DSpace Service based api

This API refactoring was approved for the DSpace 6.0 release

This Service based API refactoring was approved for release in DSpace 6.0 via a vote taken on the 'dspace-devel' mailing list.

What is the Service-based API refactoring?

The Service-based API refactoring is a major refactoring of the "dspace-api" (DSpace's Java API) to better support "separation of concerns /responsibilities". Simply put, often, in the existing (5.x and below) API, there is an intermingling of business logic and database logic which is difficult to maintain, debug and/or build against. One of the most obvious examples is how we deal with database software support (PostgreSQL vs. Oracle), but such intermingling of logic exists in many of our core classes. The DSpace "Service-based API" attempts to tease apart the database logic from the business logic into separate layers, while also adding support for Hibernate. The goal is to provide an easier to maintain, more modular API, while also enhancing how we deal with database logic in general (via Hibernate).

- This is essentially a *new* Java API for DSpace. (Some areas of our API are still the same, but most have been refactored)
  - NOTE: It is not backwards compatible with the existing (5.x and below) Java API.
- It modularizes our primary API in a way that makes it much easier to achieve future goals on our Roadmap (especially, moving us towards potentially better support of third-party modules)
- It cleans up one of the "messiest" areas of our existing API, the Database management / hardcoded Oracle and PostgreSQL queries, in favor of using Hibernate. This allows us the potential to support additional database platforms in the future (e.g. MySQL or similar). It decreases the likelihood of Oracle-specific bugs (which have always been a problem), as the Oracle queries are delegated to Hibernate. It also simplifies the process of testing for database-specific problems in general (as again, all queries are delegated to Hibernate).
- It begins teasing apart a true "business logic layer" in the API (see the "service layer" of this new API)
- It provides (internal) globally unique UUIDs for all DSpace objects (instead of existing incremental, non-unique database identifiers)
- The refactored API itself should not affect the fresh installation or upgrade process of DSpace 6.0, provided that you have not made local changes that rely on the existing Java API (dspace-api). If you have made such local API changes, your refactor process should not be difficult. But, it will be necessary before you can successfully upgrade to 6.0. See "Tutorials" below for examples.

Additional Resources

This Service API was presented/discussed in a Special Topic Developer Meeting on July 23, 2015. Slides and video from that meeting are now available:

- Presentation Slides (from Ben Bosman): Service Based API.pdf
- Video of meeting (~60mins including Q&A, requires Flash to view): http://educause.acms.com/p7hebf8ddc7/
- Code that was merged on Oct 1st 2015 in the DSpace master branch: https://github.com/DSpace/DSpace/pull/1083

Module refactoring now COMPLETE!

All existing DSpace interfaces/user interfaces have been refactored to support the new API, and all unit tests succeed. This refactoring took place on the DS-2701-service-api feature branch.

We'd like to thank the following individuals for their hard work in helping to refactor and test our various modules to support this new API:

- [x] New dspace-api - Kevin Van de Velde (Atmire) - NOTE: This merger must occur before any other work can begin.
- [x] dspace-jspui - Andrea Bollini (4Science) and Kevin Van de Velde (Atmire) and Mark H. Wood
- [x] dspace-lni - Robin Taylor will refactor LNI out entirely (as suggested in DS-2124)
- [x] dspace-oai - Mark H. Wood
- [x] dspace-rdf - Pascal-Nicolas Becker
- [x] dspace-rest - Peter Dietz (DSPR#1026)
- [x] dspace-services (will this be completed as part of the inital refactor?)
- [x] dspace-solr (should not need refactoring. No dspace-api methods called)
- [x] dspace-sword - Andrea Schweer
- [x] dspace-swordv2 - Andrea Schweer
- [x] dspace-xmlui-mirage2 - Kevin Van de Velde (Atmire)
- [x] dspace-xmlui - Kevin Van de Velde (Atmire)
- [x] Unit Testing Framework fixes - Tim Donohue

- 1 Introduction
- 2 How things are now
    - 2.1 Bad distribution of responsibilities
    - 2.2 Database layer access
    - 2.3 Postgres/oracle support
    - 2.4 Alternate implementation of static flows
    - 2.5 Local modifications
    - 2.6 Inefficient caching mechanism
    - 2.7 Excessive database querying when retrieving objects
- 3 Service based api
    - 3.1 DSpace api structure
        - 3.1.1 Service layer
        - 3.1.2 Database Access Layer
        - 3.1.3 Database Object
        - 3.1.4 Schematic representation
    - 3.2 Hibernate: the default DAO implementation
    - 3.3 Service based api in DSpace (technical details)
        - 3.3.1 Database Object: Technical details
            - 3.3.1.1 Tracking changes in database classes
        - 3.3.2 Database Access Layer: Technical details
        - 3.3.3 Service Layer: Technical details
            - 3.3.3.1 Database based object based services
            - 3.3.3.2 Business logic services
            - 3.3.3.3 Using a service outside of spring beans
- 4 Changes to the DSpace api
    - 4.1 Changes to the DSpace domain model
        - 4.1.1 DSpaceObject is a separate database table (UUID is new identifier)
        - 4.1.2 Site is now a database object
        - 4.1.3 Using old static DSpaceObject methods
        - 4.1.4 DSpaceObject creation changes
    - 4.2 ItemIterator class removed
    - 4.3 Context class and database connection
    - 4.4 Redundant features that have been removed
    - 4.5 Workflow system refactoring
- 5 How the service api solves design issues
    - 5.1 Bad distribution of responsibilities
    - 5.2 Database layer access
    - 5.3 Postgres/oracle support
    - 5.4 Alternate implementation of static flows
    - 5.5 Local Modifications
    - 5.6 Inefficient caching mechanism
- 6 Current development status
    - 6.1 What there is now
    - 6.2 What still needs to be done for a DSpace 6.0 release
    - 6.3 Long term improvements
- 7 Tutorials
    - 7.1 Editing item metadata
    - 7.2 Adding files to an item
    - 7.3 Adding permissions to a bitstream
    - 7.4 Moving an item to another collection
    - 7.5 Archive an item
    - 7.6 Withdraw an item
    - 7.7 Edit collection metadata
    - 7.8 Creating new EPerson
    - 7.9 Adding EPerson to a Group
    - 7.10 Altering an existing method in a service
    - 7.11 Adding a new method to a service
    - 7.12 Creating a new service without database access
    - 7.13 Creating a new service with database table
        - 7.13.1 Introduction
        - 7.13.2 Database changes
        - 7.13.3 Database object
        - 7.13.4 Service interface
        - 7.13.5 Data Access Object (DAO) interface
        - 7.13.6 Factory
            - 7.13.6.1 Create new service factory
            - 7.13.6.2 Create a new implementation of the service factory
            - 7.13.6.3 Configure the service factory in spring
        - 7.13.7 Service implementation
            - 7.13.7.1 Configure the Service implementation in spring
        - 7.13.8 Data Access Object (DAO) implementation
            - 7.13.8.1 Configure the Service implementation in spring
    - 7.14 Example code

Introduction

The DSpace API was originally created in 2002. After working with the API day in day out for over 7 years I felt it was time for a change. In this proposal I will highlight the issues that the current API has and suggest solutions for these issues.

How things are now

Bad distribution of responsibilities

Currently every database object (more or less) corresponds to a single java class. This single java class contains the following logical code blocks:

- CRUD methods (Create & Retrieve methods are static)
- All business logic methods
- Database logic (queries, updating, …)
- All getters & setters of the object
- ….

Working with these types of GOD classes, we get the following disadvantages:

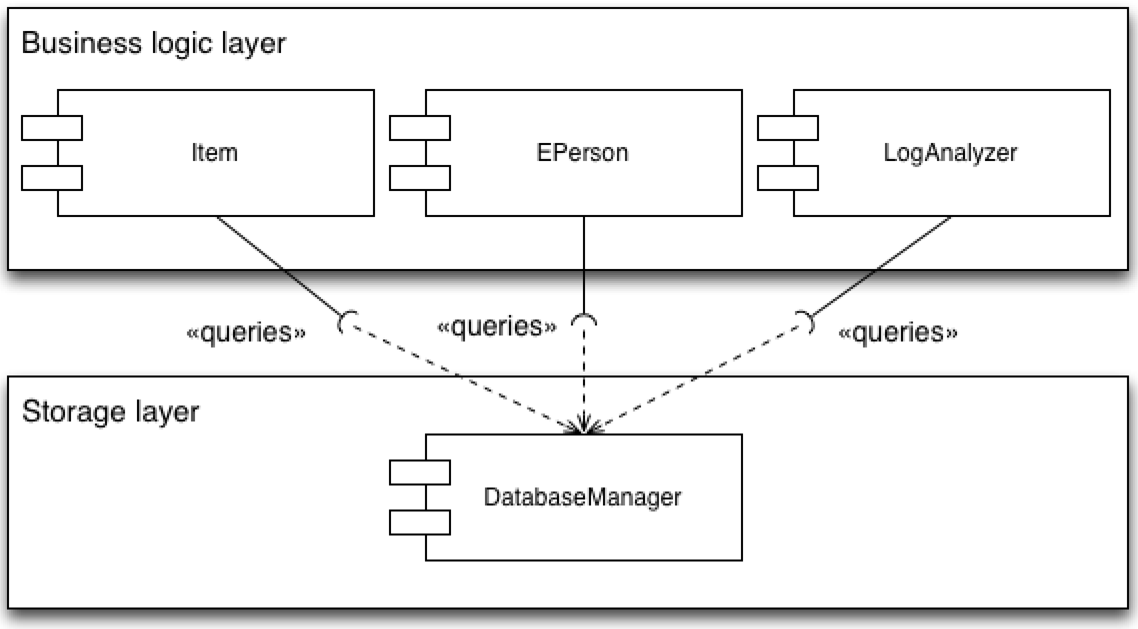
- By grouping all of these different functionality's into a single class it is sometimes difficult to separate the setters/getters from the business logic.
- Makes it very difficult to make/track local changes since you need to overwrite the entire class to make a small change.
- Refactoring/altering GOD classes becomes complex since it is unclear where the database logic begins & where the business logic is located.

Database layer access

All database access is made through a DatabaseManager, this object can be accessed from any object (including the UI layer), some examples:

- EPerson table is queried from the Groomer and EPerson class.
- Item is queried from the LogAnalyser, Item, EPerson classes

As a consequence of this, making changes to a database table becomes more complex since it is unclear which classes have access to the database table. Below is a schematic overview of the usage of the item class:



Postgres/oracle support

Since DSpace supports both postgres AND oracle this classes contains a lot of "if postgres do X or else if oracle do Y". This tends to lead to bugs & makes support for 2 database system hard to maintain.
Since every query needs to be run against oracle and postgres, developers need to write 2 queries or use query concatenation which can lead to complex code to create your queries, below is an example of such a complex query:

```
String params = "%"+query.toLowerCase()+"%";
StringBuffer queryBuf = new StringBuffer();
queryBuf.append("select e.* from eperson e " +
        " LEFT JOIN metadatavalue fn on (resource_id=e.eperson_id AND fn.resource_type_id=? and fn.
metadata_field_id=?) " +
        " LEFT JOIN metadatavalue ln on (ln.resource_id=e.eperson_id AND ln.resource_type_id=? and ln.
metadata_field_id=?) " +
        " WHERE e.eperson_id = ? OR " +
        "LOWER(fn.text_value) LIKE LOWER(?) OR LOWER(ln.text_value) LIKE LOWER(?) OR LOWER(email) LIKE LOWER(?)
ORDER BY  ");
if(DatabaseManager.isOracle()) {
    queryBuf.append(" dbms_lob.substr(ln.text_value), dbms_lob.substr(fn.text_value) ASC");
}else{
    queryBuf.append(" ln.text_value, fn.text_value ASC");
}
// Add offset and limit restrictions - Oracle requires special code
if (DatabaseManager.isOracle())
{
    // First prepare the query to generate row numbers
    if (limit > 0 || offset > 0)
    {
        queryBuf.insert(0, "SELECT /*+ FIRST_ROWS(n) */ rec.*, ROWNUM rnum  FROM (");
        queryBuf.append(") ");
    }
    // Restrict the number of rows returned based on the limit
    if (limit > 0)
    {
        queryBuf.append("rec WHERE rownum<=? ");
        // If we also have an offset, then convert the limit into the maximum row number
        if (offset > 0)
        {
            limit += offset;
        }
    }
    // Return only the records after the specified offset (row number)
    if (offset > 0)
    {
        queryBuf.insert(0, "SELECT * FROM (");
        queryBuf.append(") WHERE rnum>?");
    }
}
else
{
    if (limit > 0)
    {
        queryBuf.append(" LIMIT ? ");
    }
    if (offset > 0)
    {
        queryBuf.append(" OFFSET ? ");
    }
}
```

Alternate implementation of static flows

Adding an alternate solution to a DSpace default "flow" is not an easy undertaking.
Looking at the configurable workflow (introduced in DSpace 1.8), the choice between which workflow to use if an item has gone through the submission process depends on an “if this than workflow X else workflow Y”. Should there ever be a third option the “if statement" would expand even further and would need to be adjusted in a lot of places. A small sample of this is displayed below.

```
// Should we send a workflow alert email or not?
if (ConfigurationManager.getProperty("workflow", "workflow.framework").equals("xmlworkflow")) {
    if (useWorkflowSendEmail) {
        XmlWorkflowManager.start(c, wi);
    } else {
        XmlWorkflowManager.startWithoutNotify(c, wi);
    }
} else {
    if (useWorkflowSendEmail) {
        WorkflowManager.start(c, wi);
    }
    else
    {
        WorkflowManager.startWithoutNotify(c, wi);
    }
}
```

Both workflow systems use separate objects to represent objects that are in the workflow which leads to more if workflow X else workflow Y logic, a great example is when deleting a collection. When we delete a collection we need to delete all workflow items linked to that collection, a code sample is included below.

```
if(ConfigurationManager.getProperty("workflow","workflow.framework").equals("xmlworkflow")){
    // Remove any xml_WorkflowItems
    XmlWorkflowItem[] xmlWfarray = XmlWorkflowItem
            .findByCollection(ourContext, this);
    for (XmlWorkflowItem aXmlWfarray : xmlWfarray) {
        // remove the workflowitem first, then the item
        Item myItem = aXmlWfarray.getItem();
        aXmlWfarray.deleteWrapper();
        myItem.delete();
    }
}else{
    // Remove any WorkflowItems
    WorkflowItem[] wfarray = WorkflowItem
            .findByCollection(ourContext, this);
    for (WorkflowItem aWfarray : wfarray) {
        // remove the workflowitem first, then the item
        Item myItem = aWfarray.getItem();
        aWfarray.deleteWrapper();
        myItem.delete();
    }
}
```

Local modifications

Take for example the WorkflowManager, this class consist of only static methods, if a developer for an institution wants to alter one method the entire class needs to be overridden in the additions module. When the Dspace needs to be upgraded to a new version it is very difficult to locate the changes in a big class file.

Inefficient caching mechanism

DSpace caches all DSpaceObjects that are retrieved within a single lifespan of a context object (can be a CLI thread of a page view in the UI). The cache is only cleared when the context is closed (but this is coupled with a database commit/abort) or when explicitly clearing the cache or removing a single item from the cache. When writing large scripts one must always keep in mind the cache, because if the cache isn't cleared DSpace will keep running until an OutOfMemory exception occurs. So contributions by users with a small dataset could lead to memory leaks.

Excessive database querying when retrieving objects

When retrieving a bundle from the database it automatically loads in all bitstreams linked to this bundle, for these bitstreams it retrieves the bitstreamFormat. For example when retrieving all bundles for an item to get the bundle names all files and their respective bitstream formats are loaded into memory. This leads to a lot of obsolete database queries which aren't needed since we only need the bundle name.

Service based api

DSpace api structure

To fix the current problems with the API and make it more flexible and easier to use I propose we split the API in 3 layers:

Service layer

This layer would be our top layer, will be fully public and used by the CLI, JSPUI & XMLUI, ... Every service in this layer should be a stateless singleton interface. The services can be subdivided into 2 categories **database based services** and **business logic block services**.

The **database based services** are used to interact with the database as well as providing business logic methods for the database objects. Every table in DSpace requires a single service linked to it. The services themselves will not perform any database queries, it will delegate all database queries to the Database Access Layer discussed below. So for example the service for the Item class would contain methods for moving it between collections, adjusting the policies, withdrawing, ... these methods would be executed by the service itself while the database access methods like create, update, find, ... would add business logic to check the authorisations but delegate the actual database calls to the database access layer.

The **business logic services** are replacing the old static DSpace Manager classes. For example the AuthorizeManager has been replaced with the AuthorizeService which contains the same methods and all the old code except that the AuthorizeService is an interface and a concrete (configurable) implemented class contains all the code. This class can be extended/replaced by another implementation without ever having to refactor the use of the class.

Database Access Layer

As discussed above to continue working with a static DatabaseManager class that contains if postgres else oracle code will lead to loads of bugs, therefore I propose we replace this “static class” by a new layer.

This layer is called the Database Acces Object layer (DAO layer). It contains **no business logic** and it's sole responsibility is to **provide database access for each table** (CRUD (create/retrieve/update/delete)). This layer consist entirely of interfaces which are automagically linked to implementations by using spring. The reason for interfaces is quite simple, by using interfaces we can easily replace our entire DAO layer without actually having to alter our service layer. So if one would like to use another ORM framework than the default, all that needs to be done is implement all the interfaces and configure them, no code changes are required to existing code.

Each database table has its own DAO interface as opposed to the old DatabaseManager class, this is to support Object specific CRUD methods. Our metadataField DAO interface has a findByElement() method for example while an EPerson DAO interface would require a findByEmail() retrieval method. These methods are then called by the service layer which has similar methods.

In order to avoid queries to the DAO layer from various points in the code **each DAO can only be utilised from a single service**. Linking a DAO to multiple services will result in the messy separation we are trying to avoid.

The name of each class **must end with DAO.** A **single database table can only be queried from a single DAO** instance.

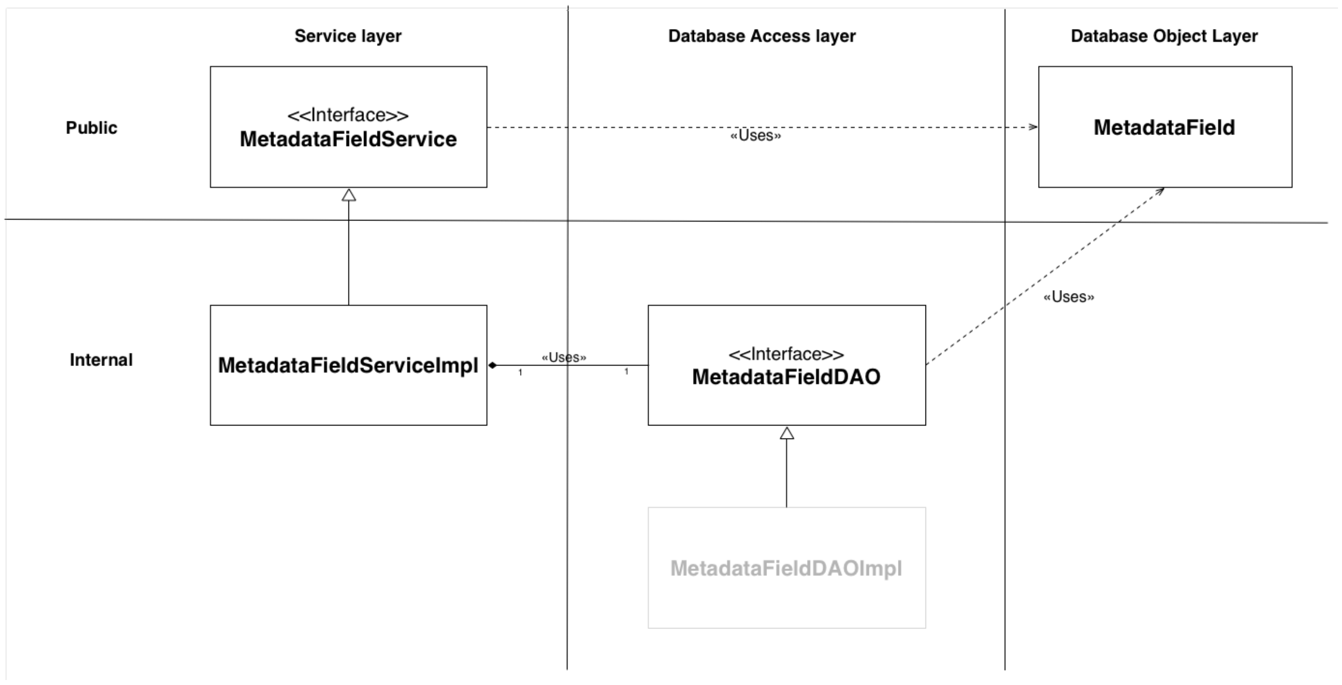
It is also important to remember that the DAO layer **should never be exposed outside of the service layer**. This is to avoid skipping past the business logic which in turn will lead to conflicts of responsibility.

Database Object

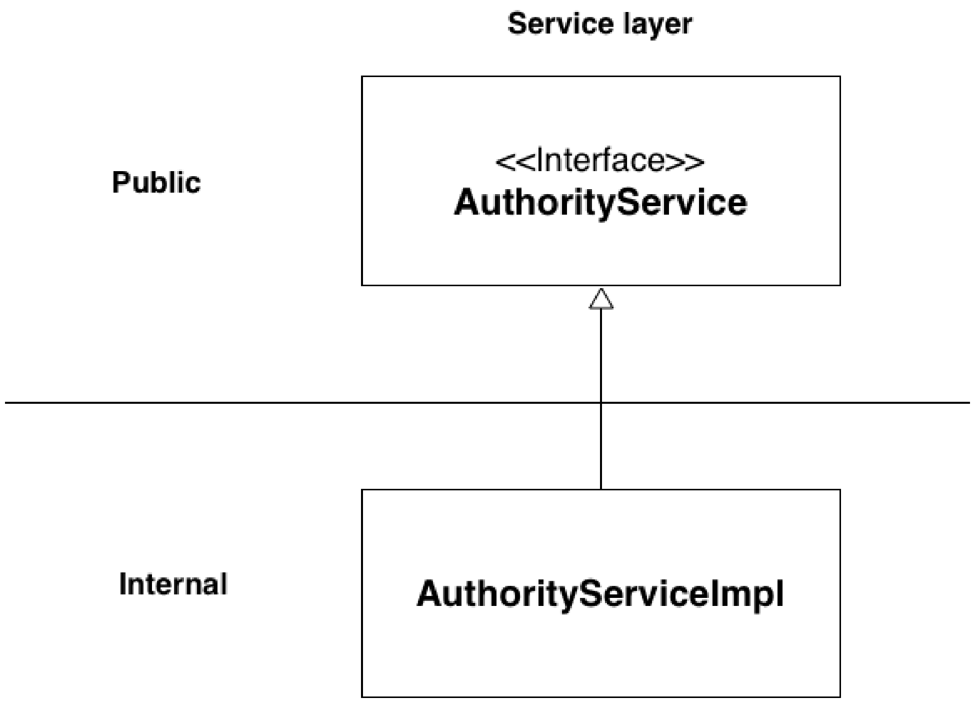
Each table in the database that is not a linking table (collection2item, community2collection) is represented by a database object. This object contains no business logic and has setters/getters for all columns (these may be package protected if business logic is required). Each Data Object has its own DAO interface as opposed to the old DatabaseManager class, this is to support Object specific CRUD methods.

Schematic representation

Below is a schematic representation of how a refactored **database based object** class would look (using the simple MetadataField class as an example):



Below is a schematic representation of how a refactored **business logic services** class would look (using the AuthorityService (previously AuthorityManager) class as an example):



The public layers objects are the only objects that can be accessed from other classes, the internal objects represent objects whose only usage is documented below. This way the internal usage can change in its entirety without ever affecting the DSpace classes that use the api.

Hibernate: the default DAO implementation

Instead of straight JDBC implementation, we wanted to harness the power and simplicilty of an Object-Relational Mapping (ORM), that would allow querying and manipulating the database through an object paradigm. Hibernate was the particular implementation chosen. The reasons that motivated the preference for Hibernate include:

- **Performance:** When retrieving a bundle the bitstreams are loaded into memory, which in turn load in the bitstreamformat, which in turn loads in file extensions, ….. This consists of a lot of database queries. With hibernate you can use lazy loading to only load in certain linked objects at the moment they are requested.
- **Effective cross database portability:** No need to write “if postgres query X if oracle query Y” anymore, hibernate takes care of all of this for you !
- **Developers’ Productivity:** Although hibernate has a learning curve, once you get past this linking objects & writing queries takes considerably less time than the old way where each link would require a developer to write all the sql queries themselves.
- **Improved caching mechanism:** The current DSpace caching mechanism throws objects onto a big pile until the pile is full which will result in an OutOfMemory exception. Hibernate auto caches all queries in a single session and even allows for users to configure which tables should be cached across sessions (application wide).

Nevertheless, we are convinced the choice for Hibernate is not necessarily a permanent one, since the proposed architecture easily allows replacing it with another backend ORM implementation or even a JDBC based one.

Service based api in DSpace (technical details)

Database Object: Technical details

Each non linking database table in DSpace must be represented by a single class containing getters & setters for the columns. Linked objects can also be represented by getters and setters, below is an example of the database object representation of MetadataField. These classes cannot contain any business logic.

```
@Entity
@Table(name="metadatafieldregistry", schema = "public")
public class MetadataField {

    @Id
    @Column(name="metadata_field_id")
    @GeneratedValue(strategy = GenerationType.AUTO ,generator="metadatafieldregistry_seq")
    @SequenceGenerator(name="metadatafieldregistry_seq", sequenceName="metadatafieldregistry_seq",
allocationSize = 1)
    private Integer id;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "metadata_schema_id",nullable = false)
    private MetadataSchema metadataSchema;

    @Column(name = "element", length = 64)
    private String element;

    @Column(name = "qualifier", length = 64)
    private String qualifier = null;

    @Column(name = "scope_note")
    @Lob
    private String scopeNote;

    protected MetadataField()
    {

    }

    /**
     * Get the metadata field id.
     *
     * @return metadata field id
     */
    public int getFieldID()
    {
        return id;
    }

    /**
     * Get the element name.
     *
     * @return element name
     */
    public String getElement()
    {
        return element;
    }
    /**
     * Set the element name.
     *
     * @param element new value for element
     */
    public void setElement(String element)
    {
        this.element = element;
    }

    /**
     * Get the qualifier.
     *
     * @return qualifier
     */
    public String getQualifier()
    {
        return qualifier;
    }

    /**
     * Set the qualifier.
     *
     * @param qualifier new value for qualifier
     */
    public void setQualifier(String qualifier)
    {
        this.qualifier = qualifier;
    }

    /**
     * Get the scope note.
     *
     * @return scope note
     */
    public String getScopeNote()
    {
        return scopeNote;
    }

    /**
     * Set the scope note.
     *
     * @param scopeNote new value for scope note
     */
    public void setScopeNote(String scopeNote)
    {
        this.scopeNote = scopeNote;
    }

    /**
     * Get the schema .
     *
     * @return schema record
     */
    public MetadataSchema getMetadataSchema() {
        return metadataSchema;
    }

    /**
     * Set the schema record key.
     *
     * @param metadataSchema new value for key
     */
    public void setMetadataSchema(MetadataSchema metadataSchema) {
        this.metadataSchema = metadataSchema;
    }



    /**
```

```
     * Return <code>true</code> if <code>other</code> is the same MetadataField
     * as this object, <code>false</code> otherwise
     *
     * @param obj
     *            object to compare to
     *
     * @return <code>true</code> if object passed in represents the same
     *         MetadataField as this object
     */
    @Override
    public boolean equals(Object obj)
    {
        if (obj == null)
        {
            return false;
        }
        if (getClass() != obj.getClass())
        {
            return false;
        }
        final MetadataField other = (MetadataField) obj;
        if (this.getFieldID() != other.getFieldID())
        {
            return false;
        }
        if (!getMetadataSchema().equals(other.getMetadataSchema()))
        {
            return false;
        }
        return true;
    }

    @Override
    public int hashCode()
    {
        int hash = 7;
        hash = 47 * hash + getFieldID();
        hash = 47 * hash + getMetadataSchema().getSchemaID();
        return hash;
    }

    public String toString(char separator) {
        if (qualifier == null)
        {
            return getMetadataSchema().getName() + separator + element;
        }
        else
        {
            return getMetadataSchema().getName() + separator + element + separator + qualifier;
        }
    }

    @Override
    public String toString() {
        return toString('_');
    }
}
```

This example demonstrates the use of annotations to represent a database table. With hardly any knowledge of hibernate you can get a quick grasp of how the layout of the table will look, by just looking at the variables at the top. This documentation will not expand on the annotations used, the hibernate documentation is far more suitable here: https://docs.jboss.org/hibernate/stable/annotations/reference/en/html/entity.html.

Another improvement demonstrated above is the fact that linking of objects can now be done by using annotations. Just take a look at the metadataSchema variable, this variable can be gotten by using the getter, the database query to retrieve the schema will NOT be executed until we request the schema using the getter. This makes linking of objects a lot easier since no queries are required.

Each database entity class must be added to the hibernate.cfg.xml files, these files can be found in the resources directory in the additions and test section of the dspace-api. an excerpt from the file is displayed below, it displays how entities are configured.

```
<mapping class="org.dspace.content.Item"/>
<mapping class="org.dspace.content.MetadataField"/>
<mapping class="org.dspace.content.MetadataSchema"/>
<mapping class="org.dspace.content.MetadataValue"/>
<mapping class="org.dspace.content.Site"/>
```

The constructor of an entity **can never be public**, this is to prevent the creation of an entity from the UI layers (which is something we do not want). The service creates the entity, calls setters and getters that are required and then calls upon the database access layer to create the object.

Tracking changes in database classes

The database entities are the only classes that are stateful. By default all variables in an entity are linked to database columns, but sometimes the need arises to store other information in an object that we don't want to write to the database. An example of this would be to track if the metadata of an item has been modified, since if it has been modified and we trigger an update we want to fire a "Metadata_modified" event. But this "variable" would be useless inside a database table. To accomplish this we can make use of the *Transient* annotation, just add @Transient above a variable and it can be used to contain a state of a certain variable which will not be written to the database. Below is an example of how this looks for the item's metadataModified flag.

```
@Transient
private boolean metadataModified = false;
boolean isMetadataModified()
{
        return metadataModified;
}
void setMetadataModified(boolean metadataModified) {
    this.metadataModified = metadataModified;
}
```

Database Access Layer: Technical details

Each database object must have a single database access object, this object will be responsible for all Create/Read/Update/Delete calls made to the database. The DAO will always be an interface this way the implementation can change without ever having to modify the service class that makes use of this DAO.

Since each database object requires it's own DAO this will result in a lot of "duplicate" methods. A "GenericDAO" class was created with basic support for the CRUD methods; it is recommended for every DAO to extend from this interface. The current implementation of this interface is displayed below.

```
public interface GenericDAO<T>
{
    public T create(Context context, T t) throws SQLException;

    public void save(Context context, T t) throws SQLException;

    public void delete(Context context, T t) throws SQLException;

    public List<T> findAll(Context context, Class<T> clazz) throws SQLException;

    public T findUnique(Context context, String query) throws SQLException;

    public T findByID(Context context, Class clazz, int id) throws SQLException;

    public T findByID(Context context, Class clazz, UUID id) throws SQLException;

    public List<T> findMany(Context context, String query) throws SQLException;

}
```

The generics ensure that the DAO classes that are extending from this class cannot use these methods for other classes. Below is an example of the interface for the metadataField table, it extends the GenericDAO class and adds its own specific methods to the DAO.

```
public interface MetadataFieldDAO extends GenericDAO<MetadataField> {

    public MetadataField find(Context context, int metadataFieldId, MetadataSchema metadataSchema, String
element, String qualifier)
            throws SQLException;

    public MetadataField findByElement(Context context, MetadataSchema metadataSchema, String element, String
qualifier)
            throws SQLException;

    public MetadataField findByElement(Context context, String metadataSchema, String element, String qualifier)
            throws SQLException;

    public List<MetadataField> findAllInSchema(Context context, MetadataSchema metadataSchema)
            throws SQLException;
}
```

The GenericDAO is extended using the MetadataField type, this means that the create, save, delete methods from the GenericDAO can only be used with an instance of MetadataField.

It could very well be that a certain entity doesn’t require any specialised methods, but we do require an implementation class for each DAO to get the generic methods.

Since a developer doesn't want to duplicate code to implement the "generic" methods a helper class was created, this way the implementation of the generic methods resides in one place. For the hibernate DAO implementation the class is named AbstractHibernateDAO. It extends the GenericDAO, implements all the generic methods and also comes with a few additional helper methods. These additional helper methods are shortcuts so you don't have to write the same couple of lines of codes for each method. Some examples: return a type casted list from a query, return an iterator from a query, ... Below is the current implementation of the AbstractHibernateDAO (for reference only).

```
public abstract class AbstractHibernateDAO<T> implements GenericDAO<T> {

    @Override
    public T create(Context context, T t) throws SQLException {
        getHibernateSession(context).save(t);
        return t;
    }

    @Override
    public void save(Context context, T t) throws SQLException {
        getHibernateSession(context).save(t);
    }

    protected Session getHibernateSession(Context context) throws SQLException {
        return ((Session) context.getDBConnection().getSession());
    }

    @Override
    public void delete(Context context, T t) throws SQLException {
        getHibernateSession(context).delete(t);
    }

    @Override
    public List<T> findAll(Context context, Class<T> clazz) throws SQLException {
        return list(createCriteria(context, clazz));
    }

    @Override
    public T findUnique(Context context, String query) throws SQLException {
        @SuppressWarnings("unchecked")
        T result = (T) createQuery(context, query).uniqueResult();
        return result;
    }

    @Override
    public T findByID(Context context, Class clazz, UUID id) throws SQLException {
        @SuppressWarnings("unchecked")
        T result = (T) getHibernateSession(context).get(clazz, id);
        return result;
    }

    @Override
    public T findByID(Context context, Class clazz, int id) throws SQLException {
        @SuppressWarnings("unchecked")
        T result = (T) getHibernateSession(context).get(clazz, id);
        return result;
    }

    @Override
    public List<T> findMany(Context context, String query) throws SQLException {
        @SuppressWarnings("unchecked")
        List<T> result = (List<T>) createQuery(context, query).uniqueResult();
        return result;
    }

    public Criteria createCriteria(Context context, Class<T> persistentClass) throws SQLException {
        return getHibernateSession(context).createCriteria(persistentClass);
    }

    public Criteria createCriteria(Context context, Class<T> persistentClass, String alias) throws SQLException
{
        return getHibernateSession(context).createCriteria(persistentClass, alias);
    }

    public Query createQuery(Context context, String query) throws SQLException {
        return getHibernateSession(context).createQuery(query);
    }

    public List<T> list(Criteria criteria)
    {
        @SuppressWarnings("unchecked")
        List<T> result = (List<T>) criteria.list();
        return result;
    }

    public List<T> list(Query query)
    {
        @SuppressWarnings("unchecked")
        List<T> result = (List<T>) query.list();
        return result;
    }

    public T uniqueResult(Criteria criteria)
    {
        @SuppressWarnings("unchecked")
        T result = (T) criteria.uniqueResult();
        return result;
    }

    public T uniqueResult(Query query)
    {
        @SuppressWarnings("unchecked")
        T result = (T) query.uniqueResult();
        return result;
    }

    public Iterator<T> iterate(Query query)
    {
        @SuppressWarnings("unchecked")
        Iterator<T> result = (Iterator<T>) query.iterate();
        return result;
    }

    public int count(Criteria criteria)
    {
        return ((Long) criteria.setProjection(Projections.rowCount()).uniqueResult()).intValue();
    }

    public int count(Query query)
    {
        return ((Long) query.uniqueResult()).intValue();
    }

    public long countLong(Criteria criteria)
    {
        return (Long) criteria.setProjection(Projections.rowCount()).uniqueResult();
    }
}
```

With our helper class in place creating & implementing the a metadataFieldDAO class just becomes a case of implementing the metadataField specific methods. The MetadataFieldDAOImpl class was created, it extends the AbstractHibernateDAO and implements the MetadataFieldDAO interface. Below is the current implementation:

```
public class MetadataFieldDAOImpl extends AbstractHibernateDAO<MetadataField> implements MetadataFieldDAO {

    @Override
    public MetadataField find(Context context, int metadataFieldId, MetadataSchema metadataSchema, String
element,
                           String qualifier) throws SQLException{
        Criteria criteria = createCriteria(context, MetadataField.class);
        criteria.add(
                Restrictions.and(
                        Restrictions.not(Restrictions.eq("id", metadataFieldId)),
                        Restrictions.eq("metadataSchema", metadataSchema),
                        Restrictions.eq("element", element),
                        Restrictions.eqOrIsNull("qualifier", qualifier)
                )
        );
        return uniqueResult(criteria);
    }

    @Override
    public MetadataField findByElement(Context context, MetadataSchema metadataSchema, String element, String
qualifier) throws SQLException
    {
        Criteria criteria = createCriteria(context, MetadataField.class);
        criteria.add(
                Restrictions.and(
                        Restrictions.eq("metadataSchema", metadataSchema),
                        Restrictions.eq("element", element),
                        Restrictions.eqOrIsNull("qualifier", qualifier)
                )
        );
        return uniqueResult(criteria);
    }

    @Override
    public MetadataField findByElement(Context context, String metadataSchema, String element, String
qualifier) throws SQLException
    {
        Criteria criteria = createCriteria(context, MetadataField.class);
        criteria.createAlias("metadataSchema", "s");
        criteria.add(
                Restrictions.and(
                        Restrictions.eq("s.name", metadataSchema),
                        Restrictions.eq("element", element),
                        Restrictions.eqOrIsNull("qualifier", qualifier)
                )
        );
        return uniqueResult(criteria);
    }

    @Override
    public List<MetadataField> findAllInSchema(Context context, MetadataSchema metadataSchema) throws
SQLException {
        // Get all the metadatafieldregistry rows
        Criteria criteria = createCriteria(context, MetadataField.class);
        criteria.add(Restrictions.eq("metadataSchema", metadataSchema));
        return list(criteria);
    }
}
```

The DAO implementations in DSpace make use of the criteria hibernate object to construct its queries. This makes for easy readable code, even with a basic understanding of sql you can easily write queries. Read more about Criteria in the hibernate documentation.

Now that we have a DAO implementation we also need to configure it in spring, this is done in the [dspace.dir]/config/spring/api/core-dao-services.xml file. It it mandatory to keep the DAO a singleton so the scope attribute of a bean must be absent (defaults to singleton) or set to singleton. Below is the configuration of the MetadataFieldDAO implementation shown above.

```
<bean class="org.dspace.content.dao.impl.MetadataFieldDAOImpl"/>
```

Service Layer: Technical details

The service layer is where all our business logic will reside, each service will consist of an interface and an implementation class. Every stateless class in the DSpace api should be a service instead of a class consisting of static methods.

Database based object based services

The service layer encompasses all business logic for our database objects, for example we used to have Item.findAll(context), now we have itemService.findAll(context) (where itemService is an interface and & findAll is not a static method). Each database object has exactly one service attached to it, which should contain all the business logic and calls upon a DAO for its database access. Each of these services must have an implementation configured in a spring configuration file.

Below is an example of the MetadataFieldService, it clearly specifies the “business logic” methods which one would expect. Including create, find, update and delete.

```
public interface MetadataFieldService {

    /**
     * Creates a new metadata field.
     *
     * @param context
     *            DSpace context object
     * @throws IOException
     * @throws AuthorizeException
     * @throws SQLException
     * @throws NonUniqueMetadataException
     */
    public MetadataField create(Context context, MetadataSchema metadataSchema, String element, String
qualifier, String scopeNote)
            throws IOException, AuthorizeException, SQLException, NonUniqueMetadataException;

    /**
     * Find the field corresponding to the given numeric ID.  The ID is
     * a database key internal to DSpace.
     *
     * @param context
     *            context, in case we need to read it in from DB
     * @param id
     *            the metadata field ID
     * @return the metadata field object
     * @throws SQLException
     */
    public MetadataField find(Context context, int id) throws SQLException;

    /**
     * Retrieves the metadata field from the database.
     *
     * @param context dspace context
     * @param metadataSchema schema
     * @param element element name
     * @param qualifier qualifier (may be ANY or null)
     * @return recalled metadata field
     * @throws SQLException
     */
    public MetadataField findByElement(Context context, MetadataSchema metadataSchema, String element, String
qualifier)
            throws SQLException;

    /**
     * Retrieve all metadata field types from the registry
     *
     * @param context dspace context
     * @return an array of all the Dublin Core types
     * @throws SQLException
     */
    public List<MetadataField> findAll(Context context) throws SQLException;

    /**
     * Return all metadata fields that are found in a given schema.
     *
     * @param context dspace context
     * @param metadataSchema the metadata schema for which we want all our metadata fields
     * @return array of metadata fields
     * @throws SQLException
     */
    public List<MetadataField> findAllInSchema(Context context, MetadataSchema metadataSchema)
            throws SQLException;


    /**
     * Update the metadata field in the database.
     *
     * @param context dspace context
     * @throws SQLException
     * @throws AuthorizeException
     * @throws NonUniqueMetadataException
     * @throws IOException
     */
    public void update(Context context, MetadataField metadataField)
            throws SQLException, AuthorizeException, NonUniqueMetadataException, IOException;

    /**
     * Delete the metadata field.
     *
     * @param context dspace context
     * @throws SQLException
     * @throws AuthorizeException
     */
    public void delete(Context context, MetadataField metadataField) throws SQLException, AuthorizeException;
}
```

The DAO layer discussed above should be a completely internal layer, it should never be exposed outside of the service layer. If a certain service requires a DAO, the recommended way to make it available to the service implementation would be to autowire it. Below is a excerpt from the implementation class of the MetadataFieldService which shows how a service would be implemented.

```
public class MetadataFieldServiceImpl implements MetadataFieldService {
    /** log4j logger */
    private static Logger log = Logger.getLogger(MetadataFieldServiceImpl.class);

    @Autowired(required = true)
    protected MetadataFieldDAO metadataFieldDAO;

    @Autowired(required = true)
    protected AuthorizeService authorizeService;

    @Override
    public MetadataField create(Context context, MetadataSchema metadataSchema, String element, String
qualifier, String scopeNote) throws IOException, AuthorizeException, SQLException, NonUniqueMetadataException {
        // Check authorisation: Only admins may create DC types
        if (!authorizeService.isAdmin(context))
        {
            throw new AuthorizeException(
                    "Only administrators may modify the metadata registry");
        }

        // Ensure the element and qualifier are unique within a given schema.
        if (hasElement(context, -1, metadataSchema, element, qualifier))
        {
            throw new NonUniqueMetadataException("Please make " + element + "."
                    + qualifier + " unique within schema #" + metadataSchema.getSchemaID());
        }

        // Create a table row and update it with the values
        MetadataField metadataField = new MetadataField();
        metadataField.setElement(element);
        metadataField.setQualifier(qualifier);
        metadataField.setScopeNote(scopeNote);
        metadataField.setMetadataSchema(metadataSchema);
        metadataField = metadataFieldDAO.create(context, metadataField);
        metadataFieldDAO.save(context, metadataField);

        log.info(LogManager.getHeader(context, "create_metadata_field",
                "metadata_field_id=" + metadataField.getFieldID()));
        return metadataField;
    }

    @Override
    public MetadataField find(Context context, int id) throws SQLException
    {
        return metadataFieldDAO.findByID(context, MetadataField.class, id);
    }
```

This service class has an auto wired MetadataFieldDAO available, this auto wired class should always link to an interface and never to an implementation class. This way we can easily swap the DAO classes without having to touch the business logic.
When taking a closer look at the code it also becomes clear why we have this 3 tier api now. For example the “create” & “delete” methods check authorizations (with an auto wired authorization service) but they leave the actual creation/deletion to the DAO implementation.

What is also important to note for services is that these services are all “singletons”, only one instance of each service exist in the memory. The changes to the objects are handled by the "data objects”.

When working with interfaces and their implementation it is also important to make all internal service methods protected instead of private. This makes it easier to extend an existing implementation for making local implementation, since extending classes cannot use a private method.

Business logic services

On top of this, all of the “static method Managers” are also replaced by services, so the AuthorizeManager is now AuthorizeService, BitstreamStorageManager is now BitstreamStorageService and so on.
These services do not make use of a DAO, if a certain service is required to make changes to a certain database object (for example the BitstreamStoreService will want access to the Bitstream) then we autowire that service into the BitstreamStorageService and use the available methods of the BitstreamService.

A great example of an issue that has greatly benefitted from this change is the workflow. Before the service the following part of code was common to determine which workflow to use:

```
if(ConfigurationManager.getProperty("workflow", "workflow.framework").equals("xmlworkflow”))
{
    XmlWorkflowManager.start(c, wi)
}else{
    WorkflowManager.start(c, wi);
}
```

This way of working can be greatly simplified now, see below for the new code.

The old WorkflowManager code has now been moved to BasicWorkflowManager, this makes it easier to identify the workflows. It allows us to create a WorkflowService interface from which both the BasicWorkflowService & XmlWorkflowService inherit.

```
WorkflowServiceFactory.getInstance().getWorkflowService().start(context, workspaceItem);
```

Which is much a cleaner way and really shows of the benefits of working with services instead of static managers, replacing a service just becomes changing a certain class link in a spring file !

Using a service outside of spring beans

When using a service you need to request the bean for it. As a consequence, when you require a service inside another bean you can easily autowire it in. However, using services in non-Bean classes would potentially require a lot of code duplication, as demonstrated in the example below:

```
new DSpace().getServiceManager().getServiceByName(“???????”, MetadataFieldService.class)
```

This forces the user to remember and then look up a service by name, which isn’t quite that easy to use. To make the services easier to use, each package in DSpace that contains services comes with its own factory. As an example, you can retrieve the MetadataFieldService in the content package by calling:

```
ContentServiceFactory.getInstance().getMetadataFieldService()
```

This way, all you need to need to remember to get a certain service is the following, the factory class will have the following format: {package} ServiceFactory. Each of these factories comes with a static getInstance() method. The factory classes are split into an abstract class which has the getInstance() method and abstract methods for all the service getters. Below is an example of the AuthorizeServiceFactory class:

```
public abstract class AuthorizeServiceFactory {

    public abstract AuthorizeService getAuthorizeService();

    public abstract ResourcePolicyService getResourcePolicyService();

    public static AuthorizeServiceFactory getInstance()
    {
        return new DSpace().getServiceManager().getServiceByName("authorizeServiceFactory",
AuthorizeServiceFactory.class);
    }
}
```

The implementation of this factory overrides the abstract methods & returns the proper implementation of the services. The actual returned values are autowired beans. This way the factory hasn’t got a clue what the implementation actually is, so the control of all services now lies in a spring configuration file ! Below is an example of such an implementation.

```
public class AuthorizeServiceFactoryImpl extends AuthorizeServiceFactory {

    @Autowired(required = true)
    private AuthorizeService authorizeService;

    @Autowired(required = true)
    private ResourcePolicyService resourcePolicyService;

    @Override
    public AuthorizeService getAuthorizeService() {
        return authorizeService;
    }

    @Override
    public ResourcePolicyService getResourcePolicyService() {
        return resourcePolicyService;
    }
}
```

Changes to the DSpace api

Changes to the DSpace domain model

The introduction of Hibernate lead to following changes to the DSpace domain model.

DSpaceObject is a separate database table (UUID is new identifier)

Hibernate does not support the way how DSpace currently relies on "type" and "id" as a compound identifier, used to link tables together. Hibernate requires a single identifier to be used to link objects like metadata, handles, resource policies, ... to a single DSpaceObject implementation. To support this behavior a new table "DSpaceObject" was created with only a single column a UUID. All objects inheriting from DSpaceObject such as Community, Collection, Item, ... no longer have their own identifier column but link to the one used in DSpace Object. This has several advantages:

- You cannot delete a handle, metadataValue,.. without first deleting the object (or risk an sql exception)
- Each DSpaceObject now has its own unique identifier

The old integer based identifiers will still be available as a read only property, but these are not updated for new rows and should never be used for linking.

The old identifiers can easily be retrieved by using the getLegacyId() getter. Additionally all current DSpace objects services have a method termed findByIdOrLegacyId(Context context, String id) so if a certain part of the code doesn't know which type of identifier is used a developer can still retrieve the object it belongs to (without having to duplicate code to check if an identifier is a UUID or Integer). These methods are used (among others) by the AIP export to ensure backwards compatibility for exported items. Below are some code examples of how you can still use the legacy identifier.

```
//Find an eperson by using the legacy identifier
ePersonService.findByLegacyId(context, oldEPersonId);

//Find by old legacy identifier OR new identifier (can be used for backwards compatibility
ePersonService.findByIdOrLegacyId(context, id);
```

Site is now a database object

Since the DSpaceObject class now represents a database class, each class extending from this object must also be a database object. Therefore, the "Site" object, representing your instance of DSpace, also becomes a table in the database. The site will automatically be created when one is absent and the handle with suffix "0" will be automatically assigned to it. An additional benefit of having site has an object is that we can assign metadata to the site object. This is not yet supported in the code, but the possibilities are there.

Below is a code example of how you can retrieve the site object:

```
//Find the site
siteService.findSite(context);
```

Using old static DSpaceObject methods

Since there is no longer a possibility to use the old static DSpaceObject methods like find(context, type, id), getAdminObject(action), ... a generic DSpaceObjectService can be retrieved by passing along a type. This service can then be used to perform the old static methods, below are some code examples:

```
//Retrieve a DSpace object service by using a DSpaceObject
DSpaceObjectService dspaceObjectService = ContentServiceFactory.getInstance().getDSpaceObjectService(dso);

//Retrieve a DSpace object service by using a type
DSpaceObjectService dspaceObjectService = ContentServiceFactory.getInstance().getDSpaceObjectService(type);

//Retrieve an object by identifier
dspaceObjectService.find(context, uuid);

//Retrieve the parent object
dspaceObjectService.getParentObject(context, dso);

//One line replacement for DSpaceObject.find(context, type, id);
ContentServiceFactory.getInstance().getDSpaceObjectService(type).find(context, uuid);
```

DSpaceObject creation changes

In the current DSpace API, a collection can only be created by calling "createCollection()" on a community object, because the actual create method in the collection class could only be accessed from inside the package. The reasoning behind this implementation was to prevent a developer from creating a collection without a community. This behaviour was no longer possible with the service based API since an interface can only have public methods. As a result, the "creation" of a certain DSpaceObject has been moved to the service of that DSpaceObject. Therefore, creating a collection no longer requires you to call on a community but to use the collectionService.create(context, community) method. The parameters of this method ensure that a community is still required to create a collection. Below are some more code examples:

```
//Create a collection
collectionService.create(context, community);

//Create a bundle
bundleService.create(context, item, name);

//Create an item, a check is made inside the service to ensure that a workspace item can only be linked to a
single item.
itemService.create(context, workspaceItem);
```

Metadatum value class removed

The Metadatum class has been removed and is replaced by the MetadataValue class. The Metadatum class used to be a value representation which was detached from its original MetadataValue object, this was the only class in DSpace which works in this manner and it has been removed. The MetadataValue class is linked to the metadatavalue table and works like all other linked database objects, the change was important to support the lazy loading of metadata values, these will only be loaded once we request them.

Developers should keep in mind that when adding, clearing, modifying metadata that MetadataValue rows will be created. Since these values are linked to a certain metadata field it is important to keep in mind that a certain metadata field needs to be present prior to adding it to a DSpace object.

ItemIterator class removed

Although the ItemIterator class has been removed, the functionality to iterate over a list of items still remains, but instead of using a custom Iterator class the service will now return an "Iterator<Item>". The querying is done similar to the old ways, an initial query will retrieve only the identifiers and each "next()" call will query the item table to retrieve the next item from the iterator.

Context class and database connection

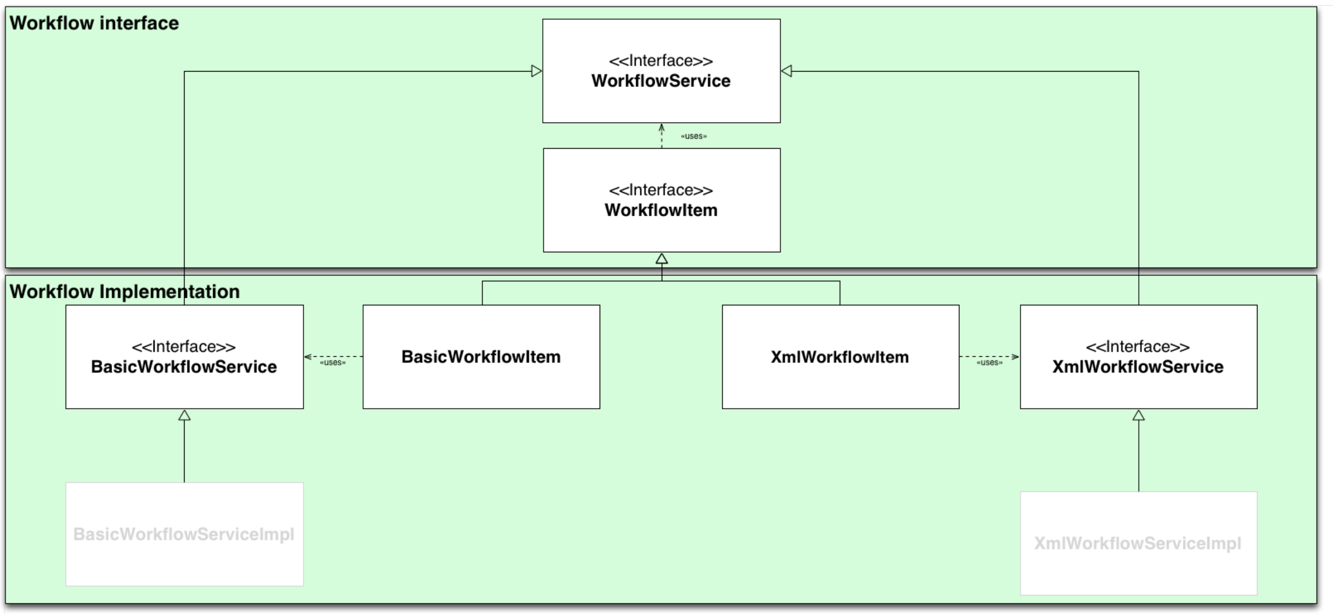
In the old DSpace API each time a new context object was created using its constructor, a new database connection was initialized. Hibernate uses a different principle, it shares a single database connection over an entire thread, so no matter how many *new Context()* calls you have in a single thread, only one database connection will remain open.

Redundant features that have been removed

Some of the older DSpace features could no longer be supported in hibernate, for example the database based browse system has been completely removed from the codebase. Another feature that will be removed is the database based OAI approach since this is not supported in hibernate.

Workflow system refactoring

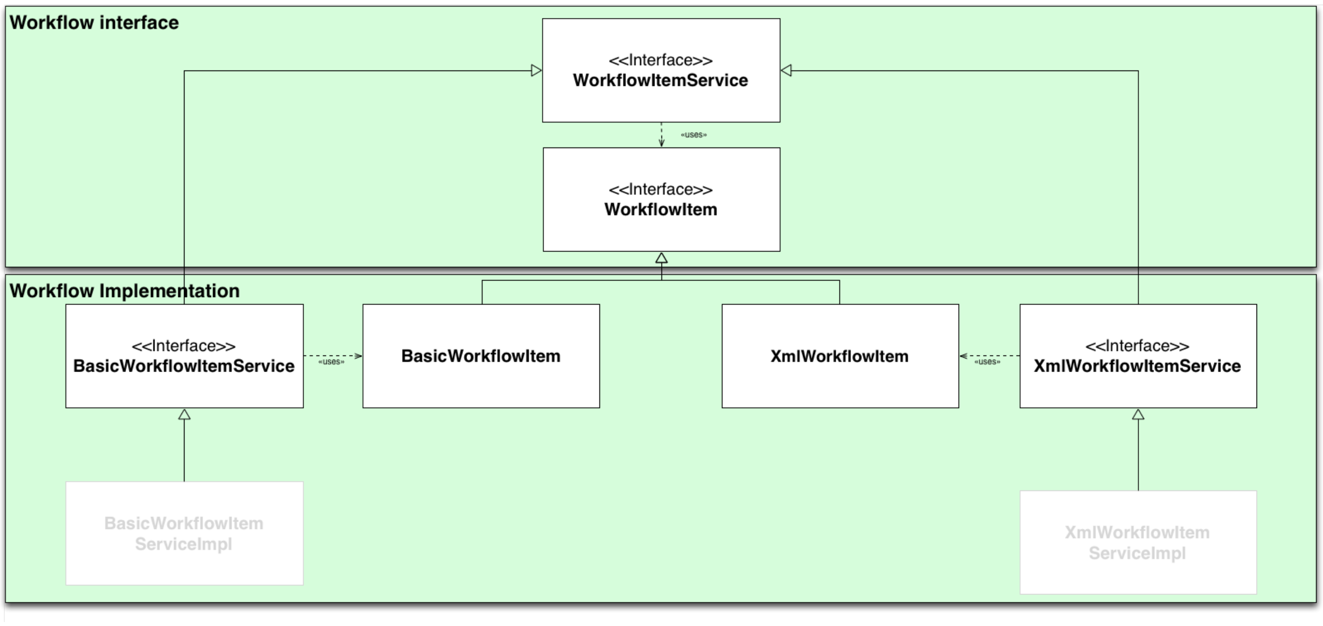
DSpace supports two workflow systems: the "original" workflow system known as workflow and the configurable XML workflow. To avoid confusion, the old WorkflowManager & WorkflowItem class have been renamed to BasicWorkflowService & BasicWorkflowItem. Below is a schematic overview of the workflow structure in the new service layout.



When a developer wants to use the workflow from a section in the code that isn't workflow specific there is no more need to add code for both workflows. The general workflowService can be used alongside with the generic WorkflowItem, the actual implementation of these classes depends purely on configuration, below are some code exerts that demonstrate this behaviour.

```
//Starting a workflow using a workspace item
WorkflowItem workflowItem;
if (useWorkflowSendEmail) {
    workflowItem = workflowService.start(context, workspaceItem);
} else {
    workflowItem = workflowService.startWithoutNotify(context, workspaceItem);
}
```

Not displayed on the graph above is the fact that both BasicWorkflowItem & XmlWorkflowItem each have a service that contains the business logic methods for these items, a schematic representation of the WorkflowItemService view displayed below.



By using the additional WorkflowItemService interface on top of our 2 workflow system the developers can interact with workflow items without even knowing the actual implementation, some code examples can be found below.

```
//Find all workflow items
List<WorkflowItem> workflowItems = workflowItemService.findAll(context);

//Find workflowItem by item & delete the workflowItem
WorkflowItem workflowItem = workflowItemService.findByItem(context, item);
workflowItemService.delete(context, workflowItem);
```

As displayed above this way a developer can delete items without ever knowing which workflow service is use. If the workflow implementation were to change it wouldn't matter to the code that is using the service.

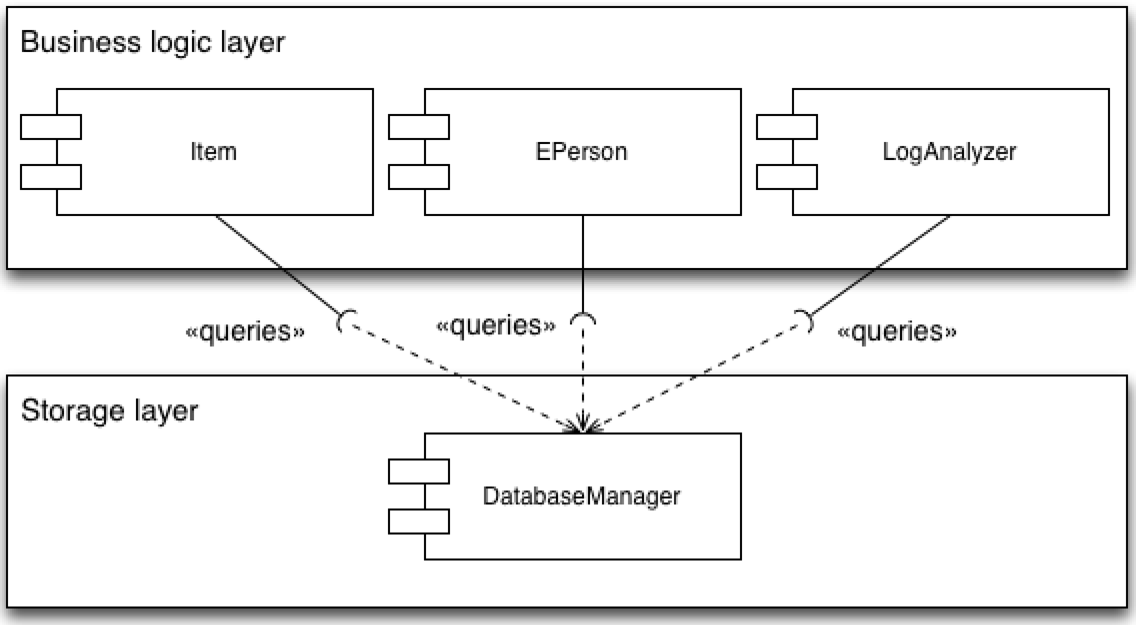
How the service api solves design issues

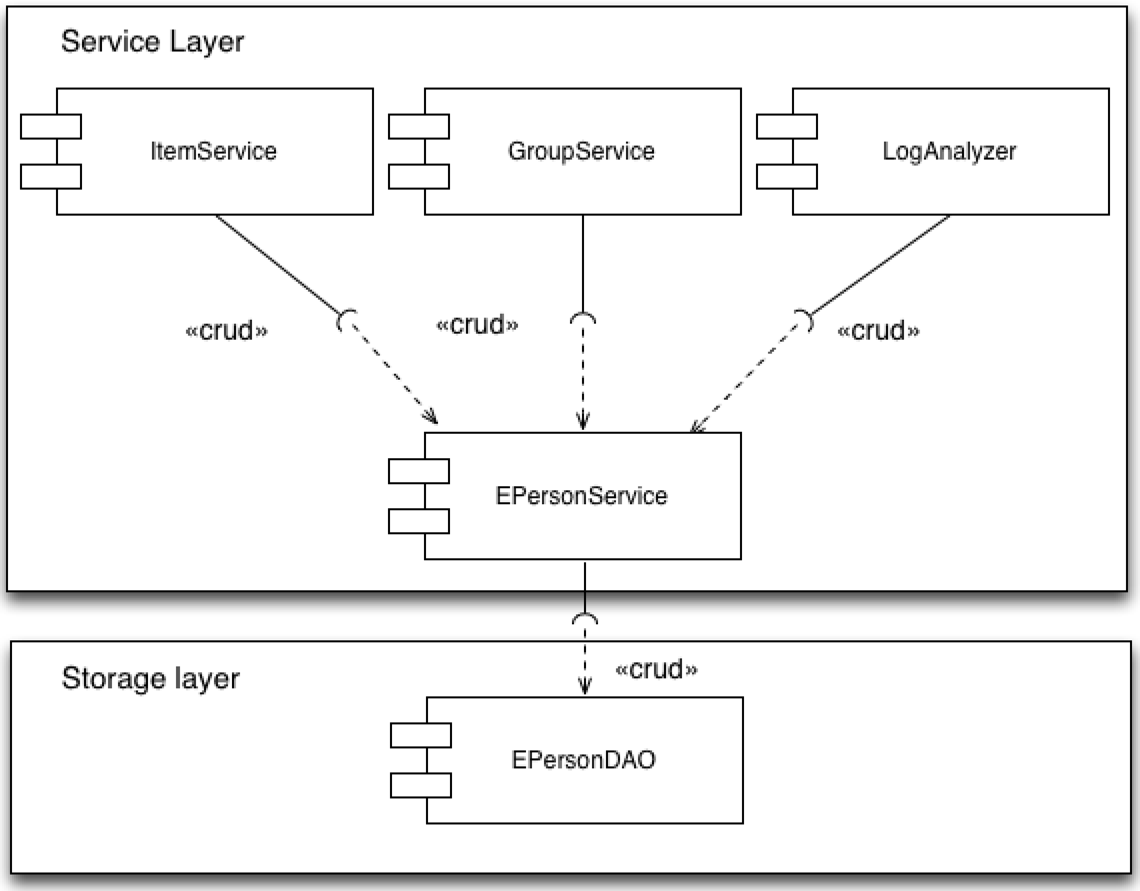
Bad distribution of responsibilities

By splitting our API in 3 layers we now have a clear separation of responsibilities. The service layer contains the business logic, the database access layer handles all database queries and the data objects represent the database tables in a clear way. Refactoring changes becomes as easy as creating a new class an extending from an existing one, instead of overwriting a single class containing thousands of lines of code to make a small adjustment.

Database layer access

As discussed above by having a single database access object linked to a single service there is only way to access a certain table and that is through a server, below is schematic representation of the before and after.





Postgres/oracle support

Hibernate will abstract the database queries away from the developer, for simple queries you can use the criteria queries (see for example: http://www.tutorialspoint.com/hibernate/hibernate_criteria_queries.htm for a quick tutorial). For more complex queries a developer can use the hibernate query language (HQL) (quick tutorial: http://www.tutorialspoint.com/hibernate/hibernate_query_language.htm), although it might take some time to get used to it, there are no longer 2 databases to test against (or dirty hacks to perform to get the same query to work for both postgres & oracle)

Alternate implementation of static flows

The alternate workflow example has been discussed at length in one of the previous chapters, see "Workflow system refactoring" chapter.

Local Modifications

By creating a new class in the additions module and then extending an existing class, methods can easily be overridden/adjusted without ever having to overwrite/alter the original class. Example of how to tackle this can be found in the tutorial sections below.

Inefficient caching mechanism

The entire DSpace context caching mechanism has been removed. This means the context class will no longer be responsible for caching certain objects. This responsibility has been entirely delegated to the hibernate framework. Hibernate allows for caching to work on a class level, so for each frequently used object a developer can easily configure it to be cached, which is much more flexible then the old DSpace way. For more information about hibernate caching please consult the hibernate documentation: http://www.tutorialspoint.com/hibernate/hibernate_caching.htm.

Current development status

What there is now

All DSpace modules have been refactored. They all compile, and all existing unit/integration tests pass.

What still needs to be done for a DSpace 6.0 release

Additional manual testing of individual interfaces (XMLUI, JSPUI, REST, SWORDv1 and v2, RDF) is necessary to ensure that all features still function properly.

As mentioned, all unit/integration tests pass. However, DSpace does not have full test coverage (in fact most tests reside in the API itself). Some basic testing of interfaces has already been performed, but more will be necessary prior to 6.0.

Long term improvements

The initial refactoring of the dspace-api is just a first step in a longer process, some areas that I believe could still use some improvement:

- At the moment only the static stateless "managers" have been refactored, the stateful classes in the dspace-api should also be refactored.
- Since the context class no longer holds the actual DB connection (a getter is still present but the connection is connected to the thread), we should consider dropping the context object. The few properties that are still linked to this object could be moved into threadLocal variables. This would save us the burden of dragging a "context" around.

Tutorials

Editing item metadata

- JUnit test to add metadata to an item:
  - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/test/java/org/dspace/content/ItemTest.java#L349
- Batch CSV import to add new items:
  - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/app/itemimport/ItemImportServiceImpl.java#L465
- Describe step in submission:
  - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/submit/step/DescribeStep.java#L680

Adding files to an item

- JUnit test to add files to an item:
  - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/test/java/org/dspace/content/BundleTest.java#L451
- Upload step in submission:
  - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/submit/step/UploadStep.java#L441
- AddBitstreamsAction, used by command line 'itemupdate' script:
  - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/app/itemupdate/AddBitstreamsAction.java#L62

Adding permissions to a bitstream

- Code to create a policy in the XMLUI:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-xmlui/src/main/java/org/dspace/app/xmlui/aspect/administrative/FlowAuthorizationUtils.java#L223
- Service method to create a policy:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/authorize/ResourcePolicyServiceImpl.java#L74

Moving an item to another collection

- JUnit test to set item collection:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/test/java/org/dspace/content/ItemTest.java#L287
- Moving an item in the XMLUI:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-xmlui/src/main/java/org/dspace/app/xmlui/aspect/administrative/FlowItemUtils.java#L379
- Service method to move an item to another collection
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/content/ItemServiceImpl.java#L755

Archive an item

- JUnit test to archive an item:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/test/java/org/dspace/content/ItemTest.java#L277
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/test/java/org/dspace/content/InstallItemTest.java#L104
- Install Item implementation:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/content/InstallItemServiceImpl.java#L189
- Reinstate an item:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/content/ItemServiceImpl.java#L509

Withdraw an item

- JUnit test to withdraw an item:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/test/java/org/dspace/content/ItemTest.java#L1117
- Withdraw an item in the XMLUI:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-xmlui/src/main/java/org/dspace/app/xmlui/aspect/administrative/FlowItemUtils.java#L284
- Service method to withdraw an item:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/content/ItemServiceImpl.java#L460

Edit collection metadata

- Edit a collection in the XMLUI:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-xmlui/src/main/java/org/dspace/app/xmlui/aspect/administrative/FlowContainerUtils.java#L102
- Service method to edit collection metadata:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/content/CollectionServiceImpl.java#L223

Creating new EPerson

- Creating new EPerson in the XMLUI:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-xmlui/src/main/java/org/dspace/app/xmlui/aspect/administrative/FlowEPersonUtils.java#L72
- CLI tool method to create a new EPerson:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/eperson/EPersonCLITool.java#L106

Adding EPerson to a Group

- JUnit test to add eperson to a group:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/test/java/org/dspace/eperson/GroupTest.java#L227
- Service method to add a new member to a group:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-api/src/main/java/org/dspace/eperson/GroupServiceImpl.java#L88
- The code where a group member is added via the XMLUI:
    - https://github.com/KevinVdV/DSpace/blob/dspace-service-api/dspace-xmlui/src/main/java/org/dspace/app/xmlui/aspect/administrative/FlowGroupUtils.java#L151

Altering an existing method in a service

TODO: Create tutorial

Adding a new method to a service

TODO: Create tutorial

Creating a new service without database access

TODO: Create tutorial

Creating a new service with database table

Introduction

In this tutorial we will be adding the possibility to add a bookmark of any given item to an existing eperson
This would lead to the functionality to be able to return to important items without having to remember where to find it, or having to bookmark in the browser itself (because this would differ from person to person, so if a computer switch takes place for some reason, those bookmarks would be lost)

Bookmarks linked to a specific EPerson also open a whole range of possibilities, such as, adding a bookmark for all users in a certain group, checking for "more" important items based on the amount of bookmarks it correlates to, etc.

Database changes

To be able to save bookmarks, a new database table is required, and for this example, we added the following db and its fields. (in a flyway file, more on that after the actual creation of the table)

```
CREATE TABLE bookmark (
 bookmark_id  INTEGER PRIMARY KEY not null,
 title    VARCHAR(50) ,
 date_created  DATE,
 creator  UUID references EPERSON(uuid) not null ,
 item   UUID references ITEM (uuid) not null
);
```

To make sure the database creation is always present (for example, if it is simply created through command line, not all developers on that projects would be certain they have the "latest" database), we can create a file with our database creation.

By creating this certain type of file in the correct place in the project, we can make certain that upon building the code, there is a check (and possible update) of the database to make sure all the required changes have taken place.

Creation of a flyway step can be done in the following way.
In the directory [dspace.src]/dspace/modules/additions/src/main/java/resources/org/dspace/storage/rdbms/sqlmigration we need to create (depending on what type of database we use) another directory with files, but for common usages and exchangeability it is recommended that for other types of database this file is also created. This way the code remains usable in the same way with different backend databases.

Because of the database type, another directory (postgres) needs to be created so that flyway knows what database it is dealing with and where to check. If you were to do this for another db, the name would differ (oracle for example).

In this directory we will create the file that contains our previously mentioned sql command to create the database (The file can contain as many sql commands as you like, separated by a semicolon)
This file will need to conform to some naming rules.

During the ant update or fresh_install, flyway will check if the database has to be created or updated and will do so accordingly, this way the correct db instance is always used.

Database object

A Database Object, is an object that represents the database table in a java class. The annotations in this class should all link to database columns from the table created above.
Below is the implementation of the current Bookmark class.

```
package org.dspace.content;
import org.dspace.eperson.EPerson;
import javax.persistence.*;
import java.util.Date;
@Entity
@Table(name="bookmark")
public class Bookmark {
   @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE ,generator="bookmark_seq")
    @SequenceGenerator(name="bookmark_seq", sequenceName="bookmark_seq", allocationSize = 1)
    @Column(name = "bookmark_id", unique = true, nullable = false, insertable = true, updatable = false)
    private int id;
 @Column(name="title")
 private String title;
 @Column(name="date_created")
 @Temporal(TemporalType.DATE)
 private Date dateCreated;
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "creator")
    private EPerson creator;
    @OneToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "item")
    private Item item;
    protected Bookmark(){
    }
 public int getId() {
  return id;
 }
 public void setId(int id) {
  this.id = id;
 }
 public String getTitle() {
  return title;
 }
 public void setTitle(String title) {
  this.title = title;
 }
 public Date getDateCreated() {
  return dateCreated;
 }
 public void setDateCreated(Date dateCreated) {
  this.dateCreated = dateCreated;
 }
 public EPerson getCreator() {
  return creator;
 }
 public void setCreator(EPerson creator) {
  this.creator = creator;
 }
 public Item getItem() {
  return item;
 }
 public void setItem(Item item) {
  this.item = item;
 }
}
```

As we see here, apart from the hibernate annotations, the class consists entirely of a constructor class (which is package protected because only the ServiceImplementation should be allowed to generate this object), and certain setters and getters.

If we take the annotations into account, we can see the database table we created in the previous step.

```
@Entity
@Table(name="bookmark")
```

Annotates that this class is an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.
The table annotation marks this class as the table with the name "bookmark".

Id and its annotations can be a bit harder to understand as there are some more "configurations" available for this

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE ,generator="bookmark_seq")
@SequenceGenerator(name="bookmark_seq", sequenceName="bookmark_seq", allocationSize = 1)
@Column(name = "bookmark_id", unique = true, nullable = false, insertable = true, updatable = false)
private int id;
```

- @Id states that this is the identifier of the table
- @GeneratedValue means that this ID will be generated by a certain generator (in this case "bookmark_seq"
- @SequenceGenerator further specifies the generator (here a sequence will be used based on the id)

- @Column states the name of the column, if it is unique or not, if it can be null or not, and things like that

The last, and probably most important part of the hibernate annotations are the following

```
@ManyToOne(fetch = FetchType.EAGER)
@OneToOne(fetch = FetchType.EAGER)
```

These all relate to the number of relations the column can have, such as, a single field can be referenced multiple times or perhaps only once, etc.
Depending on the requirements, different relations are advised.

For more information on these annotations check the hibernate & the javax documentation.

Service interface

The service part of the structure acts as a gate to use for all business logic actions that can be done on this object.
All functionality should go through here (such as create, read, update, delete and other relevant methods).
One of the key features of the service api is the usage of interfaces and their implementation to be able to simply add another implementation without having to alter the original one. The only thing to note here is that the BookmarkService is in the service package, while BookmarkServiceImpl as well as the Bookmark Class itself is in the content package).

We start of by creating an interface which will receive it's implementation later on in this document.

Bookmark Service Interface

```
package org.dspace.content.bookmark.service;

import org.dspace.content.bookmark.Bookmark;
import org.dspace.content.Item;
import org.dspace.core.Context;
import org.dspace.eperson.EPerson;
import java.sql.SQLException;
import java.util.List;

public interface BookmarkService {
    public Bookmark create(Context context) throws SQLException;

    public Bookmark read(Context context, int id) throws SQLException;

    public void update(Context context, Bookmark bookmark) throws SQLException;

    public void delete(Context context, Bookmark bookmark) throws SQLException;

    public List<Bookmark> findAll(Context context) throws SQLException;

    public List<Bookmark> findByEperson(Context context, EPerson ePerson) throws SQLException;

    public List<Bookmark> findByItem(Context context, Item item) throws SQLException;
}
```

Data Access Object (DAO) interface

Since the service interface implementation cannot be responsible for the database queries themselves we need to create an additional interface that will be a gateway to our database access. An interface for a certain DAO could look something like this.

Bookmark DAO Interface

```
package org.dspace.content.dao;

import org.dspace.content.Bookmark;
import org.dspace.content.Item;
import org.dspace.core.Context;
import org.dspace.core.GenericDAO;
import org.dspace.eperson.EPerson;
import java.sql.SQLException;
import java.util.List;

public interface BookmarkDAO extends GenericDAO<Bookmark> {

    public List<Bookmark> findBookmarksByEPerson(Context context, EPerson ep) throws SQLException;

    public List<Bookmark> findBookmarksByItem(Context context, Item item) throws SQLException;
}
```

Notice that this class extends the GenericDAO class with a Bookmark. This GenericDAO already has most of the standard functionality a class might need for database access, such as a create, save, delete, findyId, findAll, etc. This allows us to use these implementations, and focus on our additional methods. The service doesn't need to use all the actions of the GenericDAO but it ensures that this DAO doesn't need to declare them.

Factory

Factories ensure that we don't have to keep track of our service names & disregard the need to always use new DSpace().getServiceManager().getServiceByName() method. Creating a new factory is done in 3 steps, all of which are briefly explained below.

Create new service factory

```
package org.dspace.content.bookmark.factory;

import org.dspace.content.bookmark.service.BookmarkService;
import org.dspace.utils.DSpace;

/**
 * Abstract factory to get services for the content.bookmark package, use BookmarkServiceFactory.getInstance()
to retrieve an implementation
 *
 */
public abstract class BookmarkServiceFactory {

    public abstract BookmarkService getBookmarkService();

    public static BookmarkServiceFactory getInstance(){
        return new DSpace().getServiceManager().getServiceByName("bookmarkServiceFactory",
BookmarkServiceFactory.class);
    }
}
```

A service factory only has one mandatory method, the "getInstance()" which returns an instantiated factory that can then be used. The other methods in the factory should be abstract, our factory implementation will provide the necessary details.

Create a new implementation of the service factory

```
package org.dspace.content.bookmark.factory;

import org.dspace.content.bookmark.service.BookmarkService;
import org.springframework.beans.factory.annotation.Autowired;

public class BookmarkServiceFactoryImpl extends BookmarkServiceFactory {

    @Autowired(required = true)
    private BookmarkService bookmarkService;

    @Override
    public BookmarkService getBookmarkService() {
        return bookmarkService;
    }
}
```

A factory implementation will come down to auto wiring the service & implementing the getter, should additional business logic be required to determine which service should be used when, this class is the place to do it.

Configure the service factory in spring

In step 1 we use a getInstance() to retrieve a service factory, in order to use it we still need to configure it. It is recommended to place the factory configuration in the [dspace.dir]/config/spring/api/core-factory-services.xml file. This would result in the following configuration:

```
 <bean id="bookmarkServiceFactory" class="org.dspace.content.bookmark.factory.BookmarkServiceFactoryImpl"/>
```

Once this is done the following code can be used to retrieve an instance of our service:

```
ContentServiceFactory.getInstance().getBookmarkService();
```

Service implementation

The Service implementation lets us fill in the required behavior and business logic we want to have. Below is an example implementation of our BookmarkService.

```java
package org.dspace.content.bookmark;

import org.dspace.content.Item;
import org.dspace.content.bookmark.dao.BookmarkDAO;
import org.dspace.content.bookmark.service.BookmarkService;
import org.dspace.core.Context;
import org.dspace.eperson.EPerson;
import org.springframework.beans.factory.annotation.Autowired;

import java.sql.SQLException;
import java.util.List;

public class BookmarkServiceImpl implements BookmarkService {
 @Autowired(required = true)
 protected BookmarkDAO bookmarkDAO;

 protected BookmarkServiceImpl() {

 }

 @Override
 public Bookmark create(Context context) throws SQLException {
  return bookmarkDAO.create(context, new Bookmark());
 }

    @Override
 public Bookmark read(Context context, int id) throws SQLException {
  return bookmarkDAO.findByID(context, Bookmark.class, id);
 }

 @Override
 public void update(Context context, Bookmark bookmark) throws SQLException {
        if(context.getCurrentUser().equals(bookmark.getCreator())){
            bookmarkDAO.save(context, bookmark);
        }
 }

 @Override
 public void delete(Context context, Bookmark bookmark) throws SQLException {
        if(context.getCurrentUser().equals(bookmark.getCreator())) {
            bookmarkDAO.delete(context, bookmark);
        }
 }

 @Override
 public List<Bookmark> findAll(Context context) throws SQLException {
  return bookmarkDAO.findAll(context, Bookmark.class);
 }

    @Override
    public List<Bookmark> findByEperson(Context context, EPerson ePerson) throws SQLException {
        return bookmarkDAO.findBookmarksByEPerson(context,ePerson);
    }

    @Override
    public List<Bookmark> findByItem(Context context, Item item) throws SQLException {
        return bookmarkDAO.findBookmarksByItem(context,item);
    }
}
```

As seen in the example above, the Implementation of the BookmarkService simply links the functionality "deeper" in the structure.
In this particular class, we can see that the BookmarkDAO class is used to create, read, update, delete, etc the Bookmark object.
Also note that this is the ONLY place where a new Bookmark Object will be created, this being that the Bookmark class constructor is package protected and in the same package as this class.
One thing that has to be extremely clear is that the DAO is ONLY the link between the database and the code, NO business logic should be present there.
So for example, if only the creator of a bookmark is allowed to delete/update it, this should be handled in the service as it's possible that another service doesn't require this.

Configure the Service implementation in spring

In order to ensure that our factory retrieves the proper service some spring configuration is also required. Just add the service class as a bean in the [dspace.dir]/config/spring/api/core-services.xml file as displayed below:

```
<bean class="org.dspace.content.BookmarkServiceImpl"/>
```

Data Access Object (DAO) implementation

The DAO implementation lets us fill in the database queries that we require. Below is an example implementation of our BookmarkDAO.

```
package org.dspace.content.bookmark.dao.impl;

import org.dspace.content.bookmark.Bookmark;
import org.dspace.content.Item;
import org.dspace.content.bookmark.dao.BookmarkDAO;
import org.dspace.core.AbstractHibernateDAO;
import org.dspace.core.Context;
import org.dspace.eperson.EPerson;
import org.hibernate.Criteria;
import org.hibernate.Query;
import org.hibernate.criterion.Restrictions;

import java.sql.SQLException;
import java.util.List;

public class BookmarkDAOImpl extends AbstractHibernateDAO<Bookmark> implements BookmarkDAO {

    @Override
    public List<Bookmark> findBookmarksByEPerson(Context context, EPerson ep) throws SQLException {
        Criteria criteria = createCriteria(context,Bookmark.class);
        criteria.add(Restrictions.and(Restrictions.eq("creator",ep)));
        return list(criteria);
    }

    @Override
    public List<Bookmark> findBookmarksByItem(Context context, Item item) throws SQLException {
        Query query = createQuery(context, "from Bookmark where item = :item order by date_created");
        query.setParameter("item", item);
        return list(query);
    }

}
```

In the example above 2 different types of queries are used, the criteria & the HQL query. This documentation doesn't go into the details of how these work, for more information check the hibernate docs. What is different from the hibernate defaults are the "list", "createQuery" & "createCriteria" methods, these are helper methods provided by the AbstractHibernateDAO class. These methods are present to prevent code duplication, so checkout the class to get a list of all available helper methods.

Configure the Service implementation in spring

In order to ensure that our factory retrieves the proper service some spring configuration is also required. Just add the service class as a bean in the [dspace.dir]/config/spring/api/core-dao-services.xml file as displayed below:

```
<bean class="org.dspace.content.BookmarkServiceImpl"/>
```

Example code

Imagine the possibility to create bookmarks based on a person's mail, handles of items and given titles. Below is an example class that can be run from the command line interface that will create & display bookmarks using the service we created above.

```
package org.dspace.util;

import org.apache.commons.cli.*;
import org.dspace.content.Bookmark;
import org.dspace.content.Item;
import org.dspace.content.factory.ContentServiceFactory;
import org.dspace.content.service.BookmarkService;
import org.dspace.content.service.ItemService;
import org.dspace.core.Context;
import org.dspace.eperson.EPerson;
import org.dspace.eperson.factory.EPersonServiceFactory;
import org.dspace.eperson.service.EPersonService;
import org.dspace.handle.factory.HandleServiceFactory;
import org.dspace.handle.service.HandleService;

import java.sql.SQLException;
import java.util.*;


public class BookmarkUpdater {
    protected BookmarkService bookmarkService;
    protected EPersonService ePersonService;
    protected ItemService itemService;
    protected HandleService handleService;

    protected BookmarkUpdater() {
        bookmarkService = ContentServiceFactory.getInstance().getBookmarkService();
        ePersonService = EPersonServiceFactory.getInstance().getEPersonService();
        itemService = ContentServiceFactory.getInstance().getItemService();
        handleService = HandleServiceFactory.getInstance().getHandleService();

    }

    public void doUpdate(String epersonMail, Map<String, String> handlesAndTitles) throws SQLException {
        Context ctx = new Context();
        EPerson ePerson = ePersonService.findByEmail(ctx, epersonMail);
        for (String key : handlesAndTitles.keySet()) {
            // Don't add invalid objects
            if (handleService.resolveToObject(ctx, key) != null) {
                createBookmark(ctx, ePerson, key, handlesAndTitles.get(key));
            }
        }
        ctx.complete();
    }

    private void createBookmark(Context ctx, EPerson ePerson, String handle, String title) throws SQLException {
        Bookmark bookmark = bookmarkService.create(ctx);
        bookmark.setDateCreated(new Date());
        bookmark.setCreator(ePerson);
        bookmark.setItem((Item) handleService.resolveToObject(ctx, handle));
        bookmark.setTitle(title);
    }

    public static void main(String... args) throws ParseException {

        BookmarkUpdater bmu = new BookmarkUpdater();
        CommandLineParser parser = new PosixParser();
        Map<String, String> handlesAndTitles = new HashMap<>();
        Options options = createOptions();

        CommandLine line = parser.parse(options, args);

        printOptionsHelp(options, line);

        try {

            if (line.hasOption("f")) {
                if (line.hasOption("i")) {
                    bmu.printBookmarksBasedOnItem(line.getOptionValue("i"));
                }
            }
            String epersonMail = line.getOptionValue("e");
            addHandlesAndTitles(handlesAndTitles, line);
            bmu.doUpdate(epersonMail, handlesAndTitles);
            if (line.hasOption("p")) {
                bmu.printEpersonsBookmarks(epersonMail);
            }
        } catch (SQLException sqle) {
            System.err.println(sqle.getLocalizedMessage());
            sqle.printStackTrace();
        }
    }

    private static void printOptionsHelp(Options options, CommandLine line) {
        if (line.hasOption('h')) {
            HelpFormatter myhelp = new HelpFormatter();
            myhelp.printHelp("Usages : \n", options);
            System.out.println("\nAdd a single bookmark based on eperson,handle and possibly a title: org.
dspace.util.BookmarkUpdater -e atmirenv@gmail.com -h 123456789/4 -t 'A title'");
            System.out.println("\nAdd multiple bookmarks to a provided eperson: org.dspace.util.BookmarkUpdater
-e atmirenv@gmail.com -m");
            System.out.println("\nAdding the -p option will show all the bookmarks currently associated with
the given eperson");
            System.out.println("\nIf the f option has been provided (as well as a handle (i), all bookmarks
with this given item will be shown");
            System.out.println("\nIf no options are provided (apart from the required eperson), a fallback to
the addition of multiple bookmarks will be used");
            System.exit(0);
        }
    }

    private static void addHandlesAndTitles(Map<String, String> handlesAndTitles, CommandLine line) {
        String title;
        // If the "multiple" option has been enabled, keep asking user for input until he types stop
        // When no handle is supplied, default to this behaviour as well
        if (line.hasOption("m") || !line.hasOption("i")) {
            System.out.println("Enter valid handles(invalid handles will be skipped)\nYou can stop adding
bookmarks by typing stop");
            Scanner scanner = new Scanner(System.in);
            String handle = scanner.nextLine();
            while (!handle.equals("stop")) {
                if (handlesAndTitles.containsKey(handle)) {
                    System.out.println("This item has already been bookmarked by this user");
                } else {
                    Scanner titleScanner = new Scanner(System.in);
                    System.out.println("Enter a title for this bookmark");
                    title = titleScanner.nextLine();
                    handlesAndTitles.put(handle, title);
                }
                System.out.println("Enter another handle");
                handle = scanner.nextLine();
            }

        } else {
            // The user has provided a single handle to bookmark
            if (line.hasOption("i")) {
                if (line.hasOption("t")) {
                    title = line.getOptionValue("t");
                } else {
                    title = "No title provided for the bookmark";
                }
                handlesAndTitles.put(line.getOptionValue("i"), title);
            }
        }
    }

    private static Options createOptions() {
        Options options = new Options();
        Option epers = new Option("e", "eperson", true, "The eperson's email adress");
        epers.setRequired(true);
        options.addOption(epers);
        options.addOption("i", "itemhandle", true, "The handle of an item to add");
        options.addOption("m", "multiple", false, "Create multiple bookmarks");
        options.addOption("t", "title", true, "Enter a title");
        options.addOption("h", "help", false, "help");
        options.addOption("p", "print", false, "Print the bookmarks currently connected to a given eperson");
        options.addOption("f", "findbyitem", false, "Print the bookmarks currently connected to a given
itemhandle");

        return options;
    }

    public void printEpersonsBookmarks(String epersonMail) throws SQLException {
        Context ctx = new Context();

        EPerson ePerson = ePersonService.findByEmail(ctx, epersonMail);
        List<Bookmark> bookMarksByEperson = bookmarkService.findByEperson(ctx, ePerson);
        printBookmarks(bookMarksByEperson);

        ctx.complete();
    }

    public void printBookmarksBasedOnItem(String handle) throws SQLException {
        Context ctx = new Context();

        Item item = (Item) handleService.resolveToObject(ctx, handle);
        if (item != null) {
            List<Bookmark> bookMarksByEperson = bookmarkService.findByItem(ctx, item);
            printBookmarks(bookMarksByEperson);
        }

        ctx.complete();
    }

    private void printBookmarks(List<Bookmark> bookMarksByEperson) {
        for (Bookmark b : bookMarksByEperson) {
            System.out.println("Generated UUI :" + b.getId());
            System.out.println("Title :" + b.getTitle());
            System.out.println("Date of creation :" + b.getDateCreated());
            System.out.println("Creator : " + b.getCreator().getFullName());
            System.out.println("Item : " + ((b.getItem() != null) ? b.getItem().getName() : "No item provided
for this bookmark"));
        }
    }
}
```