SpagoBI – Unit Test Environment

Authors Daniele Balestrin
Davide Zerbetto

Index

1. Version

Version/Release n° :	0.1	**Data Version/Release :**	08/03/05
Update description:	Alfa release – Paragraphs 2, 3 and 4 by D. Balestrin		
Version/Release n° :	0.2	**Data Version/Release :**	08/18/05
Update description:	Alfa release – Review by G. Ruffatti		
Version/Release n° :	0.3	**Data Version/Release :**	10/18/05
Update description:	Alfa release – Paragraphs 5, 6, 7 and 8 by D. Zerbetto		

2. Document goal

This document outlines the SpagoBI's testing guidelines. An automatic unit test environment is provided with the aim of:

- granting the SpagoBI's components correctness thanks to an effective testing environment;
- granting the regression testing execution;
- defining the testing guidelines to be followed by the SpagoBI developers during the development of the new components.

The document is intended to technicians that want to configure an environment to run the pre-defined tests.

For more information about SpagoBI, please refer to all the other documents available on: http://spagobi.eng.it.

3. Software Requirements

The testing environment requires the following software components:

- Java Development Kit Version 1.4.2_08;
- Spago 2.0.0;
- Ant 1.6.5.

Please see Chapter 8 of this document for the dependencies list of this project. These dependencies are required to compile and run the application

4. Methodologies

The SpagoBI testing environment uses different testing frameworks and components; they are JUnit, DBUnit and Mock Objects.

JUnit is a commonly used framework that allows to write some tests for Java object methods and it controls the successful execution by means of the assertions on the expected results. You can find more information about JUnit's methodology and API on http://www.junit.org.

DBUnit is a JUnit extension specifically developed for the unit test of methods which involve a database. The aim of the testing is to run it every time in the same condition so that the initial state of the DB is known, because the testing unit is the only one able to access the data during the test. This framework is used in order to provide some DB facilities, such as a flat xml file to reinitialize the database each time for every test (in SpagoBI testing environment the reinitializing file is `src/FullDataSet.xml` with DTD file `src/FullDataSet.dtd`); so, you are sure that the rows involved in a test are only the expected ones and that there are no other modification caused by the data or tests history. You can find more information about DBUnit on http://www.dbunit.org.

Mock Objects is a methodology to test objects which depend by other complex ones. Mock Objects substitute the complex objects with simpler ones and assuming the complex objects runs successfully. You can find more information about Mock Objects on http://www.mockobjects.com.

5. Testing scope and coverage

The SpagoBI development doesn't follow a full-test approach (like test-first-development); anyhow, the main SpagoBI's components are submitted to an automatic unit testing.

The goal of the developed tests is to examine the correctness of some core components and logical features of SpagoBI platform:

1. integrity between data model and data stored on DB (load-insert-delete-update operations into DB);
2. integrity between data stored on DB and on CMS (Content Management System). Since SpagoBI has a double persistence, on DB for the execution configuration and setting of documents and on CMS for documents templates, it's very important to verify the coherence between the two persistence systems. It means that: a successful operation in writing on DB (or on CMS) must be reflected on a successful correspondent operation on CMS (DB), and an operation that raises an error while writing on DB (CMS) must have no consequences on the latter CMS (DB);
3. coherence between user's requests and application execution flow;
4. coherence between security policy and documents visibility rules. SpagoBI sets administrators, developers, end-users and testers as different technical roles and associates them to the portal user group definition; then administrators manage documents permissions (visibility, execute, update and create rights) on the functionality folders. In this way one user may see and interact with only a part of the entire documents collection present on the system, according to his profile and roles and the documents state. The correctness of permissions management is obviously a very critical point for security;
5. absence of ambiguity in the use modalities of the parameters. A role must have only one use modality for one parameter: if a parameter has two different use modalities for the same role, the platform must raise an exception;
6. input parameters validation. The documents execution is conditioned by some parameters values: these values are set by the user and are submitted to a validation process.

In order to satisfy the above scopes the following units were be tested:

- ✔ Hibernate DAOs (referring to points 1, 2, 4 and 5): they are classes for data access (according to pattern DAO - "Data Access Objects") and they use Hibernate for DB interfacing. JDBC DAOs were not tested, as they will be soon deprecated;
- ✔ actions and modules (referring to point 3): the dispatching modalities for applications execution inherited from Spago framework (on which SpagoBI is based). In the Model-View-Controller pattern, SpagoBI inherits Controller component from Spago, while actions and modules take part in the Model component as they examine the user's request and produce the desired response. Every module or action is tested only in itself for the execution flow it contains (calls to DAO classes, logic controls, response preparation); a test on cooperation between different modules and actions is still lacking, it would not be a Unit Test, indeed;
- ✔ publishers (referring to point 3): the response presentation dispatching system based on Spago framework;
- ✔ generators of the functional documents trees (referring to point 4), the collection of documents (where folders are internal nodes), that are visible to the user. The generators are the components returning documents tree according to each user profile (administrator, developer, tester and end-user) and the roles associated to it;
- ✔ static and dynamic validators (referring to point 6): utility methods for the input parameters values control for correctness. SpagoBI offers both predefined (static) and configurable (dynamic) checks;
- ✔ some other utilities classes necessary for the right execution of the application.

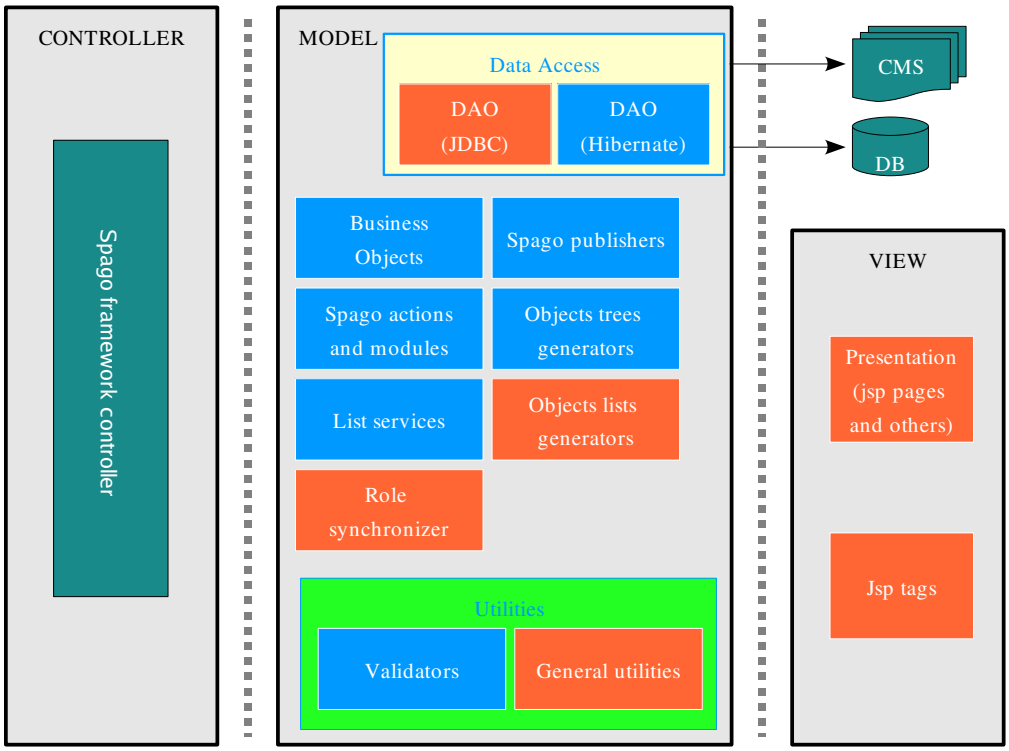
Modules and trees or lists generators are not tested in isolation: it means that actual Hibernate DAO classes are called and not substituted by mock objects. The reason is that: since tests on those components need a minimal set of data (which is provided by the used database), the more simple and immediate way to get right evaluation on methods under test is using the database itself and the already written DAO methods (already tested, obviously).

The following open source products were used for running tests:

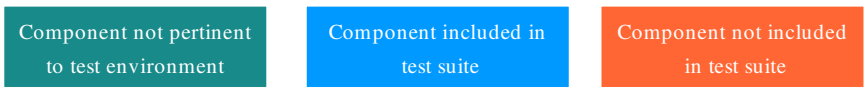
- databases: HSQL in-memory engine (version 1.7.3) and PostgreSQL (Driver 8.0-312). The default database is HSQLDB; with few modifications on the configuration files, it's also possible to run tests on PostgreSQL (if available), but tests on this database take much more time than using HSQLDB;
- CMS content repository: JackRabbit;

- Business Intelligence specific engines: Jasper Report Engine for static reporting and JPivot-Mondrian Olap Engine for OLAP analisys.

The following diagram shows SpagoBI's components included and not included in this testing suite.



Legend:



Tested Component	Test scope (critical points)	Test coverage
DAO (Hibernate) - Business Objects	integrity between data-model and persistence systems (DB and CMS); coherence between security policy and documents permissions; absence of ambiguities for documents execution;	load-insert-modify-erase operations into DB and CMS; logic controller for documents permissions with reference to the roles associated to the user;
Spago publishers	coherence between execution flow and application output;	methods for publishers name determination with regard to the application execution flow;
Spago actions and modules	dynamic validation; coherence between user's request and application execution flow;	dynamic validators; logic controller of execution flow inside each module or action;
Functional documents trees generators	coherence between security policy and documents permissions;	logic controller for documents permissions with reference to the roles associated to the user and his profile;
List services	coherence between DB and list services;	lists generators;
Validators	correctness of input parameters validation;	execution of validation operations.

6. Environment set-up

Before starting the test, you have to prepare your environment.

After download of `SpagoBITest-src-1.0.zip` from SpagoBI Project download site, unzip it in a folder; in the following we will refer to this folder as "test directory".

In the first step you have to configure the test properties files to see the HSQLDB testing database, that is the default.

The files involved in this operation are:

- `src/conf/data_access.xml`
- `src/hibernate.cfg.hsqldb.xml`

You need to udpate the uri with the absolute path where the folder HSQLDB was unzipped (you have only to type the test directory in both files where requested).

Note: If you want to run tests on your own database, you need the following files for setting up your database connection:

- `src/conf/data_access.xml`
- `src/hibernate.cfg.xml`

You must set here your database driver class and version, SQL dialect, user, password, uri and mapping classes. Please consider that the default version uses HSQLDB and the file `src/conf/spagobi/spagobi.xml` refers to `src/hibernate.cfg.hsqldb.xml`, that is the Hibernate configuration file for HSQLDB: for this reason, after having modified `src/hibernate.cfg.xml` for your database requirements, remember to change the reference in `src/conf/spagobi/spagobi.xml`.

In `src` folder you find Hibernate configuration files for Oracle, PostgreSQL and MySQL as sample, and in `SQL_scripts` folder the scripts for tables creation in those databases.

You need also to configure the CMS: in `src/jackrabbit.properties` you have to configure the folder in which the CMS will be installed and also the absolute path of the zip file for the CMS re-initialization. Proceed as follows:

- type the absolute path of the test directory in `repository_path` and `zip_repository_file_for_recovery`, where requested.

Finally you can start `maven` from the command line at the test directory.

Maven will resolve all the dependencies, compile test classes, run tests and produce the junit reports. After maven script execution, you can see an overview-summary of all tests in file `target/docs/junit-report.html` or in `target/docs/index.html` clicking on `Project Reports --> Unit Tests`.

You must pay attention to the fact that, if a test involving CMS is abruptly interrupted, CMS environment should be restored to avoid further test failures. So, before re-starting the tests, you have to delete the content of the folder `jackrabbitRepositoriesForTest/spagobi_demo` and then re-starting `maven` command.

7. Guidelines for new test components development

Each test method needs different contexts for execution. In the following we provide you with some useful practices:

- run the initializer classes from `setUp()` method through the command `InitializerManager.init()`. So, the framework read the configuration file

`src/conf/master.xml` and load coherently the `ConfigSingleton` object; moreover some general utilities are available (tracing system first);

- for tests using database, new Junit Test Cases should extend the following class: `it.eng.spagobi.test.dbunit.utility.DBConnectionTestCase` which provides the connection parameters (retrivied from the configuration file `src/conf/data_access.xml`) and the reinitializer xml data set;
- use assertion methods supplied by JUnit to compare expected results and actual results.

8. Dependencies

The following is a list of dependencies for this project. Maven will try to resolve them automatically.

Dependencies required to compile the application:

- `junit-3.8.1.jar`
- `dbunit-2.1.jar`
- `spago-core-2.0.0.jar`
- `hibernate-3.0.5.jar`
- `spagocms.renamejcr.jar`
- `sbi.securityprovider.api-1.4.jar`
- `portlet-api-1.0.jar`
- `spagobi-1.4.jar`
- `servlet-api-2.4.jar`
- `spagoportlet.jar`

Dependencies required to run the application:

- `hsqldb-1.7.3.0.jar`
- `eng.jackrabbit-0.16.4.1-dev.jar`
- `eng.jcr.1.0.jar`
- `sbi.cmsfactory.jackrabbit-1.4.jar`
- `asm-attrs-1.5.3.jar`
- `asm-1.5.3.jar`
- `cglib-2.1_2.jar`
- `sbi.drivers.api-1.4.jar`

- sbi.drivers.jpivot-1.4.jar
- sbi.drivers.jasperreport-1.4.jar
- geronimo-spec-jta-1.0-M1.jar
- sbi.cmsfactory.api-1.4.jar
- antlr-2.7.5H3.jar
- commons-collections-3.1.jar
- commons-dbcp-1.2.1.jar
- commons-logging-api.jar
- commons-logging.jar
- commons-pool-1.2.jar
- concurrent-1.3.4.jar
- dom4j-1.6.jar
- ehcache-1.1.jar
- log4j-1.2.8.jar
- lucene-1.4.3.jar
- xercesImpl-2.6.2.jar
- xml-apis-2.0.2.jar
- jamon-1.0.jar
- commons-fileupload-1.1-dev.jar
- commons-validator-1.1.4.jar
- jakarta-oro-2.0.8.jar