# Introduction to Tests

There are three types of tests which are used in this application:

* Unit testing - Tests small pieces of independent code
* Feature/functional testing - Tests end-to-end features and interactions
* Browser testing - Tests what happens in the browser

Another type of test, which isn’t used in this application, is integration testing, which tests how the system interacts with others.

In this Laravel+Vue application, unit testing and feature testing are written using PHPUnit, a unit testing framework for PHP, and browser testing is written using Laravel Dusk, a browser automation testing API.

Laravel Dusk lets developers see output from the JavaScript console, navigate to a webpage, execute JavaScript, take screenshots, inspect elements, interact with elements, generate elements, scroll the page, and more.

Tests may require creating fake database entries for testing, which will get deleted once testing ends, as it is not recommended to use real data. This is done using the Illuminate RefreshDatabase and DatabaseMigrations classes to re-migrate the database for each test, and the Illuminate Eloquent Factory class to generate fake data.

Since unit tests test small pieces of code, they tend to be the fastest type of test, while browser tests tend to be the slowest since they emulate a real browser that refreshes webpages and waits for elements to display.

# Unit Tests

A unit test is a piece of code that is written to test units, small pieces of code that can be isolated logically, like a function or class method which is independent of everything else.

The point of unit testing is to make sure the individual pieces work as expected before putting them together (integration).

To create a unit test, we should know the output of the function for a given input. For example, to test a multiplication function, we input two numbers, and , and check if the result is correctly calculated as :

|  |
| --- |
| class UnitTest extends TestCase {      public function test\_multiply() {          $result = multiply(6, 2);          $this->assertEquals($result, 12);      }  } |

Units tests should be well-thought about: They should contain edge cases, error handling, and properties of the function. For example, a multiplication function should be tested with two positive numbers, one positive and one negative numbers, two negative numbers; but also multiplication by zero should always result in a zero, and the commutative property should be verified as well:

|  |
| --- |
| public function test\_multiply() {      $this->assertEquals(multiply(6, 2), 12);             // Two positives      $this->assertEquals(multiply(-1, 2), -2);            // One negative      $this->assertEquals(multiply(-2, -10), 20);          // Two negatives      $this->assertEquals(multiply(0, 2), 0);              // Zero      $this->assertEquals(multiply(3, 5), multiply(5, 3)); // Commutative      $this->expectException(multiply('a', 5), BadArgumentException::class); // Error  } |

An example of unit test relevant to the project is checking the relationship between products and categories. In their Model classes, they are defined in a way that a category has many products, and a product belongs to a category pointed by the category\_id property/column.

To test that, a new product is created by its factory, which generates a product having a category that exists. The test then checks if the product’s category is an instance of the Category class:

|  |
| --- |
| public function test\_product\_belongs\_to\_category() {      $product = Product::factory()->make();      $this->assertInstanceOf(Category::class, $product->category);  } |

It is also possible to test if API endpoints show the correct product given its ID in the URL. No fake products have been created for this test, which is testing the API with products of IDs 1, 2, and 3.

|  |
| --- |
| public function test\_get\_product\_id() {      for ($i = 1; $i <= 3; $i++) {          $url = 'http://127.0.0.1:8000/api/products/'.$i;          $product = json\_decode(file\_get\_contents($url));          $this->assertEquals($product->id, $i);      }  } |

# Feature Tests

The difference between unit and feature tests is that unit tests are written to test small, independent pieces of code, while feature tests are written to test the interactions between methods or classes as users are expecting the application to.

Feature tests are about what the user will see or do in the final application.

An example of feature test is making sure that a certain route/endpoint/webpage exists and is running by checking if it responds with HTTP status code 200 OK:

|  |
| --- |
| class FeatureTest extends TestCase {  public function test\_products\_page\_exists() {       $response = $this->get('/products');       $response->assertStatus(200);  }  } |

The test above check only that the page exists. A test should also be written to check if a product’s page not only exists, but contains the product as well:

|  |
| --- |
| public function test\_product\_page\_has\_product\_name() {      $product = Product::factory()->create();  $response = $this->get('/product/'.$product->id);      $response->assertStatus(200);     $response->assertSee($product->name);  } |

In the test above, a new product is temporarily created in the database using a product factory. The tests checks that the webpage exists and is loaded, and that displays the product’s name.

If the application has an API for, for example, displaying information about a product in JSON format, a test can be written to check if that format has the correct properties/keys:

|  |
| --- |
| public function test\_product\_structure() {      $product = Product::factory()->create();      $response = $this->get('/api/products/'.$product->id);      $response->assertStatus(200);      $response->assertJsonStructure([          'id', 'name', 'description', 'price', 'category\_id', 'subcategory'      ]);  } |

# Browser Tests

Browser tests are tests that require to run in the browser, e.g., functional tests that need JavaScript. The functional tests above only work for pages in which the elements are static (Laravel), but would not work for pages in which elements change or are generated after opening the page (Vue).

As for functional tests, browser tests can assert than a certain text is present on the page, but are flexible and precise, as it is possible to look for a particular HTML element or selector, and more:

|  |
| --- |
| class BrowserTest extends DuskTestCase {      public function test\_laravel\_single\_product\_page() {          $this->browse(function (Browser $bro) {              $product = Product::factory()->create();              $bro->visit('/product/'.$product->id)                  ->screenshot('test\_laravel\_single\_product\_page')                  ->assertSeeLink('Back')                  ->assertSee('Category')                  ->assertSee('Product Code: '.$product->code);          });      }  } |

The test above creates a fake product, then visits the URL showcasing this product, which represents a webpage built with Laravel. The test then takes and saves a screenshot, then checks for any link (<a> element) with the text “Back”, as well as an element with the text “Category”, and another with the product code.

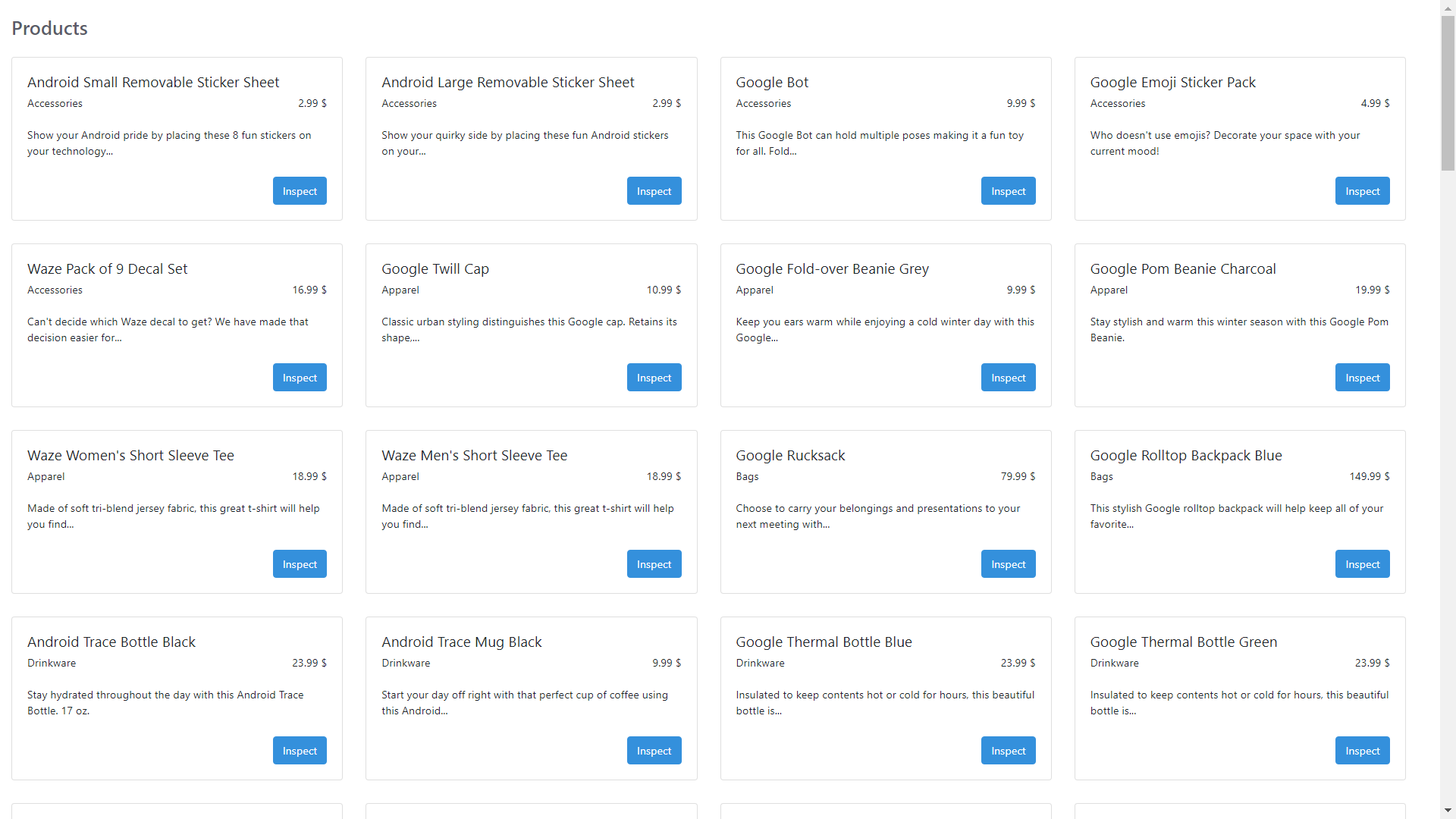


After this, since the webpage displays as expected, the “Back” link should be tested to check that it works and redirects to the page showcasing all the products. Since the products on that page are dynamically fetched from the database to create its elements, the test should be paused for a bit after clicking on the link.

The new webpage is built in Vue, which, by default, should show a grid of products cards with the CSS ID #products-grid. This webpage sometimes has an element with the CSS ID #product-single that should not be present until after clicking on a product.

This webpage contains a header with the text “Products”, and should show, among others, a product card having the fake product’s name:

|  |
| --- |
| public function test\_laravel\_single\_product\_page\_back\_button() {      $this->browse(function (Browser $bro) {          $product = Product::factory()->create();          $bro->visit('/product/'.$product->id)              ->clickLink('Back')              ->pause(1000)              ->screenshot('test\_laravel\_single\_product\_page\_back\_button')              ->assertSee('Products')              ->assertVisible('#products-grid')              ->assertNotPresent('#product-single')              ->assertSee($product->name);      });  } |

The following screenshot has been generated after clicking on the “Back” button of the previous page:

The last test should be written to test if once a product has been chosen by clicking on its “Inspect” button, #products-grid disappears and #product-single appears instead, showing details of the selected product.

This new view is similar to the first test’s page, but is created with Vue instead of Laravel and sits on the same webpage as the products grid, thus, some of the same checks are available in both tests.

Furthermore, Laravel Dusk has assertions for Vue properties, which can be used to check if the Vue has the expected properties. In this case, the test should check if the Vue’s product property has the correct product information:

|  |
| --- |
| public function test\_vue\_index\_product\_page\_card\_button() {      $this->browse(function (Browser $bro) {          $product = Product::factory()->create();          $bro->visit('/products/index\_vue/')              ->pause(1000)              ->click('#product-card-'.$product->id.' button')              ->screenshot('test\_vue\_index\_product\_page\_card\_button')              ->assertVisible('#product-single')              ->assertNotPresent('#products-grid')              ->assertSee('Category')              ->assertSee($product->code)              ->assertVue('product.name', $product->name);      });  } |