**Unit Test**

Table of Contents

- Table of Contents
- Overview
- Example
  - How to set environment variables
- Debugging
- Various Test Techniques
  - Stubbing
  - HTML scraping
  - Submitting forms
  - Dealing with problems in JavaScript
  - Configuration round-trip testing
  - Web page assertions
  - Doing things differently in JavaScript when it runs as unit test
  - TestCase as a RootAction
  - Testing authorization behaviours
  - Custom builder
  - Registering Extensions during tests
  - Problems and hacks for running tests in Windows
- Test harness annotations
  - Informational annotations
    - @Issue("JENKINS-12345")
    - @For(FooBar.class)
    - @Url(http://internet.org)
    - @Email(http://....)
  - Test environment annotations
    - @Recipe(SetupClass)
    - @LocalData
    - @PresetData(SecurityPreset)
    - @WithPlugin(NameOfPlugin)
    - @WithoutJenkins
    - @WithTimeout(TimeoutInSeconds)
- HudsonTestCase (JUnit 3)

---

**Overview**

Jenkins comes with a test harness built around JUnit to make test development simpler. This harness provides the following features:

1. Starts an embedded servlet container so that the test code can exercise user interaction through HTTP and assert based on the outcome.
2. **HtmlUnit** with a bit of enhancements allows you to productively test HTTP/UI interaction.
3. Prepares and tears down a fresh Jenkins instance for each test case. So each test method will run in a fresh environment, isolated from other tests.
4. Test code can also directly access Jenkins object model. This allows tests to assert directly against the internal state of Jenkins, as well as perform some operations directly, as opposed to doing so only through the HTML scraping.
5. Declarative annotations to specify the environment in which a test will be run. For example, your test method can request that Jenkins shall be started with a certain **HUDSON_HOME** contents.
6. Declarative annotations to maintain association between tests and bugs/discussions.

If you are using the Plugin Parent POM 2.3 or later the version of the Jenkins Test Harness (which starting in version 2.0 is an artifact independent from Jenkins Core) to be used for the tests can be configured using the `jenkins-test-harness.version`. For 1.x versions of the Plugin POM the version used is the same of the parent POM which is equal to the Jenkins baseline version. The recommended version of Jenkins Test Harness is 2.1 or above unless your plugin depends on Jenkins < 1.580.

**Example**

The following code shows a very simple test case. Your test will use **JenkinsRule** to provide test fixtures for some of those features outlined above.

Each test method will start with a fresh temporary installation of Jenkins. The `first` method doesn't request any particular Jenkins data set to seed Jenkins, so it will start from an empty installation.

The test then proceeds to create a new project and set it up. As you can see, the code directly talks to the in-memory Jenkins object model. (There is the `jenkins` variable defined in JenkinsRule that you can use for access.) While you can do the same by emulating the user HTTP interaction through HtmlUnit, this way is often a convenient way to prepare an environment for the code that you want to test.

The test code in this example then switch to HtmlUnit to emulate the UI interaction and verify that we get results that we expect.

When a test completes, the temporary Jenkins installation will be destroyed.
import org.jvnet.hudson.test.JenkinsRule;
import org.apache.commons.io.FileUtils;
import hudson.model.*;
import hudson.tasks.Shell;
import org.junit.Test;
import org.junit.Rule;
public class AppTest {
    @Rule public JenkinsRule j = new JenkinsRule();
    @Test public void first() throws Exception {
        FreeStyleProject project = j.createFreeStyleProject();
        project.getBuildersList().add(new Shell("echo hello"));
        FreeStyleBuild build = project.scheduleBuild2(0).get();
        System.out.println(build.getDisplayName() + " completed");
        // TODO: change this to use HtmlUnit
        String s = FileUtils.readFileToString(build.getLogFile());
        assertThat(s, contains("+ echo hello"));
    }
}

How to set environment variables

When preparing your virtual test environment, you may wish to simulate Jenkins environment variables that can be set on the Jenkins configuration page. Adding environment variables to a Jenkins instance before a test is simple, as the example below demonstrates.

```java
public void setEnvironmentVariables() throws IOException {
    EnvironmentVariablesNodeProperty prop = new EnvironmentVariablesNodeProperty();
    EnvVars envVars = prop.getEnvVars();
    envVars.put("sampleEnvVarKey", "sampleEnvVarValue");
    j.jenkins.getGlobalNodeProperties().add(prop);
}
```

Debugging

Your IDE will most likely have the ability to select a single JUnit test and execute it in the debugger. Otherwise you can run `mvn -Dmaven.surefire.debug -Dtest=hudson.SomeTest` to accomplish this.

To debug slaves launched by Jenkins, set `-Dorg.jvnet.hudson.test.HudsonTestCase.slaveDebugPort=PORT` to the system property, or from your test case set `JenkinsRule.SLAVE_DEBUG_PORT` to a non-0 value.

Various Test Techniques

Stubbing

Sometimes you want to have quick tests which don't start up a 'full' Jenkins instance - as JenkinsRule does - as this can take some time. In that case you shouldn't have your test classes use JenkinsRule. (Or you can use @WithoutJenkins on certain methods.)

As creating most Jenkins core classes without a Jenkins instance is unfortunately not easy, stubbing can come in handy. One excellent stubbing and mocking framework is e.g. Mockito. For example, if you want to stub a build with a certain result you could do:

```java
AbstractBuild build = Mockito.mock(AbstractBuild.class);
Mockito.when(build.getResult()).thenReturn(Result.FAILURE);
```

See Mocking in Unit Tests for more info.

HTML scraping
If you'd like to test the HTML generated by Jenkins, XPath test is often convenient.

```java
HtmlPage page = j.createWebClient().goTo("computer/test/");
HtmlElement navbar = page.getElementById("left-top-nav");
assertEquals(1, navbar.selectNodes(".//a").size());
```

### Submitting forms

Submitting a form through HtmlUnit in Jenkins is a bit trickier than it should be. This is because Structured Form Submission is done in JavaScript at the form submission time, and the submit button is usually decorated by YUI, which internally converts `<input type="submit">` to `<button>`.

So you'd have to look for the corresponding HtmlButton element, then use that to call the submit method, like this:

```java
HtmlPage configPage = j.createWebClient().goTo("configure");
HtmlForm form = configPage.getFormByName("config");
form.submit((HtmlButton) last(form.getElementsByTagName("button")));
```

### Dealing with problems in JavaScript

When JavaScript throws an exception and causes a test to fail, it will often print a long chain of nested exceptions. Notice that the stack trace includes synthesized stack frames for JavaScript, which is different from the actual Java execution stack.

The original HtmlUnit doesn't really do a good job of chaining all exceptions together, so we are patching HtmlUnit to make sure it retains the full stack trace leading up to the root cause. If you found a case where this chain is broken, please file a bug.

If you set a break point in Java code, and if your execution suspends while its directly/indirectly invoked through JavaScript, you can use JenkinsRule.jsDebugger to introspect JavaScript call stack and its local variables. This is often very useful in identifying where in JavaScript things went wrong.

### Configuration round-trip testing

One of the very useful test idioms for Builder, Publisher, and anything that has configuration forms is the round-trip testing. This test goes like this:

1. Programmaticaly construct a fully populated instance
2. Request a configuration page via HtmlUnit
3. Submit the config page without making any changes
4. Verify that you still have the identically configured instance

```java
FreestyleProject p = j.createFreestyleProject();
YourBuilder before = new YourBuilder("a","b",true,100);
p.getBuildersList().add(before);

j.submit(j.createWebClient().getPage(p,"configure").getFormByName("config"));
YourBuilder after = p.getBuildersList().get(YourBuilder.class);

j.assertEqualBeans(before,after,"prop1,prop2,prop3,...");
```

This test ensures that your configuration page is properly pre-populated with the current setting of your model object, and it also makes sure that the submitted values are correctly reflected on the constructed model object. To be really sure, do this twice with different actual values — for example, you should try a non-null string and null string, true and false, etc., to exhaust representative cases.

### Web page assertions

HtmlUnit has a `WebAssert` class that can be used for simple assertions on HTML pages.

To assert that the System configuration page contains the CVS SCM configuration entry:

```java
HtmlPage page = j.createWebClient().goTo("configure");
WebAssert.assertElementPresent(page, "hudson-scm-CVSSCM");
```

To find Elements by name= vs. id=, use `page.getElementsByTagName(name)`.
An example from the ironmqNotifier plugin.

```java
HtmlPage page = j.createWebClient().goTo("configure");
assertEquals("Expect to find one instance of this name", page.getElementsByName("ironmqNotifier").size(), 1);
assertEquals("Expect to find one instance of this name", page.getElementsByName("org-jenkinsci-plugins-ironmqnotifier-IronMQNotifier").size(), 1);
```

See note on pom.xml below if this example gives you ClassNotFoundExceptions.

When changing from 1.625.1 to 1.625.2, there appears to be a difference in the support library for HtmlPage.

When parsing pages for Number (Long or Integer), you may get the following error:

```java
java.lang.ClassCastException: com.gargoylesoftware.htmlunit.html.HtmlNumberInput cannot be cast to com.gargoylesoftware.htmlunit.html.HtmlTextInput
```

The supporting library appears to need `HtmlNumberInput` instead of `HtmlTextInput` to function correctly with numbers.

Make the following change...

Before

```java
HtmlTextInput inputElement = page.getElementById("_.defaultExpirySeconds");
```

After

```java
HtmlNumberInput inputElement = page.getElementById("_.defaultExpirySeconds");
```

Note; If you wish to move back to a version prior to 1.625.2, you will need to modify your Tests as the previous library associated with HtmlPage will not recognize `HtmlNumberInput` as part of the class.

Doing things differently in JavaScript when it runs as unit test

JavaScript in Jenkins can test whether it's running in the unit test or not by checking the global `isRunAsTest` variable defined in `hudson-behavior.js`, which is included in all the pages. This can be used to disable some ajax operations, for example. Obviously, this has to be used with caution so that tests will continue to test the real thing as much as possible.

**TestCase as a RootAction**

An instance of the test case being executed is added to Jenkins' URL space as `/self` because JenkinsRule is itself a `RootAction`. Among other things, this enables your test class to define Jelly views, and invoke it like `j.createWebClient().goTo("self/myview")`.

**Testing authorization behaviours**

Unit test harness contains a SecurityRealm implementation suitable for unit tests. This can be installed to as follows:

```java
j.jenkins.setSecurityRealm(j.createDummySecurityRealm());
```

This virtual security realm allows login attempts by any user name so long as its password is exactly the same as the user name. `WebClient.login` method provides a convenient method that allows you to login a session object.

**Custom builder**

You can extend `TestBuilder` to write a one-off builder that can coordinate with your test. This is often convenient to stage things up for testing your `Publisher`, for example by placing files in the workspace, etc.
FreeStyleProject project = j.createFreeStyleProject();
project.getBuildersList().add(new TestBuilder() {
    public boolean perform(AbstractBuild<?, ?> build, Launcher launcher,
        BuildListener listener) throws InterruptedException, IOException {
        build.getWorkspace().child("abc.txt").write("hello","UTF-8");
        return true;
    }
});
project.scheduleBuild2(0);

OneShotEvent is also often an useful companion so that the thread that runs your test method and the thread that runs the build can coordinate — for example, the following program blocks the main thread until the build starts.

final OneShotEvent buildStarted = new OneShotEvent();

FreeStyleProject project = j.createFreeStyleProject();
project.getBuildersList().add(new TestBuilder() {
    public boolean perform(AbstractBuild<?, ?> build, Launcher launcher,
        BuildListener listener) throws InterruptedException, IOException {
        buildStarted.signal();
        ...
        return true;
    }
});
project.scheduleBuild2(0);
buildStarted.block(); // wait for the build to really start

Registering Extensions during tests

During the test, one might want to register extensions just during that particular test, for example to assist the test scenario. You can do this by defining such extension as a nested type of your test case class and put TestExtension instead of Extension.

It lets you tie an extension to just one test method, or all test methods on the same class.

Problems and hacks for running tests in Windows

See Unit Test on Windows.

Test harness annotations

There are several annotations in the Jenkins test framework.

Informational annotations

@Issue("JENKINS-12345")
Related issue id in tracker.

@For(FooBar.class)
Production classes that tests are related to. Useful when the relationship between the test class name and the test target class is not obvious.

@Url(http://internet.org)
URL to the web page indicating a problem related to this test case.

@Email(http://.....)
URL to the e-mail archive. Look for the e-mail in http://jenkins.361315.n4.nabble.com/Jenkins-users-f361316.html or http://jenkins.361315.n4.nabble.com/Jenkins-dev-f387835.html
Test environment annotations

@Recipe(SetupClass)

The specified class will be used to set up the test environment using HudsonTestCase.

@LocalData

Runs a test case with a data set local to test method or the test class.

This recipe allows your test case to start with the preset HUDSON_HOME data loaded either from your test method or from the test class.

For example, if the test method is org.acme.FooTest.bar(), then you can have your test data in one of the following places in resources folder (typically src/test/resources):

- Under org/acme/FooTest/bar directory (that is, you'll have org/acme/FooTest/bar/config.xml), in the same layout as in the real JENKINS_HOME directory.
- In org/acme/FooTest/bar.zip as a zip file.
- Under org/acme/FooTest directory (that is, you'll have org/acme/FooTest/config.xml), in the same layout as in the real JENKINS_HOME directory.
- In org/acme/FooTest.zip as a zip file.

Search is performed in this specific order. The fall back mechanism allows you to write one test class that interacts with different aspects of the same data set, by associating the dataset with a test class, or have a data set local to a specific test method.

The choice of zip and directory depends on the nature of the test data, as well as the size of it.

@PresetData(SecurityPreset)

Runs a test case with one of the preset HUDSON_HOME data set:

- NO_ANONYMOUS_READACCESS - Secured Jenkins that has no anonymous read access. Any logged in user can do anything.
- ANONYMOUS_READONLY - Secured Jenkins where anonymous user is read-only, and any logged in user has a full access.

@WithPlugin(NameOfPlugin)

Installs the specified plugin before launching Jenkins in the test. For now, this has to be one of the plugins statically available in resources */plugins/NAME*.

@WithoutJenkins

Runs a test case without create and tear down a Jenkins instance.

This Recipe has to be used in test methods that do not use a 'full' Jenkins instance, but are part of a test class that uses JenkinsRule or HudsonTestCase to prevent performance issues.

@WithTimeout(TimeoutInSeconds)

Runs a test case with the given timeout expressed in seconds.

HudsonTestCase (JUnit 3)

HudsonTestCase is available for JUnit 3 tests. The functionality is similar to JenkinsRule.

pom.xml

When seeing the following error: java.lang.NoClassDefFoundError: org/hamcrest/MatcherAssert when running mvn:test if you are using WebPage Assertions per the above example.

It may be necessary to add org.hamcrest / hamcrest-all to your pom.xml file to avoid a matcher error. The cause is not yet known but this workaround will allow you to continue to learn how to create appropriate tests.