Testing

Overview

There's a lot of information and articles out there about why and how you should be writing your tests. But, in short, it ensures your code does what you want it to and makes sure it continues to even after you or others make modifications.

Not everything is easy to test. But, at the very least, libraries and other shared pieces of code would ideally be covered by unit tests. It takes time, but it's definitely easier to do it as you write code than it is to go back and add all your tests at the end of the project.

In an effort to make testing a little more accessible, HUBzero offers some basic guidance and structure for your testing. We'll step through what's available below.

Location

Your tests will live in a tests folder within the applicable extension directory. For example, tests for the blog component can be found at:

```
core/components/com_blog/tests/...
```

Note: Tests are only supported in individual extensions with 2.1.10+. Older versions only support running tests from the HUBzero framework.

Test naming convention and structure should follow the definitions found in the PHPUnit documentation.

Test Types

There are two primary types of tests, basic and database. Basic tests involve no external resources, whereas database tests require the ability to simulate/mock database calls.

Basic Tests

Basic tests in HUBzero offer no additional functionality or abstraction over the PHPUnit_Framework_TestCase. Therefore there's really nothing that needs to be covered here that isn't already in the PHPUnit documentation.

Database Tests

To help with writing tests that require a database object, we've worked to provide some shortcuts and best practices. Database tests are tough because they can be slow, and you don't want them to depend a certain database state or mess up another developers database. So, to get around this, we either completely mock the database, or use a reloadable sqlite database.

Let's look at an example of this.

```
/**
 * Test to make sure we can run a basic select statement
 *
 * @return void
 **/
public function testBasicFetch()
{
    $dbo   = $this->getMockDriver();
    $query = new Query($dbo);

    // Try to actually fetch some rows
    $rows = $query->select('*')
                  ->from('users')
                  ->whereEquals('id', '1')
                  ->fetch();

    // Basically, as long as we don't get false here, we're good
    $this->assertCount(1, $rows, 'Query should have returned one resul
t');
}
```

You'll see above the call to a function named getMockDriver(). This method is going to give you back a database object, loading up a sqlite database named test.sqlite3, by default. This is a fully functioning database driver.

To make the database driver useful, you'll need at least two files included in your tests directory. They are:

```
Tests/Fixtures/seed.xml
Tests/Fixtures/test.sqlite3
```

The seed.xml file will contain all your sample data. This will be automatically loaded in the test framework for each test class (i.e. file). The structure and destination of all database operations

will come from the test.sqlite3 file.

The names of those files can also be changed by overwriting the $fixture and/or $seed properties on your test class.

Scaffolding and Running Tests

To get starting writing new tests, you can use the muse scaffolding command to create a test stub. This will look something like this:

```
me@me.org:~# muse scaffolding create test lib_database --type=database
Creating /var/www/example/core/libraries/Hubzero/database/Tests/Exampl
eDatabaseTest.php
```

The test scaffolding expects the first argument after test to be the extension into which the test should be placed. A --type argument can also be given to specify whether or not you're creating a basic or a database test. In this example, given that we're testing the database object, the database test type obviously makes sense.

Running Tests

Once you've created your tests, you'll need to run them. To get started, you can use the muse test command.

First, you'll probably want to list all available tests that can be run. From the console, run muse test show. This will list all the available tests.

```
me@me.org:~# muse test show
lib_base
lib_browser
lib_cache
lib_config
lib_console
lib_database
lib_debug
lib_notification
lib_pathway
lib_spam
lib_template
lib_utility
```

```
core:com_blog
```

Available tests are grouped by their respective extension or library. Components (com_), modules (mod_), plugins (plg_), and templates (tpl_) will also be prefixed with an indicator as to if the extension is in the /core or /app directory. This is because it is possible to have an extension with the same name in both /app and /core and allows for specifying to muse the exact suite of tests to be run.

Next, you can run tests with the run command:

```
me@me.org:~# muse test run lib_database
PHPUnit 4.6.2 by Sebastian Bergmann and contributors.

..................................................

Time: 2.51 seconds, Memory: 17.5Mb

OK (51 tests, 73 assertions)
```

More details on the muse functionality can be found in the Muse documentation.