The %UnitTest Framework for InterSystems IRIS

Version 2024.1
2024-07-02

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

The %UnitTest Framework for InterSystems IRIS
InterSystems Version 2024.1 2024-07-02

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

List of Tables

1

About the InterSystems IRIS %UnitTest Framework

%UnitTest is the InterSystems IRIS unit testing framework. Developers familiar with xUnit frameworks will find the structures contained within %UnitTest familiar:

- Create unit tests by extending the %UnitTest.TestCase class, adding test methods. See Extending the %UnitTest.TestCase Class for details.
- Execute preparation and cleanup tasks by adding code to special cleanup and preparation methods in the %UnitTest.TestCase class.

  See %UnitTest.TestCase Class Preparation and Cleanup Methods for details.
- Use the **RunTests()** method in the %UnitTest.Manager class to execute your tests. The general results appear in your terminal window. See Executing Unit Tests Using the %UnitTest.Manager Methods for details.
- View the test results web page in the Management Portal for more detailed information. See Viewing %UnitTest Reports in the Management Portal for details.

The %UnitTest package includes the following classes:

- TestCase — Extend this class to create your testing class, then add class methods that contain your unit tests.
- Manager — Contains methods to execute your unit tests.
- Report — Controls the output from testing, including a test results web page.

2

Creating Test Cases: The %UnitTest.TestCase Class

This is the general workflow to set up unit tests using the %UnitTest framework:

1. Extend the %UnitTest.TestCase class, adding one test method for each method to be tested. Test method names must begin with the word **Test**. See Extending the %UnitTest.TestCase Class.
2. A single test method can contain multiple tests. Typically a test method will contain one test for each aspect of the method to be tested. Within each test method, devise one or more tests using the *$$$AssertX* macros. Typically, the macro will call the method to be tested, comparing its output to some expected value. If the expected value matches the macro output, the test will be considered successful. See Macros of the %UnitTest.TestCase Class.
3. Add code to the preparation and cleanup methods to perform needed tasks. For example, if a test seeks to delete an element from a list, that list must first exist and it must contain the element to be deleted. See %UnitTest.TestCase Class Preparation and Cleanup Methods.

   Note: Preparation methods and cleanup methods are also often called setup methods and teardown methods.

2.1 Extending the %UnitTest.TestCase Class

Create a class that extends %UnitTest.TestCase to contain the test methods that execute your unit tests. This process is designed to be flexible, to accommodate your particular testing needs.

Most likely, you will add test methods, and you might add properties as well. Test methods will be executed by the **RunTests()** method from the %UnitTest.Manager class, which looks for and executes methods whose names begin with 'Test'. You can add other helper methods to your class, but a method will be run as a unit test when you call **RunTests()** only if its name begins with 'Test'.

Note: Test methods are executed in alphabetical order, so, for example, **TestAssess()** would be executed before **TestCreate()**.

Within a test method, create one or more tests. Use an *$$$AssertX* macro for each test. See Macros of the %UnitTest.TestCase Class for details about *$$$AssertX* macros.

You may decide to create a test method for each class method you wish to test. For example, suppose your class MyPackage.MyClassToBeTested contains a method **Add()**, which calls for multiple tests — you might want to create test method **MyTests.TestAdd()** to contain the code that executes the needed tests.

You may also wish to test object instances. In this case, you would create a method like **MyTests.TestMyObject()**, which could contain tests to make sure the object's properties and functionality are correct.

In addition to creating test methods, you may wish to create properties in your extended class. This enables your test methods to share information. Consider the following points when adding properties:

- Declare your custom properties in the class itself.
- Set the properties by adding code to the preparation methods **OnBeforeOneTest()** and **OnBeforeAllTests()**, using `..<property>` syntax.
- Access the properties by adding code to your test methods and/or to the cleanup methods **OnAfterOneTest()** and **OnAfterAllTests()**, using `..<property>` syntax.

  Note: For example, if your custom property is called PropertyValue, you would set it or access it using `..PropertyValue`.

2.1.1 Example: Extended %UnitTest.TestCase Class

```
Class MyPackage.MyClassToBeTested
{
 Method Add (Addend1 as %Integer, Addend2 as %Integer) As %Integer
 {
  Set Sum = Addend1 + Addend2
  Return Sum
 }
}

Class MyPackage.MyTests Extends %UnitTest.TestCase
{
 Method TestAdd()
 {
  do $$$AssertEquals(##class(MyPackage.MyClassToBeTested).Add(2,3),5, "Test 2+3=5")
  do $$$AssertNotEquals(##class(MyPackage.MyClassToBeTested).Add(3,4),5, "Test 3+4 '= 5")
 }
}
```

2.2 Macros of the %UnitTest.TestCase Class

Within each of your test methods, use one of the following `$$$AssertX` macros to test each testable aspect of the class method. For example, if a test method is designed to test the **Add()** method, it might contain a test, using `$$$AssertEquals`, to ensure that it adds 2+3 equals 5, and a second test, using `$$$AssertNotEquals`, to ensure that it does not add 3+4 equals 5.

Select the macro that best matches the desired test outcome. Another way to think of this principle is to write your test from the perspective that the assertion succeeds. If you expect two values to be equal, use `$$$AssertEquals`; if you expect the values not to be equal, use `$$$AssertNotEquals`.

A test fails if the specified `$$$AssertX` macro returns false; otherwise the test passes.

The `$$$AssertX` macros can take the following arguments:

- *arg1* — Typically either the output from the method being tested or a value calculated from that output.
- *arg2* — When present, a value compared by the macro to *arg1*.
- *test_description* — A string that appears in the displayed test outcome listing, and describes what the macro has tested. This has no effect on the outcome of the test. Don't forget that this argument can include concatenations, variables, and methods. For example, its value could be:

  "Failed to create" _ maxObjects _ "objects: " _ $system.Status.GetErrorText(status)

$$$AssertEquals (arg1, arg2, test_description)

Returns true if *arg1* and *arg2* are equal.

```
do $$$AssertEquals (##class(MyPackage.MyClassToBeTested).Add(2,3), 5, “Test Add(2,3) = 5”)
```

$$$AssertNotEquals (arg1, arg2, test_description)

Returns true if *arg1* and *arg2* are not equal.

```
do $$$AssertNotEquals (##class(MyPackage.MyClassToBeTested).Add(3,4), 5, "Test Add(3,4) '= 5")
```

$$$AssertStatusOK (arg1, test_description)

Returns true if the returned status code is 1.

```
do $$$AsserStatusOK(##class(MyPackage.MyClassToBeTested).SaveContact(valid_contact_ID),
      "Test that valid contact is saved")
```

$$$AssertStatusNotOK (arg1, test_description)

Returns true if the returned status code is not 1.

```
do $$$AssertStatusNotOK(##class(MyPackage.MyClassToBeTested).SaveContact(invalid_contact_ID),
      "Test that invalid contact is not saved")
```

$$$AssertTrue (arg1, test_description)

Returns true if the expression is true.

```
do $$$AssertStatusTrue(##class(MyPackage.MyClassToBeTested).IsContactValid(valid_contact_ID),
      "Test that valid contact is valid")
```

$$$AssertNotTrue (arg1, test_description)

Returns true if the expression is not true.

```
do $$$AssertStatusNotTrue(##class(MyPackage.MyClassToBeTested).IsContactValid(invalid_contact_ID),
      "Test that invalid contact is not valid")
```

$$$AssertFilesSame (arg1, arg2, test_description)

Returns true if two files are identical.

```
do $$$AssertFilesSame(##class(MyPackage.MyClassToBeTested).FetchFile(URL), control_file,
      "Test that fetched file is identical to control file")
```

$$$AssertFilesSQLUnorderedSame (arg1, arg2, test_description)

Returns true if two files containing SQL query results contain the same unordered results.

```
do $$$AssertFilesSQLUnorderedSame(output.log,reference.log,"Comparing output.log to reference.log")
```

$$$AssertSuccess(test_description)

Unconditionally log success. This assertion is intended to replace the convention of passing 1 to `$$$AssertTrue`

$$$AssertFailure(test_description)

Unconditionally log failure. This assertion is intended to replace the convention of passing 0 to `$$$AssertTrue`.

$$$AssertSkipped(test_description)

Logs a message that the test has been skipped for the reason described in *test_description*. This might be used, for instance, if the preconditions for a test have not been met.

Note: **OnBeforeAllTests()** does not support this macro. Calls to `$$$AssertSkipped` in **OnBeforeAllTests()** could result in false positives.

`$$$LogMessage (message)`

Writes the value of *message* as a log entry, independent of any particular test. This can, for instance, be very useful for providing context and organization in your log.

```
do $$$LogMessage("-- ALL TEST OBJECTS CREATED -- ")
```

Note: For the latest list of macros, see %UnitTest.TestCase in the Class Reference.

2.3 %UnitTest.TestCase Class Preparation and Cleanup Methods

%UnitTest.TestCase includes preparation and cleanup methods for your tests. You can add code to these methods to perform preparation tasks such as creating database connections or initializing a database with test data, or to perform cleanup tasks such as closing database connections or restoring the state of the database.

OnBeforeOneTest()

Executes immediately before each test method in the test class.

OnBeforeAllTests()

Executes only once, before any test methods in the test class.

OnAfterOneTest()

Executes immediately after each test method in the test class.

OnAfterAllTests()

Executes only once, after all of the test methods in the test class have executed.

2.3.1 Example: Preparation Method

The code in this method will execute once, before execution of the test suite. It creates a single contact for use during testing. To execute preparation tasks multiple times, once before each test in the suite, add code to **OnBeforeOneTest()** instead.

```
Method OnBeforeAllTests()
{
  Do ##class(MyPackage.Contact).Populate(1)
  Return $$$OK
}
```

2.3.2 Example Cleanup Method

The code in this method will execute once, after execution of the entire test suite. It kills all contacts in the extent once testing is complete. To execute cleanup tasks multiple times, once after each test in the suite, add code to **OnAfterOneTest()** instead.

```
Method OnAfterAllTests()
{
  Do ##class(MyPackage.Contact).%KillExtent()
  Return $$$OK
}
```

3

Executing Unit Tests Using the %UnitTest.Manager Methods

Launch tests using the methods included in the %UnitTest.Manager class.

This is the general workflow to execute unit tests using the %UnitTest framework:

1. Inform the system where to find your tests by setting the ^*UnitTestRoot* global:

   ```
   USER>set ^UnitTestRoot = "C:\UnitTests"
   ```

2. Execute your tests, using the **RunTests()** or **DebugRunTestCase()** method of the %UnitTest.Manager class:

   ```
   USER>do ##class(%UnitTest.Manager).RunTests("MyTests")
   ```

3. View the results of your tests.

Note: By default, **RunTests()** loads any test classes it finds within the ^*UnitTestRoot* directory, compiles them, executes any tests they contain, and deletes them from memory.

However, this may not be the most efficient paradigm when you are developing your code. That is, you may not want to reload and recompile your tests every time you make a small change to the method being tested. As a result, unit test classes are often stored externally. You can use the arguments of the RunTests() method to explicitly control whether to load tests and from where, whether to delete them, and other considerations.

3.1 %UnitTest Test Execution Methods

The default behavior when running unit tests is for the tests to be loaded into InterSystems IRIS, compiled, executed, and then deleted. This prevents test code from cluttering your InterSystems IRIS namespace. To deviate from this default behavior, you can either use **DebugRunTestCase()** or you can add flags to the *qualifiers* argument of either of these methods. For example, you may want to develop your test cases locally, within your namespace, without having to reload them every time you make a change. In that case, you could pass the `/nodelete` flag as part of the *qualifiers* argument.

RunTest (“testSpec”, “qualifiers”, “userparam”)

Executes a test or set of tests within the directory specified in the global ^*UnitTestRoot*. Once tests are executed, deletes from InterSystems IRIS all loaded tests and test classes.

```
USER>Do ##class(%UnitTest.Manager).RunTest("MyTests")
```

See RunTest() in the class reference for a detailed description of how to use this method and its arguments.

DebugRunTestCase (“testSpec”, “qualifiers”, “userparam”)

Executes a test or set of tests without loading or deleting any test classes.

```
USER>Do ##class(%UnitTest.Manager).DebugRunTestCase("MyTests", "/display=none/debug", "/log")
```

See DebugRunTestCase() in the class reference for more information.

Note: Occasionally, one of your unit tests may change the value of a SQL Configuration Option (such as `AutoParallel` or `AutoParallelThreshold`, for example) or may leave behind a lock or an open transaction or similar, triggering an error notification that states the problem. The unit test manager, %UnitTest.Manager, automatically resets the value of the changed SQL Configuration Option, deletes the leftover lock, or closes the transaction before executing the next unit test.

4

Viewing %UnitTest Results

You can view the results of your tests in any of the following ways:

- In the console — Basic test results are printed to the console output.
- In the %UnitTest.Result.TestAssert table — Test results are stored in tabular form in *^UnitTest.Result*, and they can be accessed via the %UnitTest.Result.TestAssert table.
- In the Management Portal — Running a unit test generates a test report that comprises a series of web pages. Test reports are organized by namespace, and they can be viewed in the Management Portal, in the UnitTest Portal area. See Viewing %UnitTest Reports in the Management Portal for details.

4.1 Viewing %UnitTest Results Programmatically

Test asserts, including test results, are logged in the %UnitTest.Result.TestAssert table for structured access to the data. The table includes the following fields:

`Status`

The success of failure value of the test assert. Possible values are as follows:

Table 4–1:

Logical Value	*Meaning*
0	failed
1	passed
2	skipped

`Action`

The name of the `$$$AssertX` macro used to perform the test. Note that the leading `$$$` are not included in the table.

`Description`

The value of the `test_description` argument you passed to the `$$$AssertX` macro. If `test_description` was not passed, this field defaults to the string representation of the first argument to the `$$$AssertX` macro. See Macros of the %UnitTest.TestCase Class for details about `$$$AssertX` macro arguments.

`Location`

The location in the test class from which the test assert originates, in `label[+offset]^[|"ns"|]doc.ext` format.

4.1.1 Troubleshooting Test Assert Locations

Under certain circumstances, it is possible the test assert locations may not be properly mapped back to the classes. For example, if all of the locations are in generated `INT` routines. In such cases, you should run your tests with the `/keepsource` and `/generatemap` qualifiers in the `qualifiers` argument to RunTest(). This enables the test manager to resolve the routine locations back to the source classes.

4.2 Viewing %UnitTest Reports in the Management Portal

Executing tests generates a hierarchical report, available in the Management Portal, containing results related to all tests executed.

If the report indicates a test has passed, that means the relevant *$$$AssertX* macro returned true: Your test produced the expected result. Test failure indicates the macro returned false: Your test did not produce the expected result, and you may need to debug the method being tested.

Follow these steps to view the report in the Management Portal:

1. Grant access for the %UnitTest classes to access the UnitTest Portal in the USER namespace:

   ```
   USER>set $namespace = "%SYS"
   %SYS>set ^SYS("Security", "CSP", "AllowPrefix", "/csp/user/", "%UnitTest.")=1
   ```

   Note: This step must be executed once, for security reasons, or you will not be able to navigate to **Unit Test Portal** in the Management Portal.

2. In the Management Portal, navigate to **System Explorer > Tools > UnitTest Portal**, and switch back to the USER namespace.
3. To launch **UnitTest Portal** and view your test report, click **Go**. Your report displays.
4. Drill down in the report by following the links in the report to find increasingly specific information.
   - The first page provides a summary for all test suites.
   - The second page displays results by each test suite.
   - The third page displays the results by each test case.
   - The fourth page displays the results broken out by test method.
   - The final page displays results for each `$$$AssertX` macro used in a test method.