DSpace Testing

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.

You can find extra information at the related GSoC page

Quick Start

If you want to start creating tests straight away do the following:

- Open *dspace-test* module
- Under the *test* packages, create a new class that extends *AbstractUnitTest* or *AbstractIntegrationTest* according to the type of test you want to create
- Add the different tests methods

To run the tests:

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

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

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 in widespread use in the community. The framework simplifies the development of unit tests and the current IDEs 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, Hamcrest is included. Hamcrest is a library of matcher objects that facilitate the validation of conditions in the tests.

JMockit

JMockit is a 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 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.

How to skip a test

It can occasionally happen that a commit to the master branch breaks a test. Such commit should be reverted ASAP, but sometimes it's just not possible /viable. What to do if a test is failing and you can't fix it? You can still run the other tests and temporarily disable the failing test using the Surefire exclusions:

In [dspace-src]/pom.xml, add a line excluding the test class (in this case "DSpaceServiceManagerTest"):

```
...
<artifactId>maven-surefire-plugin</artifactId>
  <version>2.6</version>
  <!-- tests whose name starts by Abstract will be ignored -->
  <configuration>
  <excludes>
    <exclude>**/Abstract*</exclude>
    <exclude>**/DSpaceServiceManagerTest*</exclude>
  </excludes>
...
```

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

There is 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 with a synchronized block, will ensure the code behaves as expected.

The following classes are 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

Some static analysis tools are included in the project. The following reports are available:

- 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 like Sonar but give 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.