GSOC 2010 Unit Tests - Technical documentation

Introduction

This document describes the technical aspects of the testing integrated into Dspace. In it we describe the tools used as well as how to use them and the solutions applied to some issues found during development. It's intended to serve as a reference for the community so more test cases can be created.

Issues Found

During implementation we found several issues, which are described in this section, along the solution implemented to work around them.

Multiple Maven projects

DSpace is implemented as a set of Maven 2 projects with a parent-child relationship and a module that manages the final assembly of the project. Due to the specifics of DSpace (database dependency, file system dependency) we need to set up the test environment before running any tests. While the fragmentation in projects of DSpace is good design and desirable, that means we have to replicate the configuration settings for each project, making it much less maintainable.

There is another issue related to the way Maven works. Maven defines a type of package for each project (jar, war or pom). Pom projects can contain subprojects, but their lifecycle skips all the test steps. This means that even if a Pom project would be ideal to place the tests, they would be not be run and we can't force Maven to run them by any means.

The perfect solution would be to run the tests in the Assembly project, but due to the mentioned limitations (it is a Pom project) this can't work.

To solve this issue we have created a new project, dspace-test, which will contain only unit tests. It has dependencies on the projects being tested and all the settings required to initialize DSpace. All tests must be added to this project, as it is the only one with the proper dependencies.

File system dependency

We found that some methods and/or classes have a dependency on the file system of DSpace. This means we have to replicate the file system, including some configuration files, before being able to run the tests.

The main issue is that the existing files we need are in the Assembly project, but we can't run the tests there. Also, we can't use an assembly to copy the files as the test phase runs before any packaging or install phase in Maven.

The solution has been to duplicate the file system in the dspace-test project. This replica is copied to the temp folder (a folder designed by the tester via a configuration file) before launching the tests. Once the tests finish, the files are removed. This is not an ideal solution as requires tester to duplicate files, but is a workaround while we find a definite solution.

The files are stored under the folder *dspaceFolder* in the resources folder. All the contents of this folder will be copied to the temporal area. The file *test-config.properties* contains settings to specify the temporal folder and other test values.

File configuration dependency

Dspace heavily depends on the file *dspace.cfg* for its configuration. For testing purposes we have crafted a test version of this file, available in the resources folder of *dspace-test*. This file is loaded during setup instead of the default dspace.cfg so the environment is set for unit testing.

As the assembly process is run later than the test goal in Maven, we can't use external profiles to replace values. This means the values in this file are hard-coded and might need to be changed for a specific system. The way it is set up by default, it should work on all *nix systems as it sets the */tmp/* folder as the temporal test folder. If this has to be changed, the file *test-config.properties* will also need to be updated.

Database dependency

We found that many classes have a direct dependency to the database. Specifically there is a huge coupling with both Oracle and PostgreSQL, with many pieces of code being executed or not depending on the database used. Mocking the connections is not easy due to the heavy use of default-access constructors and relations between classes that are not following Demeter's Law.This means we need a database to run the tests, something not really desirable but required.

While the perfect solution would be to migrate DSpace to an ORM like Hibernate, there is not time to do so in this project, and this would be too much of a change to add to the source. The decision has been made to use an in-memory database where the DSpace database will be recreated for the purpose of unit testing. Once the tests are finished, the database will be wiped and closed.

Taking in account the coupling with Oracle/PostgreSQL and the need to recreate the database, the solution has been to create a mock of the *DatabaseManager* class. This mock implements the exact same functionality of the base class (being a complete copy) and adds some initialization code to reconstruct the tables and default contents.

The database used is H2 . The information is stored in a copy of the *database_schema.sql* file for PostgreSQL with the following modifications:

- removal of the function that provides the next value of a sequence
- removal of clause "WITH TIME ZONE" from TIMESTAMP values
- removal of DEFAULT NEXTVAL('<seq>') constructs due to incompatibility. DatabaseManager has been changed to add the proper ID to the column. Proposed to change the affected valued to IDENTITY values, that include autoincrement.
- removal of UNIQUE constructs due to incompatibility. Tests will need to verify uniqueness
- replaced BIGSERIAL by BIGINT
- replacing getnextid for NEXTVAL on an INSERT for epersongroup
- due to the parsing process some spaces have been added at the start of some lines to avoid syntax errors

Due to H2 requiring the column names in capital letters the database is defined as an Oracle database for DSpace (db.name) and the Oracle compatibility mode for H2 has been enabled.

The code in the mock DatabaseManager has been changed so the queries are compatible with H2. The changes are minimal as H2 is mostly compatible with PostgreSQL and Oracle.

As a note, the usage of a DDL language via DDLUtils has been tested and discarded due to multiple issues. The code base of DDL Utils is ancient, and not compatible with H2. This required us to use HSQLDB, which in turn required us to change some tables definitions due to syntax incompatibilities. Also, we discovered DDL Utils wasn't generating a correct DDL file, not providing relevant meta-information like which column of a table was a primary key or the existing sequences. Due to the reliance of DatabaseManager on this meta-information, some methods were broken, giving wrong values. It seems that more recent code is available from the project SVN, but this code can't be recovered from Maven repositories, which would make much more cumbersome the usage of unit testing in DSpace as the developer would be required to download the code, compile it and store it in the local repository before being able to do a test. A lot of effort has been put to use the DDL, but in the end we feel using the database_schema.sql file is better.

Different tests

In this project we want to enable unit tests, integration tests and functional tests for DSpace. Maven 2 has a non-modifiable life-cycle that doesn't allow us to run tests once the project has been packaged.This same life-cycle doesn't allow us to launch an embedded server like Jetty to run the functional tests.

The solution to this is to create 2 infrastructures, one for the unit and integration tests and one for the functional tests. Unit and integration tests will be run by Maven using the Surefire plugin. Functional tests will be run once the program has been build from an Ant task. This will allow us to launch an embedded server to run the tests.

This option is not optimal, but due to the limitations imposed by DSpace system and Maven we have not find a better solution. Any proposals are appreciated.

The unit and integration tests solution has been implemented in the dspace-test project.

The functional tests implementation is being done.

Structural Issues

During the development the following issues have been detected in the code, which make Unit Testing harder and impact the maintainability of the code:

* Hidden dependencies. Many objects require other objects (like DatabaseManager) but the dependency is never explicitly declared or set. These dependencies should be fulfilled as parameters in the constructors or factory methods.

* Hidden constructors. It would be advisable to have public constructors in the objects and provide a Factory class that manages the instantiation of all required entities. This would facilitate testing and provide a separation of concerns, as the inclusion of the factory methods inside objects usually adds hidden dependencies (see previous point).

Refactoring would be required to fix these issues.

Development Issues

During development some issues have arised. These issues need to be solved for the unit tests to be viable.

Warnings to be taken in account:

- Unit tests may be faulty due to misunderstanding of the source code, a revision is required to ensure they behave as expected
- Unit tests may be incomplete, missing some paths of the existing code
- Due to the tight dependencies between some classes some methods can't be tested completely. In these cases more benefit can be obtained from integration tests than from unit tests
- For the aforementioned reasons, a revision (peer-review) is required to ensure the unit tests behave as expected and they are reliable
- The unit tests may lack tests for some edge cases. It's in the nature of Unit Tests to evolve as new bugs are found, so they should be added as they are detected (via peer-review or via bug reports)

Issues found:

- A mock of BrowseCreateDAOOracle has been done due to an incompatibility between H2 and the "upper" function call. This will affect tests related to case sensitivity in indexes.
- Many objects (like SupervisedItem) lack a proper definition of the "equals" method, which makes comparison between objects and unit testing harder
- Update method of many objects doesn't provide any feedback, we can only test if it raises an exception or not, but we can't be 100% sure if it worked
- Many objects have methods to change the policies related to the object or children objects (like Item), it would be good to have some methods to retrieve these policies also in the same object (code coherence)
- There are some inconsistencies in the calls to certain methods. For example getName returns an empty String in a Collection where the name is not set, but a null in an Item without name
- DCDate: the tests raise many errors. I can't be sure if it's due to misunderstanding of the purpose of the methods or due to faulty implementation (probably the previous). In some cases extra encapsulation of the internals of the class would be advisable, to hide the complexities of the Calendars (months starting by 0, etc)
- The Authorization system gets a bit confusing. We have AuthorizationManager, AuthorizationUtils, methods that raise exceptions and methods that return booleans. Given the number of checks we have to do for permissions, and that some classes call methods that require extra permissions not declared or visible at first, this makes creation of tests (and usage of the API) a bit complex. I know we can ignore all permissions via context (turning on and off authorizations) but usually we don't want that
- Community: set logo checks for authorization, but set metadata doesn't. It's in purpose?
- Collection: methods create and delete don't check for authorization
- Item: there is no authorization check for changing policies, no need to be an administrator
- ItemIterator: it uses ArrayLists in the methods instead of List
- ItemIterator: we can't verify if the Iterator has been closed
- Metadata classes: usually most classes have a static method to create a new instance of the class. Instead, for the metadata classes (Schema, Field and Value) the method create is part of the object, thus requiring you to first create an instance via *new* and then calling *create*. This should be changed to follow the convention established in other objects (or the other objects should be amended to behave like the Metadata classes)
- Site: this class extends DSpaceObject. With it being a Singleton, it creates potential problems, for example when we use DSpaceObject methods to store details in the object. It's this relation necessary?

Fixes done:

- Bitstream: added an "isDeleted" method to verify if a bitstream has been deleted
- Bundle: added methods to check the policies of bundle and its bitstreams
- Collection: just a comment: delete requires authorization to remove the template Item and write, not to remove. Is that correct?
- Community: when a community is created with a parent, it's added as a child community immediately
- DCLanguage: added checks for null name in languages
- FormatIdentifier: fixed the check for filename == null in guessFormat
- SiteTest: the test is in the abstract DSpaceObjectTest so I've make it inherit AbstractUnitTest. I see the class has almost no usage so we could remove the inheritance from DSpaceObject, but I'm not sure if to do this. It's something that we should ask the developers?
- Several equals and hashCode methods added for other issues in tests

Pending of a review by a DSpace developer:

- DCDate: Here many tests fail because I'm not sure of the purpose of the class. I would expect it to hide the implementation of Calendar (with all those things like months starting by 0 and other odd stuff) so it's easier to use, but it seems that's not the case...

- Bitstream: added an "isDeleted" method to verify if a bitstream has been deleted
- Bundle: added methods to check the policies of bundle and its bitstreams
- Collection: just a comment: delete requires authorization to remove the template Item and write, not to remove. Is that correct?
- Community: when a community is created with a parent, it's added as a child community immediately

- DCLanguage: added checks for null name in languages
- FormatIdentifier: fixed the check for filename == null in guessFormat
- SiteTest: the test is in the abstract DSpaceObjectTest so I've make it inherit AbstractUnitTest. I see the class has almost no usage so we could remove the inheritance from DSpaceObject, but I'm not sure if to do this. It's something that we should ask the developers?
- Several equals and hashCode methods added for other issues in tests

Proposals:

To solve the previous issues, some proposals are done:

- Database dependency causes too many issues, making unit testing much harder and increasing the complexity of the code. Refactoring to a database-neutral system should be a priority
- A release could be done (1.8?) centered on cleaning code, improving stability and coherency and refactoring unit tests, as well as replacing the database system. No new functionalities. This would make future work much easier.

Dependencies

There is a set of tools used by all the tests. These tools will be described in this section.

Maven

The build tool for DSpace, Maven, will also be used to run the tests. For this we will use the Surefire plugin, which allows us to launch automatically tests included in the "test" folder of the project. We also include the Surefire-reports plugin in case you are not using a Continous Integration environment that can read the output and generate the reports.

The plugin has been configured to ignore test files whose name starts with "Abstract", that way we can create a hierarchy of classes and group common elements to various tests (like certain mocks or configuration settings) in a parent class.

Tests in Maven are usually added into *src/test*, like in *src/test/java/<package>* with resources at *src/test/resources*.

To run the tests execute:

```
mvn test
```

The tests will also be run during a normal Maven build cycle. To skip the tests, run Maven like:

```
mvn package -Dmaven.test.skip=true
```

By default we will disable running the tests, as they might slow the compilation cycle for developers. They can be activated using the command

```
mvn package -Dmaven.test.skip=false
```

or by changing the property "*activeByDefault*" at the corresponding profile (*skiptests*) in the main *pom.xml* file, at the root of the project.

JUnit

JUnit is a testing framework for Java applications. It was one of the first testing frameworks for Java and it's a widespread use in the community. The framework simplifies the development of unit tests and the current IDE's make even easier building those tests from existing classes and running them.

Junit 4.8.1 is added as a dependency in the parent project. The dependency needs to be propagated to the subprojects that contain tests to be run.

As of JUnit 4.4, Harmcrest is included. Harmcrest is a library of matcher objects that facilitate the validation of conditions in the tests.

JMockit

JMockit is popular and powerful mocking framework. Unlike other mocking frameworks it can mock final classes and methods, static methods, constructors and other code fragments that can't be mocked using other frameworks.

JMockit 0.998 has been added to the project to provide a mocking framework to the tests.

ContiPerf

ContiPerf is a lightweight testing utility that enables the user to easily leverage JUnit 4 test cases as performance tests e.g. for continuous performance testing.

The project makes use of ContiPerf 1.06.

H2

H2 is an in-memory database that has been used

The project makes use of H2 version 1.2.137

Unit Tests Implementation

These are tests which test just how one object works. Typically test each method on an object for expected output in several situations. They are executed exclusively at the API level.

We can consider two types of classes when developing the unit tests: classes which have a dependency on the database and classes that don't. The classes that don't can be tested easily, using standard procedures and tests. Our main problem are classes tightly coupled with the database and its helper objects, like BitstreamFormat or the classes that inherit from DSpaceObject. To run the unit tests we need a database but we don't want to set up a standard PostgreSQL instance. Our decision is to use an in-memory database that will be used to emulate PostgreSQL.

To achieve this we mock DatabaseManager and we replace the connector to point to our in-memory database. In this class we also initialise the replica with the proper data.

Structure

Due to the Dspace Maven structure discussed in previous sections, all the tests belonging to any module (dspace-api, dspace-xmlui-api, etc) must be stored in the module *dspace-test*. This module enables us to apply common configuration, required by all tests, in a single area thus avoiding duplication of code. Related to this point is the requirement for Dspace to run using a database and a certain file system structure. We have created a base class that initializes this structure via a in-memory database (using H2) and a temporary copy of the required file system.

The described base class is called "AbstractUnitTest". This class contains a series of mocks and references which are necessary to run the tests in DSpace, like mocks of the DatabaseManager object. All Unit Tests should inherit this class, located under the package "org.dspace" in the test folder of *dspace-test*. There is an exception with classes that originally inherit *DSpaceObject, i*ts tests should inherit *AbstractDSpaceObjectTest* class.

Several mocks are used in the tests. The more relevant ones are:

- MockDatabaseManager: a mock of the database manager that launches H2 instead of PostgreSQL/Oracle and creates the basic structure of tables for DSpace in memory
- MockBrowseCreateDAOOracle: due to the strong link between DSpace and the databases, there are some classes that have specific implementations if we are using Oracle or PostgreSQL, like this one. In this case we've had to create a mock class that overrides the functionality of MockBrowseCreateDAOOracle so we are able to run the Browse related tests.

You may need to create new mocks to be able to test certain areas of code. Creation of the Mock goes beyond the scope of this document, but you can see the mentioned classes as an example. BAsically it consists on adding annotations to a copy of the existing class to indicate a method is a mock of the original implementation and modifying the code as required for our tests.

Limitations

The solution to copy the file system is not a very elegant one, so we appreciate any insight that can help us to replicate the required files appropriately.

The fact that we load the tests configuration from a dspace-test.cfg file means we are only testing the classes against a specific set of configurations. We probably would like to have tests that runs with multiple settings for the specific piece of code we are testing. This will require some extra classes to modify the configuration system and the way this is accessed by DSpace.

How to build new tests

To build a new Unit Test, create the corresponding class in the project *dspace-test*, under the *test* folder, in the package where the original class belongs. Tests for all the projects (dspace-api, dspace-jspui-api, etc) are stored in this project, to avoid duplication of code. Name the class following the format *<OriginalClass>Test.java*.

There are some common imports and structure, you can use the following code as a template:

```
//Add DSpace licensing here at the top!
package org.dspace.content;

import java.sql.SQLException;
import org.dspace.core.Context;
import org.junit.*;
import static org.junit.Assert.* ;
import static org.hamcrest.CoreMatchers.*;
import mockit.*;
```

```
import org.apache.log4j.Logger;
import org.dspace.core.Constants;

/**
 * Unit Tests for class <OriginalClass>Test
 * @author you name
 */
public class <OriginalClass>Test extends AbstractUnitTest
{

    /** log4j category */
    private static final Logger log = Logger.getLogger(<OriginalClass>Test.class);

    /**
     * <OriginalClass> instance for the tests
     */
    private <OriginalClass> c;

    /**
     * This method will be run before every test as per @Before. It will
     * initialize resources required for the tests.
     *
     * Other methods can be annotated with @Before here or in subclasses
     * but no execution order is guaranteed
     */
    @Before
    @Override
    public void init()
    {
        super.init();
        try
        {
            //we have to create a new community in the database
            context.turnOffAuthorisationSystem();
            this.c = <OriginalClass>.create(null, context);


            //we need to commit the changes so we don't block the table for testing
            context.restoreAuthSystemState();
            context.commit();
        }
        catch (AuthorizeException ex)
        {
            log.error("Authorization Error in init", ex);
            fail("Authorization Error in init");
        }
        catch (SQLException ex)
        {
            log.error("SQL Error in init", ex);
            fail("SQL Error in init");
        }
    }

    /**
     * This method will be run after every test as per @After. It will
     * clean resources initialized by the @Before methods.
     *
     * Other methods can be annotated with @After here or in subclasses
     * but no execution order is guaranteed
     */
    @After
    @Override
    public void destroy()
    {
        c = null;
        super.destroy();
    }

    /**
     * Test of XXXX method, of class <OriginalClass>
     */
```

```
    @Test
    public void testXXXX() throws Exception
    {
        int id = c.getID();
        <OriginalClass> found =  <OriginalClass>.find(context, id);
        assertThat("testXXXX 0", found, notNullValue());
        assertThat("testXXXX 1", found.getID(), equalTo(id));
        assertThat("testXXXX 2", found.getName(), equalTo(""));
    }

   [... more tests ...]
}
```

The sample code contains common imports for the tests and common structure (*init* and *destroy* methods as well as the log). You should add any initialization required for the test in the *init* method, and free the resources in the *destroy* method.

The sample test shows the usage of the *assertThat* clause. This clause (more information in JUnit help) allows you to check for condition that, if not true, will cause the test to fail. We name every condition via a simple schema (method name plus an integer indicating order) as the first parameter. This allows you to identify which specific assert if failing whenever a test returns an error.

Please be aware methods *init* and *destroy* will run once per test, which means that if you create a new instance every time you run *init*, you may end up with several instances in the database. This can be confusing when implementing tests, specially when using methods like *findAll*.

If you want to add code that it's executed once per test class, edit the parent AbstractUnitTest and its methods *initOnce* and *destroyOnce*. Be aware these methods contain code used to recreate the structure needed to run DSpace tests, so be careful when adding or removing code there. Our suggestion is to add code at the end of *initOnce* and at the beginning of *destroyOnce*, to minimize the risk of interferences between components.

Be aware that tests of classes that extend DSpaceObject should extend AbstractDSpaceObjectTest instead due to some extra methods and requirements implemented in there.

How to run the tests

The tests can be activated using the commands

```
mvn package -Dmaven.test.skip=false  //builds DSpace and runs tests

  or
mvn test -Dmaven.test.skip=false     //just runs the tests
```

or by changing the property "*activeByDefault*" at the profile (*skiptests*) in the main *pom.xml* file, at the root of the project and then running

```
mvn package  //builds DSpace and runs tests
  or
mvn test     //just runs the tests
```

Be aware that this command will launch both unit and integration tests.

Integration Tests

These tests work at the API level and test the interaction of components within the system. Some examples are placing an item into a collection or creating a new metadata schema and adding some fields. Primarily these tests operate at the API level ignoring the interface components above it.

The main difference between these and the unit tests is in the test implemented, not in the infrastructure required, as these tests will use several classes at once to emulate a user action.

The integration tests also make use of ContiPerf to evaluate the performance of the system. We believe it doesn't make sense to add this layer to the unit tests, as they are tested in isolation and we care about performance not on individual calls but on certain tasks that can only be emulated by integration testing.

Structure

Integration tests use the same structure as Unit tests. A class has been created, called *AbstractIntegrationTest*, that inherits from *AbstractUnitTest*. This provides the integration tests with the same temporal file system and in-memory database as the unit tests. The class *AbstractIntegrationTest* is created just in case we may need some extra scaffolding for these tests. All integration tests should inherit from it to both distinguish themselves from unit tests and in case we require specific changes for them.

Classes that contain the code for Integration Tests are named *<class>IntegrationTest.java*.

The only difference right now between Unit Tests and Integration Tests is that the later include configuration settings for ContiPerf. These is a performance testing suite that allows us to reuse the same methods we use for integration testing as performance checks. Due to limitations mentioned in the following section we can't make use of all the capabilities of ContiPerf (namely, multiple threads to run the tests) but they can be still be useful.

Limitations

Tests structure

These limitations are shared with the unit tests.

The solution to copy the file system is not a very elegant one, so we appreciate any insight that can help us to replicate the required files appropriately.

The fact that we load the tests configuration from a dspace-test.cfg file means we are only testing the classes against a specific set of configurations. We probably would like to have tests that runs with multiple settings for the specific piece of code we are testing. This will require some extra classes to modify the configuration system and the way this is accessed by DSpace.

Events Concurrency Issues

We have detected an issue with the integration tests, related to the *Context* class. In this class, the List of events was implemented as an *ArrayList<Event>*. The issue here is that *ArrayList* is not a safe class for concurrency. Although this would not be a problem while running the application in a JEE container, as there will be a unique thread per request (at least in normal conditions), we can't be sure of the kind of calls users may do to the API while extending DSpace.

To avoid the issue we have to wrap the List into a synchronized stated via Collections.synchronizedList . This, along a synchronized block, will ensure the code behaves as expected.

We have detected the following classes affected by this behavior:

- BasicDispatcher.java

In fact any class that calls *Context.getEvents()* may be affected by this. A comment has been added in the javadoc of this class (alongside a TODO tag) to warn about the issue.

Context Concurrency Issues

There is another related issue in the Context class. Context establishes locks in the tables when doing some modifications, locks that are not lifted until the context is committed or completed. The consequence is that some methods can't be run in parallel or some executions will fail due to table locks. This can be solved, in some cases, by running *context.commit()* after a method that modifies the database, but this doesn't work in all cases. For example, in the *CommunityCollection Integration Test*, the creation of a community can mean the modification of 2 rows (parent and new community). This causes this kind of locks, but as it occurs during the execution of the method *create*() it can't be solved by *context.commit()*.

Due to these concurrency issues, ContiPerf can only be run with one thread. This slows the process considerably, but until the concurrency issue is solved this can't be avoided.

How to build new tests

To build a new Integration Test, create the corresponding class in the project *dspace-test*, under the *test* folder, in the package where the original class belongs. Tests for all the projects (dspace-api, dspace-jspui-api, etc) are stored in this project, to avoid duplication of code. Name the class following the format *<RelatedClasses>IntegrationTest.java*.

There are some common imports and structure, you can use the following code as a template:

```
//Add DSpace licensing here at the top!
package org.dspace.content;

import java.sql.SQLException;
import org.dspace.core.Context;
import org.junit.*;
import static org.junit.Assert.* ;
import static org.hamcrest.CoreMatchers.*;
import mockit.*;
import org.apache.log4j.Logger;
import org.dspace.core.Constants;
/**
 * This is an integration test to validate the metadata classes
 * @author pvillega
 */
public class MetadataIntegrationTest  extends AbstractIntegrationTest
```

```
{
    /** log4j category */
    private static final Logger log = Logger.getLogger(MetadataIntegrationTest.class);


    /**
     * This method will be run before every test as per @Before. It will
     * initialize resources required for the tests.
     *
     * Other methods can be annotated with @Before here or in subclasses
     * but no execution order is guaranteed
     */
    @Before
    @Override
    public void init()
    {
        super.init();
    }

    /**
     * This method will be run after every test as per @After. It will
     * clean resources initialized by the @Before methods.
     *
     * Other methods can be annotated with @After here or in subclasses
     * but no execution order is guaranteed
     */
    @After
    @Override
    public void destroy()
    {
        super.destroy();
    }

    /**
     * Tests the creation of a new metadata schema with some values
     */
    @Test
    @PerfTest(invocations = 50, threads = 1)
    @Required(percentile95 = 500, average= 200)
    public void testCreateSchema() throws SQLException, AuthorizeException, NonUniqueMetadataException,
IOException
    {
        String schemaName = "integration";

        //we create the structure
        context.turnOffAuthorisationSystem();
        Item it = Item.create(context);

        MetadataSchema schema = new MetadataSchema("htpp://test/schema/", schemaName);
        schema.create(context);
        [...]

        //commit to free locks on tables
        context.commit();

        //verify it works as expected
        assertThat("testCreateSchema 0", schema.getName(), equalTo(schemaName));
        assertThat("testCreateSchema 1", field1.getSchemaID(), equalTo(schema.getSchemaID()));
        assertThat("testCreateSchema 2", field2.getSchemaID(), equalTo(schema.getSchemaID()));
[...]
        //clean database
        value1.delete(context);
        [...]

        context.restoreAuthSystemState();
        context.commit();
    }

}
```

The sample code contains common imports for the tests and common structure (*init* and *destroy* methods as well as the log). You should add any initialization required for the test in the *init* method, and free the resources in the *destroy* method.

The sample test shows the usage of the *assertThat* clause. This clause (more information in JUnit help) allows you to check for condition that, if not true, will cause the test to fail. We name every condition via a simple schema (method name plus an integer indicating order) as the first parameter. This allows you to identify which specific assert if failing whenever a test returns an error.

Please be aware methods *init* and *destroy* will run once per test, which means that if you create a new instance every time you run *init*, you may end up with several instances in the database. This can be confusing when implementing tests, specially when using methods like *findAll*.

If you want to add code that it's executed once per test class, edit the parent AbstractUnitTest and its methods *initOnce* and *destroyOnce*. Be aware these methods contain code used to recreate the structure needed to run DSpace tests, so be careful when adding or removing code there. Our suggestion is to add code at the end of *initOnce* and at the beginning of *destroyOnce*, to minimize the risk of interferences between components.

How to run the tests

The tests can be activated using the commands

```
mvn package -Dmaven.test.skip=false  //builds DSpace and runs tests
  or
mvn test -Dmaven.test.skip=false     //just runs the tests
```

or by changing the property "*activeByDefault*" at the profile (*skiptests*) in the main *pom.xml* file, at the root of the project and then running

```
mvn package  //builds DSpace and runs tests
 or
mvn test     //just runs the tests
```

Be aware that this command will launch both unit and integration tests.

Code Analysis Tools

Due to comments in the GSoC meetings, some static analysis tools have been added to the project. The tools are just a complement, a platform like Sonar should be used as it integrates better with the structure of DSpace and we could have the reports linked to Jira.

We have added the following reports:

- FindBugs : static code bug analyser
- PMD and CPD: static analyser and "copy-and-paste" detector
- TagList: finds comments with a certain annotation (like XXX or TODO)
- Testability Explorer: detects issues in classes that difficult the creation of unit tests

These reports can't replace a Quality Management tool but can give you an idea of the status of the project and of issues to be solved.

The reports can be generated by launching:

```
mvn site
```

from the main folder. Be aware this will take a long time, probably more than 20 minutes.

Functional Tests

These are tests which come from user-based use cases. Such as a user wants to search DSpace to find material and download a pdf. Or something more complex like a user wants to submit their thesis to DSpace and follow it through the approval process. These are stories about how users would preform tasks and cover a wide array of components within Dspace.

Be aware that in this section we don't focus on testing the layout of the UI, as this has to be done manually to ensure we see exactly the same in different browsers. In this section we only consider tests that replicate a process via the UI, to ensure it is not broken due to some links missing, unexpected errors or similar issues.

Choices taken

To decide on a specific implementation of these tests some choices have been taken. They may not be the best but at this time they seemed the appropriate ones. Contributions and criticism are welcomed.

On one hand, functional tests run against a live instance of the application. This means we need a full working environment with the database and file system. It also means we are running them against the modified UI of a specific installation. As a consequence, we want the tests to be very generic or easily customizable, but we have to be aware that due to way Maven (and particularly its packaging system) works it isn't possible to run the tests as a step of the unit and integration testing described in the above sections.

On the other hand, if we focus on the tools available, the best choice we have available is Selenium, a suite of tools to automate testing of web applications. Selenium can be run in two modes: as a Firefox plug in that will allow us to record test scripts and run them later, or as a distributed system that allows us to run the tests against several browsers in different platforms.

We have to choose a way to run the tests that is easy to set up and adapt to a particular project by the developers, while limited by the options that maven and selenium provide. The decision has been to use the Selenium IDE to run the tests. This means the tests can only be run in Firefox and they have to be launched manually, but on the other hand they are easily customizable and runnable.

The Selenium RC environment was discarded due to the complexity it would add to the testing process. If you are a DSpace developer and have the resources and expertise to set it up, we clearly recommend so. You can reuse the scripts generated by the Selenium IDE and add extra tests to be run with JUnit alongside the existing unit and integration tests, which allows you to build more detailed and accurate tests. Even if we recommend it and it is a more powerful (and desirable) option, we are aware of the extra complexity it would add to just run the functional tests, as not everybody has the resources to deploy the required applications. That's why we have decided to provide just the Selenium IDE scripts, as they require much less effort to be set up and will do the job.

Structure

Selenium tests consist of two components:

- The Selenium IDE, downloadable here , which is a Firefox addon that can record our actions and save them as a test
- The tests, which are HTML files that store a list of actions to be "replayed" by the browser. If any of the actions can't be executed (due to a field or link missing or some other reason) the test will fail.
    - The recommendation is to create one test per user-action (like creating an item). Several tests can be loaded at once in the IDE and run sequentially.

To install the Selenium IDE, first install Firefox and then try to download it from Firefox itself. The browser will recognize the file as an addon and it will install it.

Limitations

The resulting tests have several limitations:

- Tests are recorded against a specific run on a particular machine. This means some steps may include specific values for some variables (id numbers, paths to files, etc) that link the test to a particular installation and state in the system. As a consequence we have to ensure we have the same state in the system every time we run the tests. That may mean run the tests in a certain order and probably starting from a wiped Dspace installation.
- For the same reason as above, some scripts may require manual changes before being able to be run (to ensure the values expected by Selenium exist). This is specially important in tests which include a reference to a file (like when we create an item) as the path to the file is hard coded in the script.
- Tests have to be launched manually and can only be run in Firefox using the Eclipse IDE. We can launch all our tests at once, as a test suite, but we have to do so manually.
- Due to they way Selenium works (checking HTML code received and acting upon it) high network latency or slowness in the server may cause a test to fail when it shouldn't. To avoid this, it is recommended to run the tests at minimum speed and (if required) to increase the time Selenium waits for an answer (it can be set in the Options panel).
- Tests are linked to a language version of the system. As Selenium might reference, in some situations, a control by its text, a test will fail if we change the language of the application. Usually this is not a big problem as the functionality of the application will be the same independently of the language the user has selected, but if we want to test the application including its I18N components, we will need to record the actions once per language enabled in the system.

How to build new tests

Building a test in Selenium IDE is easy. Open the IDE (*in Firefox, Tools > Selenium IDE*) and navigate to your DSpace instance in a tab of Firefox (i.e.: http://localhost:8080/jspui/). Press the record button (red circle at top-right corner) in the Selenium IDE and navigate through your application. Selenium will record every click you do and text you write in the screen. If you do a mistake, you can right-click over an entry in the list and remove it.

Actions are stored by default as a reference to the control activated. This reference is generic, meaning that Selenium might look for an anchor link (<a>) that points to a certain url (like '/jspui/handle/1234/5678'), independently of its id, name or position in the application. This means that the test will not be affected (usually) by changes in the layout or some refactoring. That said, in some specific cases you may need to edit the test cases to change some values.

Once you are finished, press again the record button. Then, in the Selenium IDE, go to File > Save Test Case and save your test case.

In Selenium the tests cases are a HTML file that stores data in a table. The table contains a row per action, and each row has 3 columns :

1. An action to be run (mandatory). This can be an action like open, click, etc.
2. A path or control id against which the action is executed (mandatory). This can point to an URL, a control (input, anchor, etc) or similar.
3. A text to be added or selected in an input control (optional).

A sample of a Selenium test is:

```
open          /xmlui
clickAndWait  link=Subjects
clickAndWait  //div[@id='aspect_artifactbrowser_Navigation_list_account']/ul/li[1]/a
type          aspect_eperson_PasswordLogin_field_login_email                          admin@dspace.org
type          aspect_eperson_PasswordLogin_field_login_password                       test
clickAndWait  aspect_eperson_PasswordLogin_field_submit
```

The code generated by the Selenium IDE for this would be like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head profile="http://selenium-ide.openqa.org/profiles/test-case">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<link rel="selenium.base" href="" />
<title>Create_Comm_Coll_Item_XMLUI</title>
</head>
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">Create_Comm_Coll_Item_XMLUI</td></tr>
</thead><tbody>
<tr>
    <td>open</td>
    <td>/xmlui</td>
    <td></td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>link=Subjects</td>
    <td></td>
</tr>

[... more actions ...]

</tbody></table>
</body>
</html>Create_Comm_Coll_Item_XMLUI
```

You can use the Selenium IDE to generate the tests or write them manually by using the Selenese commands.

How to run the tests

To run the tests simply open the Selenium IDE (*in Firefox, Tools > Selenium IDE*) and navigate to your DSpace instance in a tab of Firefox (i.e.: http://localhost:8080/jspui/). Then, in the Selenium IDE, click on *File > Open* and select a test case. You can open as many files as you want, they will be run in the order you opened them.

Once you have selected the test cases to run, ensure the speed of Selenium is set to slow (use the slider) and press either the "Play entire test suite" or "Play current test case" button (the ones with a green arrow), according to your intentions. Selenium will run the actions one by one in the order recorded. If at some point it can't run an action, it will display an error and fail the test. You can see the reason of the error in log at the bottom of the Selenium IDE window.

A very common reason why a test fails is because the server returned the HTML slowly and Selenium was trying to locate and HTML element before having all the HTML. To avoid this make sure that Selenium speed is set to slow and increase the default timeout value in Options.

Provided tests

We have included some sample Selenium tests in Dspace so developers can experiment with them. The tests are located under *"<dspace_root>/dspace-test/src/test/resources/Selenium scripts".* They are HTML files following the format described above. They are doing some assumptions:

- Tests assume that you are running them against a vanilla environment with no previous data. They may work in an environment with data, but it's not assured
- Tests assume you are running the English UI, other languages may break some tests.
- Tests assume a user with user name *admin@dspace.org* and password *test* exists in the system and has administrator privileges
- Tests assume a file exists at */home/pvillega/Desktop/test.txt*

You can edit the tests (see the format above) and change the required values (like user and path to a file) to values which are valid in your system.

Advanced Usage

If you set up Selenium RC, you can reuse the test scripts to be run as JUnit tests. Selenium can export them automatically to Java classes using JUnit. For this open the Selenium IDE (*in Firefox, Tools > Selenium IDE*), click on *File > Open* and select a test case. Once the test case is loaded, click on File > Export Test Case As > Java (JUnit) - Selenium RC. This will create a Java class that reproduces the test case, as the following:

```
package com.example.tests;

import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class CreateCommunity extends SeleneseTestCase {
    public void setUp() throws Exception {
        setUp("http://localhost:8080/", "*chrome");
    }
    public void testJunit() throws Exception {
        selenium.open("/jspui/");
        selenium.click("xpath=//a[contains(@href, '/jspui/community-list')]");
        selenium.waitForPageToLoad("30000");
        selenium.click("link=Issue Date");
        selenium.waitForPageToLoad("30000");
        selenium.click("link=Author");
        selenium.waitForPageToLoad("30000");

        [... ]
    }
}
```

As you can see in the code, this class suffers from the same problems as the Selenium IDE scripts (hardcoded values, etc) but can be run using Selenium RC in a distributed environment, alongside your JUnit tests.

Future Work

This project creates a structure for testing that can expanded. Future tasks would include:

- Integrating with a Quality Management tool like Sonar
- Integrating with a Continuous Integration tools
- Adding Unit and Integration tests for the remaining classes
- Extending Functional tests
- Creating a "Code quality" release, where priority is not new functionalities but stability and quality of code

Thanks

This page has been created with help from Stuart Lewis, Scott Phillips and Gareth Waller. I want to thank them all for their comments. Some information has been taken from Wikipedia to make the text more complete. I'm to blame for errors in the text.

Feel free to contribute to this page!