[**C3 Test Driven Development (TDD)**](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350)

[Skip to end of metadata](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#page-metadata-end)

* Created by [Nicholas Jaskolowski](javascript:;) , last modified on [Dec 07, 2017](https://wiki.web.att.com/pages/diffpagesbyversion.action?pageId=625541350&selectedPageVersions=5&selectedPageVersions=6)

[Go to start of metadata](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#page-metadata-start)

* [Perl](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-Perl)
  + [File Locations and Filenames](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-FileLocationsandFilenames)
  + [Test Modules](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-TestModules)
  + [Running Tests](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-RunningTests)
  + [Unit Tests](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-UnitTests)
    - [Setting up Mocks](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-SettingupMocks)
    - [Subtests](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-Subtests)
  + [Integration Tests](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-IntegrationTests)
* [Javascript](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-Javascript)
  + [File Locations and Filenames](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-FileLocationsandFilenames.1)
  + [Test Modules](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-TestModules.1)
* [Python](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-Python)
  + [File Locations and Filenames](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-FileLocationsandFilenames.2)
  + [Test Modules](https://wiki.web.att.com/pages/viewpage.action?pageId=625541350#C3TestDrivenDevelopment(TDD)-TestModules.2)

This page assumes you have a basic understanding of how test driven development works, and shows more of the style and guidelines for writing tests for C3.

If you're not familiar with TDD, [here is the BizOps page to get started](https://wiki.web.att.com/pages/viewpage.action?pageId=549290743).

**Perl**

**File Locations and Filenames**

When writing perl tests, the actual test files go in the /tests/ats/tests folder in the C3 repository. Each module or script should have it's own test file. Often times, the test file should already exist, but if it doesn't, filenames should follow the format of "test\_" + Name of file being tested + "Integration" (if applicable) + ".t". Ex. test\_myModule.t, test\_myModuleIntegration.t.

Be sure to add the test file to the c3.suite or c3\_integration.suite.

**Test Modules**

There a several perl modules used that are built for testing. I suggest you read up on the documentation for Test::Most and it's included modules such as Test::Deep and also DBD::Mock.

There are also a few custom modules built within Communicate to making testing easier. These can be found in /modules/C3/EET. The first is Test.pm which should be included in any test files, not production code. In there are helper subs to create database mocks, run the assert call, and do automated input checking.

**Test Helper Examples**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #creating mock database handles  my $dbh = EET::Test::createMockMySQLDatabase();      #automated tests for verifyInputPlus  # test returns error if missing input  {       EET::Test::inputCheckPlus({          VALIDATION\_PARAMS => [              {KEY => 'customerId', REGEX => qr/^\d+$/, VALID => 123, INVALID => ["ABC123"]},              {KEY => 'dbh'}          ],          TEST\_CALLBACK => sub{              return $myModule->getCustomerInfo(shift);          },          SUB\_NAME => 'getCustomerInfo'      });  } |

In the Test subfolder are different mock objects and Helper.pm. Helper.pm can currently be used in production code to verify input of required keys or values in hash passed into a sub.

**VerifyInputPlus Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | sub test{      my $args = shift;      # Arg check      my $inputCheck = verifyInputPlus({          VALIDATION\_PARAMS => [              {KEY => 'alarmDetail'},              {KEY => 'clientid'},              {KEY => 'serviceID', REGEX => qr/^\d+$/},              {KEY => 'referenceTicketNum', TYPE => "ARRAY"}          ],          INPUT  => $args,          SUB\_NAME => 'test'      });      if(!$inputCheck->{status}) {          return $inputCheck;      }      #rest of sub  } |

**Running Tests**

To run your tests, start your docker container and exec into it.

To run an individual test file, just run perl on the .t file like so: **perl /opt/tests/ats/test/test\_C3.t**

To run all of the tests in a suite file, use the command: **perl /opt/tests/ats/lib/dev\_test.pl /opt/tests/ats/c3.suite**

**Unit Tests**

**Setting up Mocks**

Mocks allow you to stub out calls that normally reach out to other modules or services and return expected results to control the flow of the code you're writing. Mocks should be declared at the top of the file to allow a global scope and the Test::MockModule is used to load in the module you want to overwrite.

**Mock File Setup**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | my $c3Mock;      BEGIN {      use lib (          "/opt/modules/C3",      );      use Test::MockModule;      $c3Mock = Test::MockModule->new('C3', no\_auto => 1);  } |

Once you set up the mock variable, you just mock out whatever sub you want before calling the sub you're testing.

**Mock Sub Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | $c3Mock->mock(      thatOneSub => sub{          return {              status => 1,              data => {                  key => "value"              }          };      }  );    #unmock after using at the end of the test  $c3Mock->unmock('thatOneSub')  #or  $c3Mock->unmock\_all(); |

**Mock Database Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | my $dbh = EET::Test::createMockMySQLDatabase();    $dbh->{mock\_add\_resultset} = {      sql => "SELECT hqsc, flash FROM cnoc\_wb\_events WHERE trackingticket = ?;",      results => [          ['hqsc', 'flash'],          ['ABC', '30566'],      ]  }; |

For the database mock above, you specify what sql statement the mock should match on. It doesn't actually run the sql, it's a string match so spacing/formatting is important in both the code and test. The results key is essentially a two dimensional array that represents the table your query would return. The first row is the column headers and the subsequent rows are the data that's returned.

You should read up more on the Test::MockModule documentation as well.

**Subtests**

Subtests should contain all of the unit tests for a particular sub. In other words, there is one subtest for each sub that you have in your perl module or script.

Unit tests within the subtest should be contained within blocks. This allows the unit tests to be run in isolation and minimize the risk of one test affecting another. The blocks should be preceded by a comment describing, with detail, what logical decision or output that unit test is proving. That same comment should also go into the assertion portion of the test to clearly show what test failed or passed when actually running the tests.

**Subtest Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | subtest "testSub" => sub{        #testSub returns an error if the database call fails      {          my $result = $c3->testSub(undef, "123");          assert($result, "DB error\n", "testSub returns an error if the database call fails");      }    }; |

When a larger return is expected back, use a separate variable to contain the expected result;

**Subtest Expected Result Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | subtest "testSub" => sub{        #testSub returns hash of information on success      {          my $result = $c3->testSub(undef, "123");          my $expectedResult = {              flash => "30566",              hqsc => "ABC",              key => "value"          };          assert($result, $expectedResult, "testSub returns hash of information on success");      }    }; |

And below is a more complete example with some mocks of modules and the database.

**Full Subtest Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | subtest "testSub" => sub{        my $expectedInput = "123";      #testSub returns an error if the database call fails      {          my $result = $c3->testSub(undef, $expectedInput);          assert($result, "DB error\n", "testSub returns an error if the database call fails");      }         #testSub returns hash of information on success      {          my $result = $c3->testSub(undef, $expectedInput);          my $expectedResult = {              flash => "30566",              hqsc => "ABC",              key => "value"          };          assert($result, $expectedResult, "testSub returns hash of information on success");      }        #testSub returns asset for ticket      {          my $dbh = EET::Test::createMockMySQLDatabase();          $aotsMock->mock(              aotsGetTicket => sub{                  return {                      STATUS => "OK",                      DATA => {                          AssetID => "TestAsset"                      }                  };              }          );            $dbh->{mock\_add\_resultset} = {              sql => "SELECT hqsc, flash FROM cnoc\_wb\_events WHERE trackingticket = ?;",              results => [                  ['hqsc', 'flash'],                  ['ABC', '30566'],              ]          };            my $result = $c3->testSub($dbh, $expectedInput);          my $expectedResult = {              flash => "30566",              hqsc => "ABC",              assetID => "TestAsset"          };          assert($result, $expectedResult, "testSub returns asset for ticket");          $aotsMock->unmock("aotsGetTicket");      }    }; |

**Integration Tests**

Integration tests follow the same format of each test should be wrapped in a subtest. Since the nature of an integration test can span multiple subs, the subtest should capture more of the use case it's trying to prove works. There should be an evaled setup and teardown phase for each test, which would include seeding the database, setting up connections to services and more.

**Javascript**

**File Locations and Filenames**

When writing javascript tests, the actual test files go in the /node/react folder in the C3 repository.

**Test Modules**

Mocha, chai, enzyme, sinon

**Python**

**File Locations and Filenames**

When writing python tests, the test files should go into the respective container's test folder and under the appropriate unit or integration test folder. Conftest.py and other files outside are a good place to put fixtures or other testing helpers. Filenames should begin with "test\_" in order to be picked up by pytest's auto-discovery, followed by the module name.

**Test Modules**

Pytest, mock