Unit tests

Your Rating: Results: 96 rates

Introduction

Magnolia uses JUnit 4 and if dynamic mocks are required Mockito (new) for creating unit tests. Before we were using EasyMock for dynamic mocks.

Mockito is a more recent mocking library. We'll not bulk convert existing EasyMock-Tests as this would be to big an effort. Instead we set up the following rules:

- all new tests requiring dynamic mocks use Mockito
- whenever you touch (fix, adapt, complete) an existing test that's using EasyMock: convert it to Mockito

Independently from that make sure your tests match our conventions. Information on how to best migrate JUnit3-style tests to JUnit4 can be found here.

Magnolia Provided Testing Support

In general we try to use the slimmest possible approach in tests. For testing basic functionality that does depend on itself there is no need to use mock objects. We can just test that method functionality straight away.

In cases we need to test objects that depend on other objects and this is where we use mock objects. Beware that they can just return what you want but the code behind won't be executed.

Imagine you need to test something that depends on a repository, well you can 'mock' a repository and it's content without having to create the repository itself.

```
MockSession session = SessionTestUtil.createSession("testWorkspace", propertiesString);
```

By doing this you can access to the nodes of you 'fake' repositoy as if it where real. But if you really need to use a real repository you can extend from RepositoryTestCase and use the methods declared in it.

Useful classes included in magnolia-core for building tests:

Here are some of the clases we have for Magnolia testing, you can find more classes in package info.magnolia.test

Type	Remark
ComponentsTestUtil	Set default implementations or instances when IoC can't be used yet.
MgnlTestCase	Sets up a basic environment for the test, loads beans and modules properties and initializes a mock context as the local context.
RepositoryTestCase	Superclass for Tests requiring access to a real jcr repo.
MockUtil	Util to create mock objects - especially MockContexts.
MockContext	Context where you can add Sessions and set a User
MockNode	Mock implementation of a jcr Node
MockContent	MockImplemenation of a Content

What to choose when

Here's a few examples that should help to understand what approach should preferably be used in what situation:

General Setup	Specifics	Preferred Approach	Example
Class under tests operates on JCR Node	few calls to common methods of the Node	MockNode if it supports those calls, Mockito mock else	
	need a simple hierarchy of Nodes	MockNode if it supports those calls, Mockito mock else	
	need a simple hierarchy of Nodes but with several properties	use SessionTestUtil to instantiate MockSession + MockNodes from propertiesStream or String	
	need a complex hierarchy of Nodes, real NodeTypes or issue real queries	use RepositoryTestCase	

Naming

A test class should end with `Test`. Typically, it will have the exact same name (and package) as the class it tests, with the `Test` suffix.

If you're writing a test class meant to be re-used as a super class for other tests, make it actually `abstract` and that should suffice to exclude it from execution. (it would otherwise fail, because it likely has no `@Test` methods). Another pattern we've used in the past (but I'd like to get away from it) is suffixing with `TestCase` (e.g `RepositoryTestCase`).

Methods: since we're using jUnit 4, there is no need to prefix method names with `test`. Choose a method name that describes what the test asserts. (`fooDoesBarWhenX()`)

Self tests: if you need to test methods of the test class itself (e.g utility methods that the tests use need to be themselves to validate their behaviour), do that in a method called `selfTest()`. If there are multiple such test methods, refer to the point above.

If you're testing a reusable test class, two options:
- don't suffix it with `TestCase`, but just with `Test` and use `selfTest()` methods.
- if that's not applicable, `FooBarTestCaseSelfTest` might seem a little over the top.

If you're testing an abstract class whose name is prefixed with `Abstract`, follow the same conventions. Name it `AbstractFooTest`. Add your `@Test` methods. If the test class isn't abstract, they'll be executed. Specify in the Javadoc if its meant to be reused for implementing tests for concrete `Foo` implementations. If the tests need a concrete an instance of `Foo`, implement it as an inner class of your tests and explicitly don't implement the methods that `AbstractFoo` doesn't implement and aren't relevant to the test (throw some exception) - this should be very easily generated and maintained by your IDE even if the interface of `AbstractFoo` changes.

In some cases, we also want to test external libraries. In particular, when a certain Magnolia feature relies on a specific behavior of such library, we might want to assert that it indeed does behave the way we think it does, and that it continues on doing so in their future releases. For such cases, selfTest methods can help, but if we're testing more than that, then we can envision `FooBarLibTest` classes (where `FooBar` is the class under test, and `LibTest` is the suffix) - we're "testing the libs of FooBar".

Resources

Mock Object: http://en.wikipedia.org/wiki/Mock_object

Mockito Documentation: http://docs.mockito.googlecode.com/hg/latest/org/mockito/Mockito.html

Mockito Examples: http://gojko.net/2009/10/23/mockito-in-six-easy-examples/