# Testing Guide

Jump to bottom

Andy Cochran edited this page on 30 Oct 2016 · 4 revisions

Foundation is in the process of developing a full test suite for every CSS and JavaScript component. The framework has three kinds of tests:

- JavaScript Unit Tests
- Sass Unit Tests
- Visual Regression Tests

## Running the Tests

There are three testing commands, one for each test suite:

- `npm run test:javascript` : runs JavaScript unit tests.
- `npm run test:sass` : runs JavaScript unit tests.
- `npm run test:visual` : runs a BrowserSync server to view the tests in.

## JavaScript Unit Tests

JavaScript unit tests are stored in the `test/javascript` folder. We use Mocha to run tests, Chai for assertions, and Sinon for test spies and stubs.

The folder has this structure:

- `components/` : tests for components.
- `util/` : tests for utility libraries.
- `lib/` : files for Mocha, Chai, and Sinon
- `index.html` : page that runs the tests.

### Example Test Suite

Refer to the test suite for Toggler to see an example of a fully-tested plugin. In illustrates in particular a few important concepts:

- Grouping tests by method.
- Maintaining a standalone environment for each test.
- Using `should`-style assertions to test both JavaScript objects and HTML.

## Test Structure

Use `describe` blocks to group tests for one plugin, and then use another `describe` block to group tests for plugin methods.

```
describe('Plugin', function() {
  describe('constructor()', function() {
    it('creates a new instance of Plugin', function() {

    });
  });
});
```

## Testing Environment

Each test needs an *isolated environment*, which means anything you do in one test shouldn't affect the outcome of other tests. Since we're mostly testing HTML/JavaScript plugins here, that means most tests will have this process:

- Create the HTML needed for a plugin and add it to the page.
- Initialize the plugin.
- Test a plugin feature.
- Destroy the plugin with its `.destroy()` method.
- Remove the HTML from the page.

You can slim down some of this boilerplate by using Mocha's `afterEach()` function, which allows you to run the same code after each test.

```
describe('Plugin') {
  var $html;
  var plugin;

  afterEach(function() {
    plugin.destroy();
    $html.remove();
  })
}
```

## Assertions

We use the `should` style of Chai assertions, mainly because it makes writing checks for HTML elements really easy. We also use the chai-jquery library, which gives us extra assertions to test the properties of HTML elements.

Here are some common assertions we use to test plugins. Any assertion can be inverted by adding `not` in the chain somewhere. Refer to the documentation for Chai and chai-jquery for more info.

```
// Check if an element has (or doesn't have) a class
$('#plugin').should.have.class('is-active');
$('#plugin').should.not.have.class('is-active');

// Check if an element has (or doesn't have) an attribute
$('#plugin').should.have.attr('aria-hidden');
$('#plugin').should.not.have.attr('aria-hidden');

// Check if an element has an attribute with a specific value
$('#plugin').should.have.attr('aria-controls', 'plugin');
```

## Sass Unit Tests

Our Sass unit tests are written in True. The output is logged with Mocha. Tests are stored under `test/sass`.

At the moment, we only test Sass functions, not mixins. Critical framework features like `grid-column()` and `breakpoint()` have had their logic abstracted into functions, which are wrapped by mixins.

Sass tests use a structure similar to `describe` and `it`. Each function is inside a `test-module()` block, and a specific feature of the function is a `test()` block. Refer to the True documentation for more info on how test structure and assertions work.

The test suite for the breakpoint function is a good model to follow, particularly because that function accepts many different kinds of values. Every possible value you could throw at it is tested, with edge cases and all.

## Visual Regression Tests

Our visual regression tests allow us to test framework features that are difficult to detect with automated testing. They're stored under `test/visual`.

Generally, new visual tests are created as bugs crop up. They allow us to identify bugs in an isolated environment, fix the bug, and then ensure that it doesn't come up again as other framework changes are made.

**To create a visual regression test:**

- With the framework files downloaded and set up, run `npm run test:visual`. Your browser will open a tab with all of the current test files.
- Create a new HTML file in the folder for the component you're testing. (Or create a new folder if it doens't exist yet.)
- Replace the placeholder title with your own title of the format `<h1>Component Name: Feature Name`.
- Below the title, write out the feature of the component being tested, and *the expected behavior*. Others who come to your test later should be able to understand what exactly they're testing.
- Copy the contents of `test/visual/_template.html` into your new file. This is boilerplate code that loads in the needed CSS and JavaScript.
  - If you need to test flexbox mode, change the stylesheet in the `<link>` tag to `foundation-flex.css`.
  - If you need to test right-to-left mode, change the stylesheet to `foundation-rtl.css`.
- Add the required HTML for the issue. *Use the least amount of code needed to recreate the bug.*
  - To add extra CSS, put it in a `<style>` tag.
  - To add extra JavaScript, put it in a `<script>` tag.

Here's an example of an explanation of intended behavior of a component:

- The dropdowns should open underneath the parent element.
- On the right side, the dropdown's right side should be aligned to the right of the parent element.
- On the left side, the dropdown's left side should be aligned to the left of the parent element.

For an example, check out one of the tests for responsive menu classes which have a lot of specific requirements.