Unit Testing 101

Author: Alex Huang

The following is a FAQ on how to do write unit tests in cloud stack.

Where are the unit tests?

Within each source directory, there's also a test directory that contains the tests for that jar file. For example, utils/src and utils/test. If you have files other than source files, they should go into "test/resource" directory.

What do we use for unit testing?

www.junit.org

How do I write a suite of tests?

A suite of tests is a set of tests that are run together. Take a look at DbUpgrade22Test.java
. Junit has annotations you can use for that.

How to run the unit tests?

To run the entire unittest suite:ant unittestTo run just one test that you wroteant test -Dtest=filename without path and without ".java"You can also run the unit tests individually from eclipse. Kelven can update this space on how to do this.

Where do the unittest outputs go to?

All outputs are confined to "unittest" directory under the directory where you are running.

Where do the logs of my source code go to?

All logging in the source code being tested go into the unittest/TEST* files. All logs are embedded inside. If you want to change that, you can modify utils /test/resource/log4j.xml. Currently, there's no separate file appender there. Everything goes to console which gets captured by junit and put into its output file.

How do I get a DB connection to use in my tests?

Transaction.getStandaloneConnection(). This returns a raw DB connection so you're responsible for closing out any resources, including ResultSet, Connection, PreparedStatements, etc.Connection conn = Transaction.getStandaloneConnection();
PreparedStatement pstmt = conn.prepareStatement("SELECT 1");
ResultSet rs = pstmt.executeQuery();
assert (rs.next() && rs.getInt(1) == 1) : "How come I'm not getting back 1?";
rs.close();
pstmt.close();
conn.close();

If I need to execute some SQL scripts to setup the database, what should I do?

1. Write your SQL script and put it under test/resource/[your own directory name]
2. Call DbTestUtils.executeScript("test/resource/[your own directory name]/[your script filename]");

For example, assuming you have a script file in "server/test/resource/VerionDaoImplTest/2.1.7/2.1.7.sample.sql", you will write the following. Note that "server" is missing in the path.DbTestUtils.executeScript("test/resource/PreviousDatabaseSchema/2.1.7/2.1.7.sample.sql");

What if the component to be tested needs to be injected?

ComponentLocator has a static method to inject classes. For example, if you're testing VersionDaoImpl, you can do the following.VersionDao dao = ComponentLocator.inject(VersionDaoImpl.class);

What if the component tested needs to have other components injected?

There is a MockComponentLocator that allows you go add all of the components you needed. For example, if you need to test a FirstFitPlanner which has the following components injected.

@Inject private HostDao _hostDao;

@Inject private CapacityDao _capacityDao;

@Inject private DataCenterDao _dcDao;

```
    @Inject private HostPodDao _podDao;

    @Inject private ClusterDao _clusterDao;

    @Inject DetailsDao _hostDetailsDao = null;

    @Inject GuestOSDao _guestOSDao = null;

    @Inject GuestOSCategoryDao _guestOSCategoryDao = null;

    @Inject CapacityManager _capacityMgr;

    @Inject ConfigurationDao _configDao;MockComponentLocator locator = new MockComponentLocator("management-server");
locator.addDao(HostDaoImpl.class);
locator.addDao(CapacityDaoImpl.class);
locator.addDao(DataCenterDaoImpl.class);
locator.addDao(HostPodDaoImpl.class);
locator.addDao(ClusterDaoImpl.class);
locator.addDao(DetailsDaoImpl.class);
locator.addDao(GuestOSDaoImpl.class);
locator.addDao(GuestOSCategoryDaoImpl.class);
locator.addManager(CapacityManagerImpl.class);
locator.addDao(ConfigurationDaoImpl.class);
locator.makeActive(new DefaultInterceptorLibrary());

FirstFitPlanner planner = ComponentLocator.inject(FirstFitPlanner.class);
...testing code...
```

Note that by doing this, you have complete control of the components that you are injecting. For example, let's say you thought CapacityManagerImpl is too complicated and, instead, you want to inject a mocked up CapacityManager that supplies the values you want to test. You can do "locator.addManager(MockCapacityManagerImpl.class)" instead. You'll have to write a MockCapacityManagerImpl class file of course.

I want to test against all or most of the components, should I do this using unit testing?

No. Unit testing is for specific testing to force issues to come up early. If you need to entire components.xml or components-premium.xml for your tests, what you really need is to start a server and have a predefined set of testcases run as the web client. That's not part of the unit testing. We need to develop other means to do that. Once completed, we'll make this known on this page.

Are there any examples?

server/test/com/cloud/upgrade/AdvanceZone217To224UpgradeTest.java

Some of the current unit tests uses components-premium.xml and tries to find log4j-cloud.xml. Should I use these as examples?

You can but largely there's no reason. components-premium.xml or components.xml is a way to integrate our code together with different components and therefore the files can change quickly. Most of the time when you run unit tests, you want to specify exactly which components are necessary to test the component your testing. Sometimes, you might want even want to mock up certain components so using components.xml is not useful. MockComponentLocator gives you the best way to do this by allowing you to construct the environment before running your own test. The same goes for log4j-cloud.xml. Why rely on a log configuration for production environment when you can specify your own for unit tests?

How do I make changes to the unit test task in ant?

build/developer.xml