**Test Doubles**

As we test things, sometimes we want to get things out of the way in our code. This is where **test doubles** come into play. The general idea is to remove any variables in our code and give ourselves test versions to replace actual systems. Always returning known values and ensuring systems behave a certain way. When dealing with test doubles, a popular library for creating test doubles is **Mockery**.

public function test\_handles\_empty\_order\_list() {

    $api = Mockery::mock( Api::class )->makePartial();

    $api->shouldReceive( 'get\_all\_orders' )

        ->once()

        ->andReturn( [] );

    $this->assertEmpty( $api->get\_recent\_orders() );

}

There’s also the PHPUnit Markup assertions, powered by DOMDocument. Lets use DOMDocuments to make a DOM query.

function test\_button\_contains\_active\_state() {

    $output = some\_function();

    $this->assertContainsSelector('.button.active', $output);

}

**WP Core Test Suite**

This is what WP core itself uses to ensure all the PHP in WP is behaving the way we expect it to. If we want to use the core test suite, you can run $ wp scaffold plugin-tests my-plugin to generate test scaffolding via WP-CLI. Get the test suite out of the box.

We want to make sure certain things happen before every test method. You don’t have to write it every time, only once.

We have the concept of **groups**where we run tests of a similar nature across suites and classes. I can just run the following code.

/\*\*

 \* @group Posts

 \* @group PostMeta

 \*/

public function test\_includes\_private\_posts()

{

    // ...

}

$ phpunit --group=Posts

This comes in handy when you have a large test suite and want to make sure related things aren’t going to break.

**Data Providers**

Often in our testing you can have the same test but different data. For this, we have a nice tool called data providers. You can run through them without having to paste the same method over and over again. So we specify a data provider for it. If you’re working with simple data types like strings and integers. You can choose to define just one method for example:

/\*\*

 \* @dataProvider my\_data\_provider()

 \*/

public function test\_my\_function( $expected, $value ) {

    $this->assertEquals( $expected, my\_function( $value ) );

}

public function my\_data\_provider() {

    return [

        'Description of case 1' => ['foo', 'bar'],

        'Description of case 2' => ['bar', 'baz'],

    ];

}

/\*\*

 \* @testWith ["foo", "bar"]

 \*           ["bar", "baz"]

 \*/

public function test\_my\_function( $expected, $value ) {

    $this->assertEquals( $expected, my\_function( $value ) );

}

You can even generate **dummy data with factories tests**. You can generate users, posts and more – for testing purposes.

// Create the post and retrieve its ID.

$post\_id = $this->factory->post->create();

// Create and retrieve the new post.

$post = $this->factory->post->create\_and\_get();

// Override default parameters.

$post = $this->factory->post->create\_and\_get( [

    'post\_title'  => 'My Test Post',

    'post\_author' => $author\_id,

] );

// Create multiple instances.

$posts = $this->factory->post->create\_many( 5, [

    'post\_author' => $author\_id,

] );

**Checking for WP\_ERRORS**

Was the response an instance of WP\_Error? Coming back to the search for truth – Is truth a WP\_Error? As we write our code, there’s a pattern for how this should be arranged to set up the scenario.

public function test\_function\_can\_return\_wp\_error() {

    $response = myplugin\_function();

    $this->assertWPError($response);

}

Next we execute the code, and finally we make assertions around it – in other words, verify that things happened as you expected.

**Testing Permissions**

public function test\_non\_admins\_cannot\_clear\_cache() {

    // Arrange

    $user\_id = $this->factory->user->create( [

        'role' => 'author',

    ] );

    wp\_set\_current\_user( $user\_id );

    // Act

    $response = myplugin\_clear\_cache();

    // Assert

    $this->assertWPError($response);

    $this->assertSame(403, $response->get\_error\_code());

}

**Registering a custom post type**

public function test\_book\_cpt\_is\_registered() {

    myplugin\_register\_post\_types();

    $post\_type = get\_post\_type\_object( 'book' );

    // Verify the post type is registered along with key properties.

    $this->assertNotNull( $post\_type );

    $this->assertTrue( $post\_type->public );

    $this->assertFalse( $post\_type->hierarchical );

}

**Testing Hooks**

public function test\_function\_does\_action() {

    myplugin\_function();

    $this->assertSame( 1, did\_action( 'myplugin\_action' ) );

}

public function test\_function\_does\_action() {

    $called = false;

    // Register a callback to validate arguments.

    add\_action( 'myplugin\_action', function () use (&$called) {

        // Only return true if validations passed.

        $called = true;

    } );

    myplugin\_function();

    $this->assertTrue( $called );

}

**Testing Output**

public function test\_shortcode\_output() {

    ob\_start();

    do\_shortcode( '[recent-posts title="Latest Posts"]' );

    $output = ob\_get\_clean();

    $this->assertContains( '<h2>Latest Posts</h2>', $output );

}

public function test\_shortcode\_output() {

    $this->expectOutput( '<h2>Latest Posts</h2>' );

    do\_shortcode( '[recent-posts title="Latest Posts"]' );

}

**Stubbing HTTP Requests**

add\_filter( 'pre\_http\_request', function () {

    return [

        'headers'  => [],

        'body'     => '',

        'response' => [

            'code'    => 200,

            'message' => 'OK',

        ],

        'cookies'  => [],

        'filename' => '',

    ];

} );

**Basic Automated Testing Workflow**

Steve explains the basic idea behind TDD – test driven development.

1. Write a (failing) test to describe the functionality/behavior. You’re describing how it should work. This can be called ‘red’ – there is a broken code.
2. Write the code necessary to make the test pass. All we have to do is get the test to pass. This can be known as green – the code that works.
3. Refactor, rinse, & repeat. Now we can go back and refine the code.