Unit Testing in SugarCRM

John Mertic

SUGARCRM.
THE CLOUD IS OPEN

What is Unit Testing

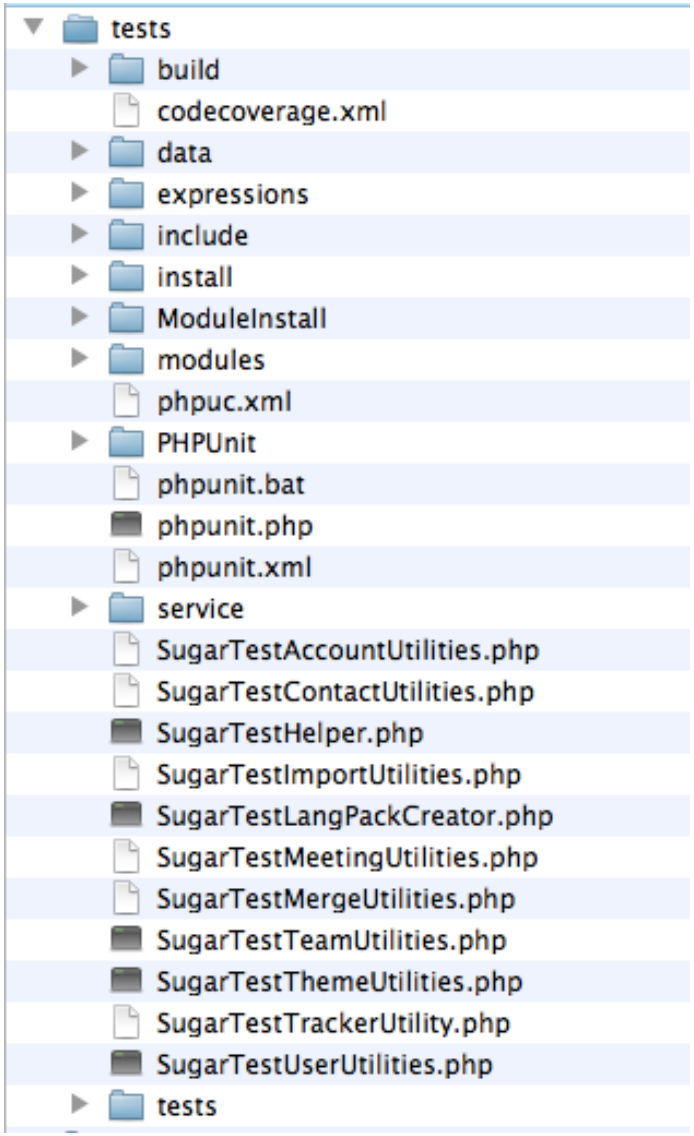
- Unit testing is a way to test individual pieces of source code for correctness.
- Best used for the following use cases:
  - Testing API functionality
  - Writing new code (Test Driven Development)
  - Verifying impact of code changes
- Not such a good idea for testing UI; use functional testing for that.

Types of Unit Tests

- Bug fix tests
  - Tests the successful fixing of a bug
    - Test should initially fail, since the test should be written to pass with the correct behavior, not the incorrect behavior
    - Test passing indicates successful fix
  - Each bug should have at least one unit test
    - Helps prevent future regressions
- Component level tests
  - Tests the basic functionality of a component
  - Should test both success and failure results, along with known edge cases
  - Should cover all code paths in a component (code coverage)

How we do Unit Testing

- We use PHPUnit 3.5 for unit testing
  - Read the manual at http://www.phpunit.de/manual/
- All unit tests are stored in the tests/ directory
  - PHPUnit/ contains PHPUnit source code
  - SugarTest*.php files are utilities to help with unit tests
  - Other directories match the file system layout for the application



How to run unit tests

- First, make sure the Sugar instance has went thru the installer
- Next, make sure you have full read/write permissions with the current test running user with all files in the instance
  - Most important is the cache/ directory

Running the entire unit test suite

```
$ cd tests/
$ ./phpunit.php
PHPUnit 3.5.0 by Sebastian Bergmann.
.....................SS.S.....SSSSSS.........................  60 / 818
--clipped--
.......................................

Time: 02:58, Memory: 207.00Mb

OK, but incomplete or skipped tests!
Tests: 818, Assertions: 1734, Skipped: 46

$
```

Running a single test file

```
$ cd tests/
$ ./phpunit.php include/JSONTest.php
PHPUnit 3.5.0 by Sebastian Bergmann.

..

Time: 1 second, Memory: 20.00Mb

OK (2 tests, 2 assertions)
```

Writing a unit test

- Naming convention
  - All test files need to have the name of <<test name>>Test.php
    - Module tests are stored under modules/<<module name>>/<<test name>>Test.php
      - No subdirectories underneath modules/<<module name>>
    - Tests of functionality in the include direct stored under include/<<dir name>>/<<test name>>Test.php
  - Tests for a specific bug should use Bug<<number>> for <<test name>>

What a simple unit test looks like

```
<?php
require_once 'include/JSON.php';

class JSONTest extends Sugar_PHPUnit_Framework_TestCase
{
    public function testCanEncodeBasicArray()
    {
        $array = array('foo' => 'bar', 'bar' => 'foo');
        $json = new JSON();
        $this->assertEquals(
            '{"foo":"bar","bar":"foo"}',
            $json->encode($array)
        );
    }

    public function testCanEncodeBasicObjects()
    {
        $obj = new stdClass();
        $obj->foo = 'bar';
        $obj->bar = 'foo';
        $json = new JSON();
        $this->assertEquals(
            '{"foo":"bar","bar":"foo"}',
            $json->encode($obj)
        );
    }
}
```

What a Bug related unit test looks like

```
<?php
require_once 'include/TimeDate.php';
require_once 'modules/Calendar/Calendar.php';
require_once 'modules/Meetings/Meeting.php';

/**
 * @ticket 20626
 */
class Bug20626Test extends Sugar_PHPUnit_Framework_TestCase
{
    public function setUp()
    {
          $GLOBALS['reload_vardefs'] = true;
        global $current_user;

        $current_user = SugarTestUserUtilities::createAnonymousUser();
          }

    public function tearDown()
    {
        SugarTestUserUtilities::removeAllCreatedAnonymousUsers();
        unset($GLOBALS['current_user']);
        $GLOBALS['reload_vardefs'] = false;
    }

--continued--
```

What a Bug related unit test looks like (cont)

```
--continued—

    public function testDateAndTimeShownInCalendarActivityAdditionalDetailsPopup()
    {
        global $timedate,$sugar_config,$DO_USER_TIME_OFFSET , $current_user;

        $DO_USER_TIME_OFFSET = true;
        $timedate = new TimeDate();

        $meeting = new Meeting();
        $format = $current_user->getUserDateTimePreferences();
        $meeting->date_start = $timedate->swap_formats("2006-12-23 11:00pm" , 'Y-m-d
h:ia', $format['date'].' '.$format['time']);
        $meeting->time_start = "";
        $meeting->object_name = "Meeting";
        $meeting->duration_hours = 2;
        $ca = new CalendarActivity($meeting);
        $this->assertEquals($meeting->date_start , $ca->sugar_bean->date_start);
    }
}
```

Tools to making testing easier

- SugarTestUserUtilities
  - Use for creating test users
  - Methods (all called statically)
    - SugarTestUserUtilities::createAnonymousUser() creates the user, returns the User object.
    - SugarTestUserUtilities::removeAllCreatedAnonymousUsers() removes all the users that have been created.
      - Should be called in tearDown() method.
    - SugarTestUserUtilities:: getCreatedUserIds() returns a list of created user_ids

Tools to making testing easier (cont.)

- SugarTestTeamUtilities
  - Use for creating test teams.
  - Methods (all called statically)
    - SugarTestTeamUtilities::createAnonymousTeam() creates the user, returns the Team object.
    - SugarTestTeamUtilities::removeAllCreatedAnonymousTeams() removes all the teams that have been created.
      - Should be called in tearDown() method.
    - SugarTestTeamUtilities:: getCreatedTeamIds() returns a list of created teams_ids

Ways to make testing easier

- Providers
  - Can be used to provide a dataset to a testing method
  - Makes it easy to test several input combinations to a method or function.

Provider example

```
<?php
require_once 'include/SugarEmailAddress/SugarEmailAddress.php';

class SugarEmailAddressRegexTest extends Sugar_PHPUnit_Framework_TestCase
{
    public function providerEmailAddressRegex()
    {
                return array(
                    array('john@john.com',true),
                    array('----!john.com',false),
                    // For Bug 13765
                    array('st.-annen-stift@t-online.de',true),
                    // For Bug 39186
                    array('qfflats-@uol.com.br',true),
                    array('atendimento-hd.@uol.com.br',true),
                    );
            }

    /**
     * @ticket 13765
     * @ticket 39186
     * @dataProvider providerEmailAddressRegex
     */
    public function testEmailAddressRegex($email, $valid)
    {
        $sea = new SugarEmailAddress;

        if ( $valid ) {
            $this->assertRegExp($sea->regex,$email);
        }
        else {
            $this->assertNotRegExp($sea->regex,$email);
        }
    }
}
```

Ways to make testing easier

- Mock Objects
  - Can be used in cases where setting up underlying components is difficult or time consuming.
    - Examples: Database Connections, etc.
  - Calls on the object can give back predetermined values, but still have the same API.

Mock Object example

```
<?php
require_once("modules/Accounts/Account.php");

/**
 * ticket 24095
 */
class Bug24095Test extends Sugar_PHPUnit_Framework_TestCase
{
    --clipped code section—

    public function testDynamicFieldsRetrieveWorks()
    {
        $bean = $this->getMock('Account' , array('hasCustomFields'));
        $bean->custom_fields = new DynamicField($bean->module_dir);
        $bean->custom_fields->setup($bean);
        $bean->expects($this->any())
             ->method('hasCustomFields')
             ->will($this->returnValue(true));
        $bean->table_name = $this->_tablename;
        $bean->id = '12345';
        $bean->custom_fields->retrieve();
        $this->assertEquals($bean->id_c, '12345');
        $this->assertEquals($bean->foo_c, '67890');
    }
}
```

Another Mock Object example

```
<?php
$k_path_url = 'http://localhost/';
require_once('include/tcpdf/config/lang/eng.php');
require_once('include/tcpdf/tcpdf.php');
/**
 * @ticket 38850
 */
class Bug38850Test extends Sugar_PHPUnit_Framework_TestCase
{
    public function testCanInterjectCodeInTcpdfTag()
    {
        $pdf = new Bug38850TestMock(PDF_PAGE_ORIENTATION, PDF_UNIT, PDF_PAGE_FORMAT,
true, 'UTF-8', false);

        --clipped--

        $this->assertNotContains('Can Interject Code',$output);
    }
}

class Bug38850TestMock extends TCPDF
{
    public function openHTMLTagHandler(&$dom, $key, $cell=false)
    {
        parent::openHTMLTagHandler($dom, $key, $cell);
    }
}
```

Pointers for effective unit testing

- Use setUp() and tearDown() to effectively setup your environment and clean it up
  - Be sure to reset any $GLOBALS back to their original values
  - Remove any data that has been added to the database
    - Also, don't destroy existing data
  - Be cognizant of how the various $sugar_config settings affect your tests
- Setup only what you need to make the test work
  - Don't worry about specifying settings or variables that don't matter

Pointers for effective unit testing (cont.)

- Don’t use assertTrue() or assertFalse() when assertEquals() or one of the other assert*() methods is more appropriate.
  - Examples:
    - assertTrue($a == $b) should be assertEquals($a,$b)
    - assertTrue(strpos($str,$substr)) should be assertRegExp (“/$substr/”,$str)
  - Other assert*() methods give back more descriptive error messages based on the inputs
- All of the assert*() methods can have a description as the last parameter. Use this when the failing result needs explanation
  - Especially true for assertTrue() and assertFalse()

Pointers for effective unit testing (cont. 2)

- Be careful with too many assert*() calls in a test
  - It can help isolate failures in a test much more easily
  - Don’t however use as a way to have a test test too much
- Use markTestSkipped() method to have a test suite skipped if needed.
  - Example: Test only applies to a certain database
- Testing code should be pragmatic
  - Unit tests are not meant to be works of art ;-)
  - Code reuse is not as important
  - Focus more on readability for later debugging purposes and cleanly testing the code.

Writing Testable Code

- Code should be designed to be tested easily.
- Things to avoid
  - ‘static’ variables in functions and methods that cannot be altered or reset outside of the function or method.
  - Global singleton objects that don’t have any reset capability
  - Using globals (especially superglobals) inside methods
  - Altering global state unnecessarily inside methods
  - Having methods/functions that directly create new class objects inside of them
    - Especially a problem with constructors
  - Having methods/functions that are really long or do too many different things
  - Having classes that do too many different things.

Example of code with testability problems

```
/**
 * Returns the bean object of the given module
 *
 * @return object
 */
public function loadBean()
{
    global $beanList, $beanFiles;

    if ( !isset($beanList) || !isset($beanFiles) )
        require('include/modules.php');

    if ( isset($beanList[$this->_moduleName]) ) {
        $bean = $beanList[$this->_moduleName];
        if (isset($beanFiles[$bean])) {
            require_once($beanFiles[$bean]);
            $focus = new $bean;
        }
        else
            return false;
    }
    else
        return false;

    return $focus;
}
```

How to fix the testability problems

```
/**
 * Returns the bean object of the given module
 *
 * @return object
 */
public function loadBean($beanList = null, $beanFiles = null, $returnObject = true)
{
    // Populate these reference arrays
    if ( empty($beanList) ) {
        global $beanList;
    }
    if ( empty($beanFiles) ) {
        global $beanFiles;
    }
    if ( !isset($beanList) || !isset($beanFiles) ) {
        require('include/modules.php');
    }

    if ( isset($beanList[$this->_moduleName]) ) {
        $bean = $beanList[$this->_moduleName];
        if (isset($beanFiles[$bean])) {
            if ( !$returnObject ) {
                return true;
            }
            require_once($beanFiles[$bean]);
            $focus = new $bean;
        }
        else {
            return false;
        }
    }
    else {
        return false;
    }

    return $focus;
}
```

Q&A

- Questions?