Core Integration Testing Framework

Work in progress.

Introduction

In the 1.5 branch, we upgraded to JUnit 4.4 from JUnit 3.8.1. This new version of JUnit is architecturally different from the old version used and it offers several advantages which we can exploit. These JUnit 4.4 advances in combination with Java 5 annotations and the new snapshotting feature resulted in some very interesting ideas for a new integration testi framework.

The material below presumes you already have a working knowledge of JUnit 4.4 as well as the older 3.8.1 version. If you need a primer there are several out there however we have one here as well.

What are integration tests?

Let's quickly define what an LDAP or Apache DS based integration test may mean to you:

#1 You have an LDAP driven application that requires a live LDAP server (or what feels like one) to be present and available to test your application as realistic a simulated environment as possible.
#2 You're an Apache Directory Developer or user and you want to quickly setup many test cases where the server is running and you can test various scenarios.

-
    - Users who want rapid resolutions can submit test cases
    - Developers - well do we have to say why we need em?

      The core framework simply starts the heart of the directory server without starting various network services. The core can still be used with the JNDI wrapper to perform tests as if they were occurring over the wire. The JNDI wrapper around the core mimics the standard JNDI LDAP interface regardless of the fact that calls are not using the network but are directly affecting entries on disk. Sometimes this might be preferred over network tests. If you want to perform network based tests then you want to use the server framework which parallels all the concepts here but uses different tags and runners as you can see here Server Integration Testing Framework. Regardless most of the concepts here will also apply to the server framework.

Pie In the Sky

Let's talk the possibilities in terms of "what if you could do .." scenarios. Just imagine the simplest case where you just want a standard instance of the core directory service up and running so you can just write a bunch of test methods. It would be real nice to just specify a runner to use and this runner would simply start the directory server and stop it:

```
@RunWith( CiRunner.class)
public class SampleITest
{
    public static DirectoryService service;

    @Test public void checkService0() throws Exception
    {
        assertNotNull( service );
        assertTrue( service.isStarted() );
    }
}
```

All we did was create a simple class (which note does not extend *TestCase*) with a single test method. This basic test method, checkService(), is tagged using the JUnit 4 **@Test** annotation to declare the method a test case. This is all thanks to JUnit 4.

Now closer inspection shows that there is no initialization of the static *DirectoryService* parameter yet this test should pass. The idea is to have the custom runner specified with the JUnit 4 **@RunWith** annotation drive the instantiation, configuration and startup of the core service. The test method is merely left to conduct it's tests.

This is pretty cool but wait! What happens if we have two tests and one changes the contents of directory? Won't the two tests collide to produce false negatives like in this scenario? :

```
@RunWith( CiRunner.class)
public class SampleITest
{
    static DirectoryService service;

    @Test public void testUserAdd() throws Exception
    {
         addUser( "uid=akarasulu,ou=users,ou=system", "secret" );
    }


    @Test public void testAddOuThenModifyAndCheck() throws Exception
    {
         addUser( "uid=akarasulu,ou=users,ou=system", "secret" );
         setUserPassword( "uid=akarasulu,ou=users,ou=system", "new_secret" );
    }
}
```

So if the server is started for this test class and these tests run. The one that runs second will fail because the user entry will have already been created. We could have written the test to delete the user if that user was present before adding that user, however this changes the characteristics of the test case.

It sure would be nice if the runner could give us the service in it's original state. And this is what we did before in the core-unit project with the JUnit 3.8.1 based unit tests using setUp() and tearDown() methods in a special AbstractTestCase. The biggest problem with this approach is the confusing overriding of the configuration by subclasses, and the needless time and CPU this wastes to shutdown, destroy, delete, create, and start new instances of the server.

With the new snapshot capability built into the core we can revert the server to a previous state that was tagged. The server then appears just like new to tests while the overhead of a shutdown, destroy, delete, create and start are not felt for each test. A 6-7 time reduction in the time taken for integration tests to run occurred on average using this revert feature.

So to answer the questions above, no the two tests will not collide. The runner will automatically tag the point at which a test starts. Then the test is run. When the next test is run the service is rolled back to it's original state before the server started running the first test.

Sometimes though tests may not need a roll back. Some may want to accumulate changes, others might want pristine installations with this expensive cycle, some may just need a restart with persistence of data, and others might not even require a service to be running at all.

Let's add a SetupMode annotation!

We're in Java 5 land so we can write our own annotation as a cue to the CiRunner. The runner will use this annotation either on the test class or on the test method to determine how to setup the service. If the value is set on the test class then all test methods will default to that value. When present on the test class and on a test method then the test method value is more specific and it overrides the test class SetupMode. If not specified on either a class or on a test method a system wide default is used: see the tip on defaults below. Initially we can allow the following enumerated values to the SetupMode annotation:

Value	Description
NOSERVICE	Nothing at all is done - a service in not required for tests but if one is running we don't care
PRISTINE	Stops running service if present, destroys their working directories, creates and configures a new instance of the service then starts it up for every test.
RESTART	Stops running service if present, then starts it without working directory destruction. The same service instance is used without reinstantiation or reconfiguration. If no service is present then one is created and started.
ROLLBACK	Rolls back the state of a running service live before a test is run. If an old service is present yet no t running it is cleaned up, a new service is created and started. If no service is present or running then a new service is created and started.
CUMULATIVE	If a service is running nothing is done. If a service has been stopped it is started. If a service is not present one is created and started.

So now we have the concept of a setup mode with test class level defaults that can be overridden for specific test methods which controls the setup life cycle. We have a good idea of the most common cases above and can add others if need be.

"SetupMode Defaults and Inheritence"

A system wide default is needed for the SetupMode. We chose to use ROLLBACK because of various reasons that are explained in the appendix. It's nice to override this default with specific test classes and test methods. Soon with the discussion of suites, we will encounter another level of potentially overriding the SetupMode. As you may have noticed some kind of SetupMode inheritence is begining to emerge. Everything inherits from the system wide default. Test classes may inherit their SetupMode from the test suite level, and test methods may inherit their SetupMode from a test class. Even though this has nothing to do with Java Class or Annotation inheritance it still is a form of custom property inheritance. Defaults coupled with inheritance helps reduce the verbosity of annotations. It can also help in other ways as well. We may attempt to leverage this for other aspects of the framework but we must be careful because it can cause problems to arise.

The CUMULATIVE SetupMode makes no sense without test ordering!

Yeah it makes no sense to accumulate changes. Why would we bother to do this unless one test needs the results of the other. Some would say if you need the results of another test just execute that test as a method at the start of your dependent test method. This works too. But then there is no reason for having a CUMULATIVE setup mode then though.

We need to figure out just how useful this is to us. However one option to use in conjunction with the CUMULATIVE SetupMode would be to enable a precondition/precursor/dependency kind of annotation. This annotation would tell the running to execute certain test methods before others. It would also have to detect potential cycles but this is not hard. Then it would establish a test plan and use that to drive the order of test method execution.

I really like this idea in general besides it's usefulness when paired with the CUMULATIVE SetupMode because it takes away that age old hack I used to use to order my tests. The tests are ordered by JUnit using the getDeclaredMethod() mechanism to query the methods in a class. This uses regular alphabetical word ordering but this may have odd results especially with internationalization. It's just a hack and this feature is long overdue in JUnit. It would be nice to have it here for our framework. At a minimum it can be used to run simpler tests first. Also if a simple test fails which another complex test depends on then the depending test can simply be aborted rather than being marked as a failure.

What about configuration and setup?

Obviously the pie in the sky picture overlooks some important things. You want to control how the server is configured and preload it with data if that is a requirement. These are things that can be handled elegantly using annotations and the power of a custom framework. Furthermore they can be leveraged for documentation purposes to better describe and document tests as well as what they are doing.

Using Reusable Factories (Builders?)

I always hated writing setUp and tearDown code in the old JUnit 3.8 way. You could not control it well enough for the entire class or individual tests. Then for complex tests you have mix a lot of configuration, startup and data loading all together and it gets messy. Then with inheritance using a base class things get a little better but it gets a bit confusing with life cycle aspects as you try to have subclasses tweak configuration a bit.

What if we divide and conquer to clearly separate service creation from any preparatory work needed to be done. Also instead of using inline methods that were not very useful what about using factories (or builder objects) to produce instances of the service. This is really nice for reuse and furthermore it allows us to utilize containers which essentially serve as builders that wire together services. Furthermore this can be coupled with the same concepts used before with the SetupMode annotation. Again an annotation can be used to specify the factory to be used, along with same pattern of defaults and inheritance.

So let's presume a Factory annotation exists which contains a Class. The system wide default is the default constructor of the DefaultDirectoryService. A special default factory is created for this which uses safe smart defaults for everything. Now a test suite, test class and test method at their respective levels can override the factory used. Furthermore their factories might be able to leverage other factories from their superiors.

Think about this case. You have a test suite A with test classes Foo and Bar in it. Suite A defines a factory which overrides the system default. Foo overrides this with it's own @Factory annotation, yet Foo's factory can access the factory of the suite. So if it wanted too Foo's factory can use the test suite factory to instantiate the base object that it will alter. This way Foo's does not have to reproduce the configuration code in the factory specified by it's superior Suite A. This reuse capability can come in very handy. Eventually a collection of factories will arise naturally enabling various features for testing.

Note that the CiRunner is responsible for figuring out which factory to use for each test that it runs. The CiSuite which we will get into later delegates to the CiRunners and allows then to access it's settings. The CiRunner must consider a parent test suite if it is included in one as well as the various SetupModes to determine when and how to use a Factory. Once it does so it can use an instance of a factory to create new instances governed by the setup mode.

Loading Data for Tests

Some test methods need data to already be present. It's hard to document these kinds of prerequisites of a test method, its test class or a suite if the load of information is done via code. It's also repetitive and error prone if done in code as well as tedious. It makes no sense to do this by hand in code.

With an **@ApplyLdif** annotation that takes an array of Strings arguments each element being a valid LDAP operation encoded as an LDIF, it's much better. This way the preload information is easy to document, explicitly stated and can be applied automatically by the runner without hand written code. Here's an example of what this would look like:

```
@ApplyLdifs ( {  // First entry
   "dn: ou=test1,ou=system\n" +
   "changeType: add\n" +
   "objectClass: organizationalUnit\n" +
   "ou: test1\n\n",  // Second entry
   "dn: ou=test2,ou=system\n" +
   "changeType: add\n" +
   "objectClass: organizationalUnit\n" +
   "ou: test2\n\n" })
@Test
public void deleteOu()
{
    delete( "ou=test,ou=system" );
}
```

Above there was a single LDIF applied but more than one can be included and separated using the annotation syntax for arrays. So a stream of operations can be conducted against the service before the test method is launched with control over the order of LDIF application.

Note that this annotation can be used at the Suite level, the Class level or the Method level.

This is neat but it will not scale well. It's obvious that it will get clunky after a while and should only be used for at most the trivial cases to prevent the java class file from really becoming an LDIF file.

So this leads us to yet another possibility. How about another tag to bulk load LDIFs from a file. We can use the **@ApplyLdifFiles** annotation to inform the runner of a set of LDIF files it should run in order. This annotation is also an array of Strings but it holds the URL for the file. Here's what it looks like:

```
@ApplyLdifFiles(
  {
     "file:///home/akarasulu/testdata/organizations.ldif",
     "file:///home/akarasulu/testdata/users.ldif"
  }
)
@Test
public void deleteEntries()
{
    delete( "o=Apache,ou=system" );
    delete( "o=Eclipse,ou=system" );
    delete( "o=OpenLDAP,ou=system" );

    delete( "uid=akarasulu,ou=users,ou=system" );
    delete( "uid=elecharny,ou=users,ou=system" );
    delete( "uid=szoerner,ou=users,ou=system" );
}
```

This mechanism can be used for more than just loading data. It can also be used to modify, rename, move and delete entries in the directory through the various changeTypes supported by the LDIF. This certainly is a powerful feature to have.

One question though does come up and it has to do with defaults and hierarchy. Should this aspect also follow the same patterns that the SetupMode, and Factory Annotations use or are their some particulars to consider specifically with this mechanism? Yes I think there are particulars because there are more options. First LDIFs can be combined instead of just being replaced. The LDIF preparatory data could be additive or can replace defaults as it does with the other properties. I really don't know what is best here. I can see it being useful in both ways.

"Mixing @ApplyLdifs with @ApplyLdifFiles can lead to problems"

The order of annotations are not preserved so ambiguity would be introduced if these two annotations were to be applied together. Which goes first and is there a dependency between the two? These kinds of questions make it even more difficult to implement this data loading feature in the same fashion that defaulting with inheritance is implemented with the other annotations.

What about Suites?

The role of test suites is very important. Suites will enable us to reduce integration test times even further. I suspect a natural dynamic that will emerge where we will begin to group tests into suites because they use the same configuration. This way the service only needs to be created and started once while using the **ROLLBACK** mode with the snapshot reverting feature. This can have a dramatic impact.

However to do this the default trickle down and the way the service life cycle is managed in the different scopes of testing from suite level, class level to test method level will need to be clearly defined. We need to be able to control situations where test classes will want to shutdown and destroy the service after all tests have already run. These tests must not do so if there is a shared instance that is being used by the suite for all tests and test classes otherwise there is little benefit.

Remote Execution

What does this mean?

1. It could mean using several machines to run the tests in parallel.
2. It could mean using a DirectoryService implementation which is an adapter to forward real over the wire requests to a remote LDAP server that may or may not be ApacheDS.

Both are cool concepts. I really thought of (2) first. As a user writing an LDAP driven application I may want to set up the unit tests to work with ApacheDS since it's embedded and easy. But I may also want to run the same tests once and a while (say before releasing my application) on OpenLDAP, Active Directory, or SUN DS. So it would be nice to have a generic DirectoryService object that acts as an adapter for over the wire requests. The tests think this is not over the wire since it's the core they are testing.

For most black box tests the DirectoryService is really used as a factory for creating JNDI contexts. This might change in the future but it's a good idea to think about how we can easily make the test code work against other service if that is something we desired.

Now option (1) is a really cool concept. It would be pretty easy to pull off too. Basically you need to have a service that sits on remote machines waiting for test requests. Since we know what tests and suites will need to run in a single thread since we want to do one test at a time per instance we can determine how to easily parallelize tests.

Dynamically with a scheduler we can swap out some runners for a remote runner which for all practical purposes runs in the same way as the local ones do. However instead the remote runner would contact the testing daemon on the remote host and request that it run a unit of tests that can be parallelized. This would require working directory synchronization but it would not be that hard to do at all. Furthermore a test daemon can always have a hot pristine service (sounds pornographic don't it?) ready and running for tests that can just start pounding it.

All good stuff - let's add that to the list of nice to have thingys.

Implementation

Tested Service States

The states of a service used for testing are important and must be tracked. We have to do a number of things based on the SetupMode, the scope of the test harness etc to determine what actions to take. The if-then-else conditions can balloon out of control especially when we combine it with the runner life cycle in JUnit. So it's a good idea for us to lists the states and draw a good old state diagram not to mention eventually using the State pattern in the implementation.

States

Below pristine means we know the service is untouched; the disk image is the same way it is exactly as it was after a fresh new install.

State	Description
NonExistant	There is no service instance.
StartedPristineState	The server is running and is clean as if it were just installed
StartedNormalState	The server is running and may have been changed since the last pristine state

Actions

Actions represent the operations performed during the testing process which transition the tested service's state. Hence they are transition arcs on a state diagram. The actions are defined below:

Action	Description
CREATE	The act of creating and configuring a service which boils down to using an instance of the DirectoryServiceFactory to get a ready to startup instance of the service.
DESTROY	The act of nulling out a reference to an already shutdown service and invoking the garbage collector.
CLEANUP	The act of deleting the working directory associated with a configured instance.
STARTUP	The act of starting a service which is not running.
SHUTDOWN	The act of stopping a service which is running.
TEST	The act of running tests against the directory service.
REVERT	The act of rolling back the server state to a point where the changes made by any tests are reversed.

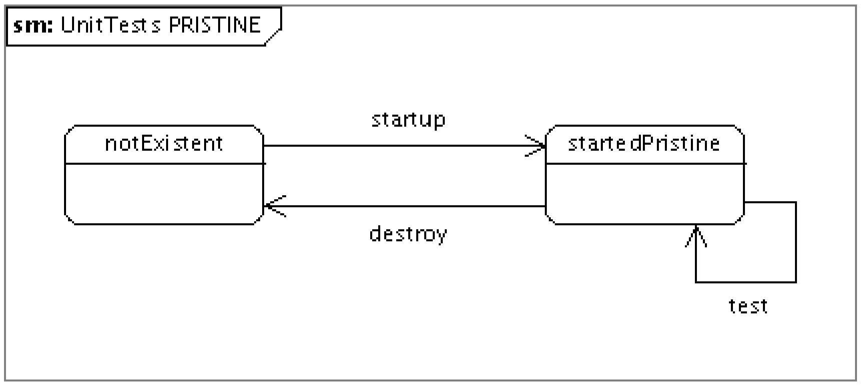
State Diagram

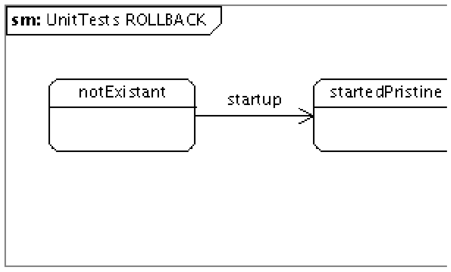
The following table expose all the different modes and states, with the associated action transitions :

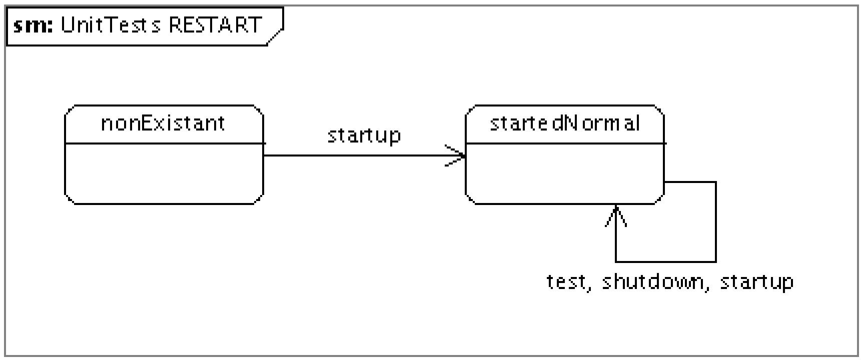
	NotExistant	StartedPristine	StartedNormal
	action / next state	action / next state	action : next state
PRISTINE	NotExistant.create / NotExistant NotExistant.cleanup / NotExistant NotExistant.startup / StartedPristine StartedPristine.test	**invokeTest** StartedPristine.shutdown / StartedPristine StartedPristine.cleanup / StartedPristine StartedPristine.destroy / NonExistant	
NOSERVICE			
ROLLBACK	NotExistant.create / NotExistant NotExistant.cleanup / NotExistant NotExistant.startup / StartedPristine StartedPristine.test	**tag** **invokeTest** / StartedNormal StartedNormal.revert / StartedNormal	**tag** **invokeTest** / StartedNormal StartedNormal.revert / StartedNormal

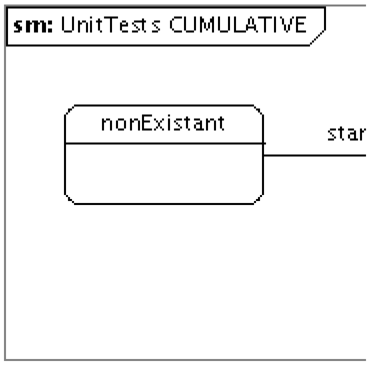
RESTART	NotExistant.create / NotExistant NotExistant.startup / StartedNormal StartedNormal.test		**invokeTest** StartedNormal.shutdown / StartedNormal StartedNormal.startup / StartedNormal
CUMULATIVE	NotExistant.create / NotExistant NotExistant.startup / StartedNormal StartedNormal.test		**invokeTest**

Now let's mix the states and the actions together:









Appendix

Appendix A: System Wide SetupMode Default

The default setup mode to use is **ROLLBACK**. Why you might ask is it not **NOSERVICE**? The user shown us their intentions and has made a conscious decision to conduct tests with a running core service by selecting this Runner in the first place via the **@RunWith** JUnit annotation. So by default it makes sense to just give them that if they decide to use it in any of the tests contained which is most likely the case otherwise why would they bother to use this runner. The thing is some tests may need the service and some might not even use it. If the **ROLLBACK** mode is used then there is no cost incurred for having a running server in the background. The server will be used anyway at some point by some tests. When not used by some tests there are no rollbacks and so the service just sits waiting for tests to use it.

- If the default was **NOSERVICE** the user would have to do more work specify a mode on each test method that needs the core service running.
- If the default was **PRISTINE** then those tests not needing a service up would shutdown existing servers, clean up, create a new one and start the new service. This process costs about 2-3 seconds churns the disk and costs memory. Turning off this default would require more effort on the user when we already know their intension's.
- If the default was **RESTART** then the impact would be similar to the pristine case with perhaps much less waste but still it would be very inefficient.
- If the default was **CUMULATIVE** then some test that may delete and create the same entries, often not worrying about cleanup would collide causing false negatives.

As you can see the best option for the default is the **ROLLBACK** mode. When a suite is present however mode inheritance can be utilized to override this default.