Unit Testing

Bullet-Proof Code

Robust applications are the result of comprehensive testing. Verifying that each part of your program conforms to the specifications and lives up to the expectations of the end-user means finding bugs and fixing them as early as possible in the application development cycle.

If you know little or nothing about unit testing methodologies, you're probably embedding pieces of code directly in your existing program to help you with debugging. That of course means you have to remove them once the program is running. Leftover code fragments, poor design and faulty implementation can creep up as bugs when you roll out your application later.

F3 makes it easy for you to debug programs - without getting in the way of your regular thought processes. The framework does not require you to build complex OOP classes, heavy test structures, and obtrusive procedures.

A unit (or test fixture) can be a function/method or a class. Let's have a simple example:

```
function hello() {
    return 'Hello, World';
}
```

Save it in a file called `hello.php`. Now how do we know it really runs as expected? Let's create our test procedure:

```
$f3=require(__DIR__.'/lib/base.php'); // path to f3

// Set up
$test=new Test;
include('hello.php');

// This is where the tests begin
$test->expect(
    is_callable('hello'),
    'hello() is a function'
);

// Another test
$hello=hello();
$test->expect(
    !empty($hello),
    'Something was returned'
);

// This test should succeed
$test->expect(
    is_string($hello),
    'Return value is a string'
);

// This test is bound to fail
$test->expect(
    strlen($hello)==13,
    'String length is 13'
);

// Display the results; not MVC but let's keep it simple
foreach ($test->results() as $result) {
    echo $result['text'].'<br>';
    if ($result['status'])
        echo 'Pass';
    else
        echo 'Fail ('.$result['source'].')';
    echo '<br>';
}
```

Save it in a file called `test.php`. This way we can preserve the integrity of `hello.php`.

Now here's the meat of our unit testing process.

For each test you want to run, call the `expect()` method of the `Test` class. `expect()` has 2 arguments:

1. `test` is the test to be run. This needs to be a simple test that will result in a Pass or Fail condition. ex: `$i==1`
2. `text` is the text to be displayed for this test, when test results are displayed. ex: "input equals numeric 1".

When the tests are run, F3's built-in `Test` class will keeps track of the result of each `expect()` method call. The output of each `$test->expect()` call is saved in a multi-dimensional array named `results` with the keys:

- `text` (mirroring argument 2 of `expect()`),

- `status` (boolean representing the result of a test),
- `source` (file name/line number of the specific test to aid in debugging)

At the end of your test, you can iterate through this multidim array, print the text of the test, and then print the status of the test (`true` if the test passed, `false` if the test failed, and include the source file and line for debugging)

Fat-Free gives you the freedom to display test results in any way you want. You can have the output in plain text or even a nice-looking HTML template, by rendering a template file that interates through the $test->results() array.

So how do we run our unit test? If you saved `test.php` in the document root folder, you can just open your browser and specify the address `http://localhost/test.php` . That's all there is to it.

Mocking HTTP Requests

F3 gives you the ability to simulate HTTP requests from within your PHP program so you can test the behavior of a particular route, just as if a website visitor requested that page through a browser.

Here's a simple mock request:

```
$f3->set('QUIET',TRUE);  // do not show output of the active route
$f3->mock('GET /test');  // set the route that f3 will run
// run tests using expect() as shown above
// ...
$f3->set('QUIET',FALSE); // allow test results to be shown later
$f3->clear('ERROR');  // clear any errors
```

`Tip:` If you have a route defined with token parameters, i.e. /test/@name, you can test that route by setting a value for the token in the mock command, and access that value during testing from F3's PARAMS assoc array

```
$f3->mock('GET /test/steve');
$name = $f3->get('PARAMS["name"]');
$test->expect(
    $name =="steve",
    'Uri param "name" equals "steve"'
);
```

To mock a POST request and submit a simulated HTML form:

```
$f3->mock('POST /test', array('foo'=>'bar')); // pass in form values using assoc arra
y
```

`Tip:` When using mock, or displaying test results using a rendered template, or testing something in your database, you need to include config settings for F3 so it knows the location of your templates, db parameters, etc.

Expecting the Worst that can Happen

Once you get the hang of testing the smallest units of your application, you can then move on to the bigger components, modules, and subsystems - checking along the way if the parts are correctly communicating with each other.

`Tip:` If you create separate test files, and your tests need access to config values or db access, use an include once directive to a file that has the path to f3 and the config values, and call that include file from each separate test file. If you include each test file into a single test batch file, the "include once" will make it easy to test individual classes and your complete application.

Testing manageable chunks of code leads to more reliable programs that work as you expect, and weaves the testing process into the fabric of your development cycle.

The question to ask yourself is: "Have I tested all possible scenarios?" More often than not, those situations that have not been taken into consideration are the likely causes of bugs.

Unit testing helps a lot in minimizing these occurrences. Even a few tests on each fixture can greatly reduce headaches. On the other hand, writing applications without unit testing at all invites trouble.

← 7. Optimization (optimization)

9. Quick Reference → (quick-reference)