DAO Prototype

NOTE: ***This is just a proposal. There is no guarantee that any of this will ever end up in the actual codebase, I just felt it was worth experimenting.*** **-JR**

Update (09-05-2007): *I've made this work successfully for* Collections*,* Items*, and* Bundles*. The performance improvements aren't fully implemented, but the separation is there, and in theory, that was the hard part. The* ArchiveManager *seems to be working pretty well too.* -JR

Update (11-05-2007): *I've implemented DAOs for the* Community *class as well. The* ArchiveManager *now supports moving* Items*,* Collection_, and_ Communities *between containers.* -JR

Update (23-05-2007): *I've totally reimplemented persistent identifiers in DSpace as well (see* PersistentIdentifiers *). As well as removing the Handle System dependency, they also use DAOs.* -JR

Update (20-06-2007): *After a bit of a hiatus while I* PersistentIdentifiers *I've come back to DAOs, and I've now (mostly) got them in place for* Bitstreams *as well. The two major classes that still need doing are* EPerson *and* Group_; once they're done, there are a few others (eg:_ SupervisedItem_,_ WorkspaceItem_, etc) but they should be relatively simple._ -JR

Just adding a comment that Handle/Pid management could be greatly improved by such an addition as well. currently with item caching, the DSpaceObject.getHandle method can become stale and using DAO's behind the scene for the HandleManagement might be beneficial -- Mark Diggory 13: 56, 10 May 2007 (EDT)

Update (14-08-2007): *Everything (apart from the code in* org.dspace.checker_*)* has been pushed through the DAO layer. Non-DAO classes no longer_ import *the* DatabaseManager *or* throw SQLExceptions*. There are interfaces for CRUD and link operations in* org.dspace.storage.dao *that I intend to write some tests to for throwing at all the implementing DAOs.* -JR

It has often struck me that DSpace would benefit from the use of Data Access Objects (DAO). If nothing else, it would make porting to alternative database platforms far easier; all we would need to do is provide alternative implementations for the DAO interfaces that worked for a given database. To this end, I have broken up some of the core classes in org.dspace.content to use DAOs.

As part of the same effort, I have done some work on making the Context less data-layer dependent (by having it hold a `org.dspace.storage.dao.GlobalDAO` rather than a `java.sql.Connection`, etc). I've also introduced a proxy for the Item that is a bit smarter about when it retrieves content from the data layer, and an ArchiveManager class that takes care of some core "archive operations" (so that other core classes don't need to).

The process to integration (if at all) would go as follows:

- Incorporate the new DAO classes into the codebase
- Refactor org.dspace.content.Item (etc) to use the DAO implementations of the data access methods internally
- Mark relevant methods in org.dspace.content.Item as @Deprecated
- Using the compile-time deprecation warnings as a guide, refactor the rest of the code to use the DAOs explicitly rather than hiding the functionality behind existing methods

Without further ado, here is how I have refactored org.dspace.content.Item to use DAOs. A few important things to note:

- "old" code has been used where possible to avoid re-implementing the wheel
- I've never liked org.dspace.content.ItemIterator so I've switched to using a "real" Iterator from a List<Item>

For examples of both of these principles, see the implementation of getItems() below. It is a fairly straightforward wrapper for the current Item.findAll(), except that it returns a List<Item> rather than an ItemIterator.

Contents

Error rendering macro 'toc'

null

org.dspace.content

The Item class will be broken up into the following classes:

- `org.dspace.content.Item`: core class that doesn't go near the database (it doesn't even know about the DAOs); behaves much like the current implementation.
- `org.dspace.content.dao.ItemDAO`: interface defining DAO API
- `org.dspace.content.dao.ItemDAOFactory`: factory for dishing out implementations of the above interface
- `org.dspace.content.dao.postgres.ItemDAOPostgres`: default implementation of the above interface for use with PostgreSQL

- `org.dspace.content.proxy.ItemProxy`: subclass of Item that needs to know about the DAO. It will be used for (eg) only loading metadata on demand, to reduce the memory footprint of Items etc.

The following classes have also been introduced:

- `org.dspace.core.ArchiveManager`
- `org.dspace.storage.dao.GlobalDAO`
- `org.dspace.storage.dao.GlobalDAOFactory`
- `org.dspace.storage.dao.GlobalDAOPostgres`

Note that it might be preferable to have a more generic implementation of the `ItemDAO` interface that supports both PostgreSQL and Oracle, but given that one motivation for adopting DAOs is to remove db-specificities from the code making it easier to port, I thought it was sensible to start with just PostgreSQL. Eventually, it ought to be possible to drop in `ItemDAOHibernate` (etc) implementations that make db portability *far* easier.

org.dspace.content.Item

Basic implementation of the Item object. This class has been stripped down to remove all contact with the database, including (but not limited to) contstructors, factory methods, `update()`, `delete()`, `find()`, etc. I haven't decided exactly how the Item API will look, but it will probably be much the same as before, only with any of the aforementioned methods. Another key difference is that it will have actual Java objects as member variables instead of pulling everything out of a `TableRow`.

org.dspace.content.proxy.ItemProxy

This will be a fairly simple proxy implementation. Specifically, it will be closest to being a *virtual proxy*, in that it will appear to be a regular `Item` object, but will have a slightly smarter implementation (not loading metadata until requested, keeping track of what has changed to make updates more efficient etc).

```
public class ItemProxy extends Item
{ // Overrides relevant methods of Item. }
```

org.dspace.content.dao.ItemDAO

This isn't final, but it's a good start.

```
public interface ItemDAO extends ContentDAO
    implements CRUD<Item>, Link<Item, Bundle>
{
    public Item create() throws AuthorizeException;
    public Item retrieve(int id);
    public Item retrieve(UUID uuid);
    public void update(Item item) throws AuthorizeException;
    public void delete(int id) throws AuthorizeException;
    public List<Item> getItems();
    public List<Item> getItemsBySubmitter(EPerson eperson);
    public List<Item> getItemsByCollection(Collection collection);
    public List<Item> getParentItems(Bundle bundle);
}
```

org.dspace.content.dao.ItemDAOFactory

```
public class ItemDAOFactory
{
    public static ItemDAO getInstance(Context context)
    {   // Eventually, the implementation that is returned will be
        // defined in the configuration.
        return new ItemDAOPostgres(context);
    }
}
```

org.dspace.content.dao.postgres.ItemDAOPostgres

This is a fairly straightforward implementation of the above interface. As much as possible, code from the original `Item` class will be used. For instance, this is how `getItems()` is implemented:

```
public List<Item> getItems()
{
    try
    {
        TableRowIterator tri = DatabaseManager.queryTable(context, "item",
            "SELECT item_id FROM item WHERE in_archive = '1'");

        List<Item> items = new ArrayList<Item>();
        for (TableRow row : tri.toList())
        {
            int id = row.getIntColumn("item_id");
            items.add(retrieve(id));
        }
        return items;
    } catch (SQLException sqle){
        // Need to think more carefully about how we deal with SQLExceptions
        throw new RuntimeException(sqle);
    }
}
```

Some changes have been made to eliminate `ItemIterator`, and to generally make things a little more consistent with the rest of the code (this looks almost identical to, eg, `CollectionDAO.getCollections()`.

org.dspace.core

org.dspace.core.ArchiveManager

The idea behind this class came from the realisation that `Item.withdraw()` and `Item.reinstate()` don't really make sense. What I'd much rather do is call (eg) `ArchiveManager.withdrawItem(Item item)`.
I've been thinking that the `ArchiveManager` could be used for certain maintenance operations as well, such as moving `Items` between `Collections`, and maybe acting as a wrapper for the `CommunityFiliator`.

```
public class ArchiveManager
{
    public static void withdrawItem(Context context, Item item)
    {       // ...      }

    public static void reinstateItem(Context context, Item item)
    {       // ...      }
    public static void moveItem(Context context, Item item, Collection source, Collection dest)
    {       // ...      }
}
```

org.dspace.storage

org.dspace.storage.dao.GlobalDAO

As suggested by Richard Jones, there probably ought to be a top-level general-purpose DAO interface that has implementations for the various storage mechanisms (`GlobalDAOPostgres` etc). The idea is to have this top-level object capture any implementation-specific details in a single top-level object, rather than in every Postgres DAO implementation. For example, with the current database "abstraction layer", the top-level implementation of `GlobalDAO` understands the `Context` object, whereas a Hibernate implementation would know what a `SessionFactory` is.

```
public interface GlobalDAO
{
    // The following methods actually currently throw SQLExceptions to
    // keep things simple, but in future SQLExceptions should be
    // eliminated from any code that doesn't directly touch a database.
    public void startTransaction() throws GlobalDAOException;
    public void endTransaction() throws GlobalDAOException;
    public void saveTransaction() throws GlobalDAOException;
    public void abortTransaction();
    public boolean transactionOpen();
    @Deprecated Connection getConnection();
}
```

org.dspace.storage.dao.GlobalDAOFactory

Super-simple `GlobalDAO` factory.

org.dspace.storage.dao.GlobalDAOPostgres

Implementation of the `GlobalDAO` interface for PostgreSQL.

```
public class GlobalDAOPostgres implements GlobalDAO
{
    private Connection connection;

    // ...

    public void startTransaction()
    {
         connection = DatabaseManager.getConnection();
         connection.setAutoCommit(false)
    // ...
}
```