();
```

Chapter 3. Server Design

It is fairly easy to work with the server without understanding all of its layers. The API is cleary outlined in the `ome.api` package and the client proxies work *almost* as if the calls were being made from within the same virtual machine. The only current caveat is that objects returned between two different calls will not be referentially (i.e. `obj1 == obj2`) equivalent. We are working on removing this restriction.

To understand the full technology stack, however, there are several concepts which are of importance.

A **layered architecture** ensures that components only "talk to" the minimum necessary number of other components. This reduces the complexity of the entire system. The Omero services (or, "business layer") are made available through a presentation layer (currently only Hessian [http://caucho.com/hessian] remoting. Services make use of the DAO objects (for an explanatino of "DAO" see below), which hide away all details of the O/R mapping framework.

Ensuring a loose-coupling of various components is also facilitated by **dependency injection**. Dependency injection is the process of allowing a managing component to place a needed resource in a component's hand. Code for lookup or creation of resources, in turn, is unneeded, and explicit implementation details don't need to be hard-coded.

The **DAO pattern [http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html]** also plays an important role. It hides away the specifics of accessing the database, whether JDBC calls or a full Object-Relational mapping tool.

Object-relational mapping is the process of mapping relational tables to object-oriented classes.

Aspect-oriented programming, a somewhat new and misunderstood technology, is perhaps the last technology which should be mentioned...

These are the relative generic technologies that we are using, but a much more interesting question is *how* are we using them.

3.1. Object Model

At the core of the work on the Open Microscopy Environment is the definition of a vocabulary for working with microscopic data. This vocabulary has a representation in the specification, in the database (the *data model*), and in code. This last representation is the object model with which we will concern ourselves here.

Because of its complexity, the object model is generated, either from the specification or from the data model. It relies on no libraries and can be used in both the server and the client. Instances of the object model have no direct interaction with the database, rather the mapping is handled externally by the O/R framework.

By and large, generated classes are composed only of getter and setter fields for fields representing columns in the database. However, to make working with the model easier and, perhaps, more powerful, there are several features which we are considering including.

"OME Remote Objects" Currently Omero does not use remote or distributed objects. [1] But to making working with these objects a bit more like truly distributed objects it would be nice to add certain functionality.

The first and most important addition is making two objects returned from the server be the same reference. To allow this, however, the client developer will need to pass in a ConflictHandler which resolves any stale data.

[1] Just what are distributed objects? TODO

setCache() new apis setLazyLoader() retrieveField (Rather than getField)

Extensibility [IDEA] extended api: ome.model.[TYPE].I (inner static interface) fill(Prototype implements [TYPE].I) (copy procedure) synchonization! extGet(""),extPut(""),extRemove(""),extAll(); or cglib or modelEvents setWrapper?

In addition to the extended functionality of the new object model, there are some changes to the actual structure, the specification, that are needed.

* image_id ==> pixel_id where appropriate * plane_info * ACL (getting ownership in each table not MEX) * one table ; one class * cleaning up container relationships (project, category, screen, etc.) * replace ST definition ("ST is immutable") with locking meechanism * possibly versioning

Inheritance

3.2. Domain Language and Type Generation [SUGGESTION]

This model has two general parts: first, the long studied and well-established core model and second, the user-specified portion. It is vital that there is a central defintion of both parts of the object model. domain language: dsl immutability=true ==> cache=true private=true ==> umask=700 combined with user_umask, system_umask, group_umask?? on create_type defaults in DTD (with local overrides) cache=true meta model xml is parsed and store in the st, st_elements, data_table, data_column tables api for saying give me valueOfField(String field,int id)-->for lazy-loading.; MetaDao mdao; mdao.getType(field); gdao.getById(type,id); mdao.getField(obj,field); preview java/schema

small domain language --> generate schema/spec/model/...

links [fields immutable "from" & "to"]

goal: no exceptions in the model. simple domain language that can produce everything. no stored procedures. no special code. (ideal, i know.)

Namespaces

3.3. MEXes [DELIBERATION]

option 1: single mex per row option 2: range table/textual option 3: many-many tables option 4: virtual mex as "composite" option 5: mex as part of the metamodel.

3.4. Access Control [PROPOSAL]

The central idea of this access control proposal is that each row in the database should have an experimenter and a permissions field (possibly a group field). An experimenter on each row simplifies implementing public-vs.-private data at the cost of redundancy (as experimenter is also stored in the mex for an object). Permissions permit a more flexible control over information Similar to the access control permissions for Unix filesystems,

Graph walking. There are complex issues with such cyclical graphs in that pure hierarchy style permissions RWX are not sufficient, more specifically the X bit as it applies to directories won't work since a user has many different ways to access any given object. Rather we need to have an extra filtering step (outlined below) to

remove unreadable entries.

9 bit (like unix) RWX==>Read_Edit_Delete. Or combine edit and delete?

or 32 bit read, edit(includes delete), locked, versioned, inactive, ...

```
ome=# select perm from image;
   perm
-----------

 111100100
 111000000
 111100000
(4 rows)

ome=# select b'000000000'<(b'100100100'&perm) from perms;
 ?column?
----------

 t
 t
 t
(4 rows)

ome=# select b'000000000'<(b'000100100'&perm) from perms;
 ?column?
----------

 t
 f
 t
(4 rows)

ome=# select b'000000000'<(b'000000100'&perm) from perms;
 ?column?
----------

 t
 f
 f
(4 rows)


OR


 0 - (none)
  1 - read
  2 - read,update
  3 - read,update,insert
  4 - read,update,insert,delete
  5 - read,update,insert,delete,admin
  > 5 - custom
```

The need to filter out those objects from a graph for which a user doesn't have permission lead to the implementation of a visitor-like pattern.

3.5. Filtering

Model objects have a acceptFilter method which accepts an implementation of the interface filter:

```
     interface Filter {
     Object filter(Object)
     Map filter(Map)
     Collection filter(Collection)
```

```
Model filter(Model)
```

Each model object is responsible for calling the appropriate filter method for each of its fields *and setting the field value to the return value of the method call.*

Early in working with Hibernate a method for filtering out the un-initialized lazy proxies (TODO add discussion on this) was needed. Both a reflective (see revision TODO) and a code generation (see revision 176) were used in the server component to create external iterators (TODO: link to pattern) for each model object.

However, reflection in the model *could be* too slow, and each type of utility in the code generated case would have needed its own generation step. Now with a modified visitor pattern (TODO: link), we can easily create a filter, which walks a model graph and optionally changes any of its values.

In the case of the security filter, it works like so:

```
Model filter(Model m) {

    Credentials c =
      ((SecureContext)ContextHolder.getContext())
        .getAuthentication().getCredentials(); // Acegi Security API

    Permissions  p = m.getPermissions();

    if (! p.allowRead(c)){
        return null; // Sets this field to null
    }
    return m;

}
```

3.6. Rules [IDEA]

Drools. Checks on all writes, WriteBlocker with reason, Checks Mexes (Blocks if user tries to enter a mex, they are created based on roles. We need an AE Role)

This is most important for writes to the database, both "saves" (of new data) and "updates" of existing data (including deletes.) Any operation which changes the database can be proceeded by a call such as:

```
Object[] objs;
//...
rules.check(objs); // throws exception if something's amiss.
dao.write(objs);
```

An example of a rule which we currently have but which is not formalized is the concept of an image being a member of one and only one category in a category group. If we were to implement this as a rule:

```
condition(Classification cl) {
  i = cl.getImage();
  c = cl.getCategory();
  cg = c.getCategoryGroup();
  c2 = cg.contains(i);
  if (c.id != c2.id) return true;
}

consequence(){
  throw new ImageCategoryExeption();
```

```
```

rule api to create conditions, which may make conditions useable from within the type-creation language.

```
      <type>
        <field name="field1">...
        <rule>
            <condition<
            return dao.conflicts(field1);
          </condition<
            <consequence<
            throw new Exception();
          </consequence<
        <rule>
      </type>
```

3.7. Working with graphs

3.8. Importing

3.9. Client libraries

Though Omero is mostly a server-side project, established libraries for accessing that server make using it substantially easier.

connectivity: asynchrounous: jmx,remoting,... events: changes from other clients (how many clients currently connected!)

cache: jbossAOPTreeCache http://docs.jboss.com/jbcache/current/TreeCacheAop/html/ OR jcache with CacheLoader! (with jms for sync) OR OScache (persistent -- changes between restarts! classes too? ==> CacheClassLoader) aop over client (or even server if have jdbc connection [pattern needs name]) ConflictResolver (per agent) new ome.client.ServiceFactory() -- no caching ome.client.ServiceFactory.withCaching(ConflictResolver) -- with caching if conflictResolver==null, throws OptimisticLockException at commit time.

```
    LAZYLOAD
    have ProxyFilter call model.UnloadedField(String fieldName);
    lazyLoader can then check and call:
    api.load(Field,id);
    this uses another filter which Hibernate.initializes() all fields (and members of sets) of an object.

    DSL
    api is currently:
       user.createType(dslXml)
    we could optionally offer:
       admin.createType(dslXml,hbmXml)
    in which namespace info etc., comes from dslXml, but special information like constraints, etc.
    could be specificed directly in the hbm. mapping file (changes with other technology)
    :: modelling exclusivity (==>rules?)
    :: currently using Hibernate to model

    NOTIFICATION:
      exception handling so all get notified
      care with the synch'ing.

      HibernateUpdateEvent-->
```

```
          JMS Msg with triple (objLsid, fieldLsid, newValue) -->
          client cache gets event, looks up objLsid in cache -->
          calls new tripleSetFilter(jmsMsg).filter(obj) -->
          call obj.notify(UpdateEvent);

      MODEL Sync.
        only if observers > 0
        only if using omero-model-multithread.jar (MultiThreadClassLoader)


      Model Report Objects (with counts of all collections)
        "lite" objects using polymorph=explicit (or one-to-one link "VIEW")
        other possibility is to define imageCount-style properties on the main model obj.
          and to use lazy-field loading???


      Usage Log: (within Sec. Filter to throttle users)

      Filter Password

      SEC: onLoad, LSID subclass of each type

      START: tomcat manager
```

Chapter 4. Developing with/for the Java server

4.1. Server-side development

4.1.1. Quick How-To

- Add to common/src/ome/api interface "[API]" Run "maven jar:install" for common
- Add to server/src/ome/logic class "[API]Impl" Add to server/src/ome/dao interface "[API]Dao" Add to server/src/ome/dao/hibernate class "[API]DaoHibernate"
- Code. (This may include writing queries in a "[API]Queries.hbm.xml" file
- Update spring configuration in: server/web/WEB-INF/{services.xml,dao.xml}
- Write test by extending AbstractDepedencyInjectionSpringContextTests with proper getConfigLocations

4.2. Client-side development

If you are satisfied with working with the generated domain model objects directly, working with the server is nearly trivial. You simply need to supply connection paramters, either in a `spring.properties` on your classpath or through system variables. After that, all calls on any `ServiceFactory` object will return a functioning proxy.

If, however, the possibility instability of this worries you, it is straight forward to design an adapter around the existing model and API. To implement an adapter, you will need to define your own domain model objects and provide an implementation of the AdapterUtil iterfaces for your service. This will convert

Currently, there is no solution for adapting in the write direction. This is a result of the original intent of the server (read-only), and is currently on the TODO list.

4.3. Using Eclipse as an IDE

There are currently `.project` and `.classpath` files stored in subversion. Maven can reproduce them (for example, after changes to `project.xml`, but the existing files contain certain modifications that make working with the code base easier. You will need, however, certain classpath variables (`MAVEN_REPO`, `OMERO_HOME`, `USER_HOME`) to make them work.

More work needs to be done to make the Eclipse projects more useful. This will be completed at a later date.

Chapter 5. Testing

or *"Dependency injection, getConfigLocations, integration, oh my."*

5.1. Unit Testing

The unit testing framework is fairly simple. Only methods which contain logic written within the OME Java Server are tested. This means that framework functionality like remoting is *not* tested. Neither is DAO functionality; this is a part of integration testing. (see below)

Therefore, most of the code which is unit tested lies in the logic packages of the server component. This is done using jMock [http://jmock.org].

You can run the unit tests for any component from its directory by entering:

```
maven test -Dmaven.test.mode=unit

# or if you haven't changed the value of maven.test.mode simply:
maven test
```

The same can be done for all components using:

```
maven test-all
```

from the top-level directory.

5.2. Integration Testing

Integration testing is a bit more complex.

Because of there reliance on a database (which is not easily mockable), all DAO classes are tested in integration mode.

To run integration tests, use `-Dmaven.test.mode=integration` while calling the `test` and `test-all` maven targets mentioned under Unit Testing.

5.2.1. DbUnit Testing

Several special integration tests, based on the DbUnit [http://dbunit.sourceforge.net/] JUnit extension, are also included in the integration tests. This currently requires the creation of a special DB (specifically `"[your standard DB url]-test"`).

Currently, these tests will fail. Documentation on preparing these tests will be added later.

Chapter 6. Todo List

or, "Who what when?"

For a list of todo items, their priorities, assignees, and so forth please see TODO.html [./TODO.html].