Alex Soto Bueno (asotobu@gmail.com)

Ten years ago to present, Enterprise Java Applications have suffered many changes, JavaScript has been gaining importance on client side because of HTML5 or JQuery, server side has changed too, so does persistence layer. In presentation we are going to explore what open source projects use to write unit, integration and acceptance tests for Enterprise Java Applications and how to integrate them in your continuous integration system.

Introduction

Ten years ago to present, Enterprise Java Applications have suffered many changes. We have moved from Enterprise Applications built with *JSP+Servlet* and *EJB*, to much more complex applications. Nowadays with the advent of *HTML5* or *JavaScript* libraries like *JQuery*, client side development has changed significantly. With the emergence of web frameworks like *Spring MVC* or *JSF*, server side code has quite changed compared to the one used when each web-form was mapped to a *Servlet*. And also persistence layer has changed with *Java Persistence* standard or with new database approaches like *Data-Grid*, *Key-Values stores* or *Document stores*.

Moreover, architectural changes have occurred too, *REST*-web applications have grown in popularity or *AJAX* is used to create asynchronous web applications. Due to development of Enterprise Java Applications have changed during these years, so testing frameworks have changed accordantly.

In this paper we are going to see how to write effective unit tests for these changes on client side, server side and persistence side. When each module is tested separately, it is time to write integration tests and resolve container lifecycle management problem. Finally to validate that our software meets functional requirements, we will see how to write acceptance tests where front end is a web.

Unit Testing

Unit testing is a method by which the smallest testable part of an application is validated.

The main advantages of writing unit tests are:

- Quick feedback about correct behaviour of developed classes.
- Confidence over change. You can change one part of code knowing that if something goes wrong it will be detected as soon as tests

are executed.

- Tests can be used as documentation to know exactly what a code does. *Javadoc* can mislead you, passed tests "not".

To write effective tests, each test must follow the next five rules, which can be summarised with acronym *FIRST*.

- (F)ast: unit tests should run fast. If they run fast you will run them more often, so you will receive feedback about possible problems earlier. To make your tests faster, there is one gold rule, do not access to I/O system (network, filesystem, ...) and this implies to not hit database in tests.

- (I)solution: code under test should not have to make calls to external dependencies, where an external dependency could be a class, a component, a subsystem, but also must be isolated from itself. Your test cannot depend on execution of another test. The code's nature is to have calls to other classes, other components, ... so we need to create *mocks* and *stubs* to these components required by code under test.

- (R)epeteable: tests must return the same result under any circumstances.

- (S)elf-Validating: unit tests are production-code too, so treat it in the same manner. When someone reads a test must understand quickly what is the purpose of this test. Moreover tests act as documentation, so it is important to write them clearly. A good point to start is naming test classes and test methods with names that describes behaviour they are going to test, with no implementation details.

- (T)imely: tests should be written before production code. This will make your code designed to be testable.

In *JEE* development, each layer requires a different approach to write unit tests. It seems logical that if client side is mostly written in *Javascript* and communication with server side is done using *JSON* or *XML*, it would require a different test framework that the one used in persistence layer.

Client Side

JS TestDriver

There are many frameworks to test *Javascript* client code like *QUnit*, *Jasmine* or *JSUnit*, but for me one of the most complete is *JS TestDriver* (http://code.google.com/p/js-test-driver/). *JS TestDrive*r is a *Javacript* test framework that aims us to write *Javascript* tests using *xUnit* approach. But also

offers valuable features like *Eclipse* integration, running tests through multiple browsers, code coverage calculation or *Maven* plugin to generate test reports.

To write *Javascript* tests with *JS TestDriver*, first of all you need to write a *Javascript* file with test implementation and assertions.

If you are familiar to *JUnit* there is not much difference apart from being written in *Javascript*. Also you can define *tearDown* and *setUp* methods as you normally do in JUnit.

Also you can create optional *HTML* files known as fixtures. Fixtures are *HTTL* code (embedded into test or at external file) which is used during test. They are really useful when your *Javascript* code modifies dynamically *HTML DOM* structure.

Javascript tests are run directly on browser, so it is likely we need an *HTML* file to run tests. With *JS TestDriver* instead of creating manually an *HTML* file with production and test code, it comes with a server which automatically creates *HTML* including production and test code. Then you only have to open a browser and access the server address and configured tests will be run in opened browser.

Server Side

JUnit

If you are not using a polyglot approach, *JEE* applications are written in Java. We can consider JUnit (www.junit.org) as the "*de facto*" test framework. *JUnit* is widely used and clearly does not require many presentations.

But there are some extensions of *JUnit* that can help in the task of developing server side tests.

First extension is *Parameterized* test. Some parts of our code could be tested using *Data-Driven* testing (*DDT*) approach. *DDT* is based on supplying to tests input parameters and verifiable outputs from a table, which means that each row represents one test execution. This approach is really useful in business code that requires some kind of mathematical calculation or a wide variety of possible parameter values. A parameterized test is a JUnit extension that lets us do *DDT* with minimum of fuss.

To write Parameterized tests in *JUnit*, unit test should follow some particularities.

- Runner must be changed to *Parameterized* class.
- A static method annotated with *@Parameters* and returning a Collection of Object array. Each element of the collection represents a row, meanwhile each element of the array represents the column element of

that row.

- Unit test must contain a constructor with all attributes that forms a row. For each row, *JUnit* runner will create a new instance of test calling parameterized constructor.

- And finally a test which will use class attributes.

Another *JUnit* extension is *Rules*. *JUnit Rules* are classes that modify how tests are run and reported. *JUnit Rules* implement *TestRule* interface and we can do the same as we do previously with *@Before* (method is executed before test method), *@After* (method is executed after test method), *@BeforeClass* (method is executed before test class) and *@AfterClass* (method is executed after test class) annotations, but because they are classes, they can be easily shared between projects and tests. You simply need to annotate *TestRule* instance with *@Rule,* if rule is executed for each method, or *@ClassRule* if rule is executed for each test class or test suite.

Last *JUnit* extension that can help you to organise tests are Categories. Categories are the way *JUnit* has to group tests by type. Each type is implemented as interface, and for each test you need to set in which category it belongs.

Hamcrest

One of important rules to write effective tests is that they must be self-validating. With a quick overview everyone should be able to read and understand what a test does and what is validating. To accomplish this rule, it is important to correctly name test classes and test methods, but it is also important to write assertions in a clear way and *JUnit* assertion method (*assertEquals* family) does not help too much.

A further problem is that you cannot concatenate assertions, for example exists *assertEquals* method, but not *assertNotEquals*.

We can enhance assertions using Hamcrest (http://code.google.com/p/hamcrest/) library. Hamcrest is a library of matchers for building test expressions in a more natural language.

Mockito

Unit tests must be isolated, no calls to external components. But code has external dependencies like calls to other classes or system calls. To test classes with external dependencies we need a mock object. *Mock objects* are simulated objects that mimic the behaviour of real objects in controlled way.

In Java exists two major mock frameworks, *EasyMock* and *Mockito*. Both of

them have similar features, in fact *Mockito* starts from a fork of *EasyMock*, but what makes *Mockito* really good is that is thought in terms of expressiveness. *Mockito* uses an explicit language for better readability using *Fluent Interfaces* and Hamcrest matchers. Moreover *Mockito* library is simpler than *EasyMock*, it needs less code to achive the same behaviour.

Persistence Side

It is rare to think about *JEE* applications without a persistence layer. Nowadays most applications have Rational database management systems (*SQL*) at backend, but gradually *NOSQL* database management systems are gaining terrain. Both systems are conceptually different, meanwhile rational systems work with tables, *NOSQL* systems work with heterogenous data structures, like Document Stores (*MongoDB* or *CouchDB*), Graph Databases (*Neo4J*), Column Stores (*HBase* or *Cassandra*) to cite a few.

When testing persistence layer we must focus on not breaking fast and isolation rules.

To not break fast rule, it is as simple as using an in-memory database system (you are not hitting disk or network) so it will run fast. In case of *RDBMS*, many in-memory databases exists like *HSQLDB*, *H2* or *Derby*. But a major problem is found in *NOSQL* systems, where there is no homogeneous system. Each vendor should develop the "in-memory" mode, but this would be the desired scenario, some engines like *MongoDB* does not contain an in-memory mode, but *Neo4j* do. So this problem should be resolved depending on system chosen. In case that no in-memory mode exists, I recommend to not write persistence unit tests, and run them as integration tests.

To not break isolation rule, each test should find the database in a known state at start, so any database modification done by other tests does not affect current run. Again with rational engines a JUnit extension called *DUnit* (http://www.dbunit.org/) exists which is the responsible of maintaining database into stable state before each execution. In summary *DBUnit* cleans database before each method and inserts data defined in a dataset file.

But again with *NOSQL* a common solution does not exist to isolate tests YET. *NOSQL Unit* (https://github.com/lordofthejars/nosql-unit) is a *JUnit* extension which will work as *DBUnit* works but for *NOSQL* systems; it is a newly project and for now it only supports *MongoDB*, but in next versions *Neo4J*, *CouchDB* and *Cassandra* will be supported.

Integration Testing

Integration tests are the counterpart of unit tests, they test collaboration

between components, deal with Input/Output system, remote databases and set special environment configuration.

The problem arises when we write integration tests of *JEE* 6 applications which imply an application server. Every time more and more features are being implemented within a container, to cite some examples *CDI* (Dependency Injections with *@Inject*), *JPA* (Persistence ORM with *@PersistenceContext*), *Bean Validation* or *Servlet Annotations*, and writing integration tests of code that uses these container features plan a new kind of problems.

- code under test must run into a *JEE* container and this leads us to first problem, how to manage lifecycle of container like open/close server operation, deploy application, … from tests.

- how to create the deployment file of code under test.

Arquillian

To solve these problems and other collateral problems, people of *JBoss* have developed *Arquillian*.

Arquillian is a test framework which minimises container lifecycle management by starting and stopping server when tests are executed. With *Arquillian*, and using *Shrinkwrap*, deployment files can be created programmatically and deploy it automatically to defined container. Furthermore rather than creating a deployment file with all project files of your application, you can create a micro deployment file with only code under test.

Arquillian tests can be run within container or as external client. If your test is enriched, that is that your test also uses container features, then it should be run within container, but if integration tests are testing communication protocol, then it should run against server.

Arquillian follows next steps during its execution:

- First *Arquillian* starts configured container (*Tomcat*, *JBoss*, *Glassfish*, …)
- Second step is creating deploy file and upload to server.
- Then tests are executed.
- And finally test results are sent back to runner, and server is stopped.

Acceptance Testing

Acceptance Tests are created by stakeholders and are expressed using business domain language. These are high-level tests to test that business logic and *UI* are implemented meeting requirements accorded by business customers, business analysts, testers and developers.

To write acceptance tests correctly the first thing to do is write user stories. A user story is one or more sentences in everyday language that describes from the point of view of end user, an action which gives business value. User stories typically follow next template:

As a <role>, I want <goal> so that <benefit>. For example:

"*As administrator, I want to add new books to a collection, so users can borrow them.*"

After a user story is specified, acceptance criteria is written. Acceptance criteria defines a set of conditions to consider user story as complete. Typically acceptance criteria follows next template:

Given [Precondition], When [Actor + Action], Then [Observable Result]. For example:

"*Given a new book needs to be added to library, when administrator adds it, then book is visible to all use.*".

"*Given a user into system, when new book is added, then book is available to be borrowed.*".

But also a more natural language can be used, for example:

"*Administrator can categorise books.*"

Apart form acceptance criteria, examples to write tests must be provided too. These examples cannot be technical or impersonal sentences, but should describe exactly which data and under what scenario test is valid in a step-by-step form. These examples will be used by test writers to develop acceptance tests. As an example of specification by example:

Scenario: Administrator wants to add a new book.

1. Create Lord Of The Rings Book.

2. Assign J.R.R Tolkien as author.

3. Set Book Description.

4. Add Book to System.

5. Check that Book is shown on screen.

Almost all *JEE* applications are using a web as graphical user interface, so obviously the acceptance test framework should be able to access web page elements. Moreover because there are many different browsers (*Firefox*, *Chrome*, *Opera*, *IExplorer*, ...) in market, it should be easy to run acceptance tests against any browser. And finally because acceptance tests are organised by user stories and specifications, chosen framework should be able to organise tests and stories.

Thucydides

Acceptance test framework that meets all these requirements is *Thucydides* (http://www.wakaleo.com/thucydides/). Thucydides is a tool designed to make writing automated acceptance and regression tests easier.

Thucydides is a *Selenium 2 Extension*, so it uses *WebDriver API* to access *HTML* page elements. But also helps you to organise your tests and user stories by using a concrete programming model, create reports of executed tests, and finally it also measure functional cover.

To write acceptance tests with *Thucydides* next steps should be followed.

- First of all choose a user story.
- Then implement the PageObject class. *PageObject* is a pattern which models web application's user interface elements as objects, so tests can interact with them programmatically.
- Next step is implementing a steps library. This class will contain all steps that are required to execute an action. For example creating a new book, requires to open *addnewbook* page, insert new data, and click to submit button.
- And finally implement chosen user story by following Acceptance Criteria.

Continuous Integration

In this paper we have talked about three kind of tests, unit tests, integration tests and acceptance tests, but also other kind of tests exists like smoke tests, performance tests, regression tests, user acceptance tests... but only unit tests should be run directly by developers, the other ones should be executed by continuous integration system because it is the responsible of giving feedback loop about the state of the code.

Jenkins/Hudson

The widely adopted open source continuous integration system in Java is *Jenkins/Hudson*. *Jenkins* (and *Hudson*) provides continuous integration services for software development, and although is not mandatory to use *Maven*, Jenkins integrates seamlessly with *Maven*.

Build jobs are at the core of the *Jenkins* build process. Build jobs are particular tasks that should be executed application build and can involve compiling source code, running unit tests or measuring code coverage. Typically a project will contain more than one build task chained, for example one for compiling and running unit tests, if build is successfully run, launch another one for running integration tests, or deploying application to a web server.

To manage deployment pipeline, *Build Pipeline Plugi*n is a *Jenkins* plugin which gives you the ability to form a chain of jobs and view the state of each job for each execution.

Conclusions

Sadly, software culture has tended treat tests as a residual portion of code that can be avoided if no time. This is something hard to change, but gradually more and more developers are seeing for themselves that writing tests (any kind of them) correctly reduce the time it would be spend out in the future to fix an error found in production stage.

Tests are not more important than business code, it should be treated as an equal, same coding rules, same code reviews, and same resources.

Moreover if you are planning to adopt *Continuous Delivery* principles, the only way to guarantee the correctness of software is writing tests.

Finally it is important to note that 80% of your test code should be written in unit tests, because they are responsible to test border conditions, unusual cases or exceptions that should not happen. Integration tests should not be more than 15%, and only 5% of lines of tests should be involved in Acceptance tests.

Alex Soto, www.lordofthejars.com