accusoft®

Test Design Best Practices Using Mocha Chai, Page Objects, and Linters

Neal Armstrong
Sr. Software Development Engineer in Test

“We need automation.” It’s often the phrase stated by management. What does that mean exactly? Does it mean, “Write test code however you want as long as it automates the product’s functionality?” No. I think not! There are many aspects to writing test automation that are often overlooked and could cause issues later on. I have found that a majority of problems can be addressed by focusing on three areas including living documentation, simplification, and return on investment (ROI). Of course, there are other areas to consider. However, with these suggestions, I believe you will get the most out of your test framework.

Whether you are using a behavior driven development (BDD) or test driven development (TDD) approach to testing, testing can become an arduous task. I have worked with many developers who prefer to write tests after they have the application code written. There is nothing wrong with this approach, as long as tests are written well. To write TDD/BDD tests effectively, one must understand the behaviors of the feature under development. TDD is similar to BDD in that the tests are written before you write the product code. Many times, there are little to no requirements written and as a result, it is left to the developer to investigate and ask the product owner for details. Regardless of the testing approach your team takes, using Mocha as your test framework is a great choice.

Before we get into the details, it is important to understand how Mocha works. Why Mocha? Well, I like frameworks that are simple to use, open source, flexible, and have a great support community. The standard out-of-the-box configuration of Mocha uses the concept of Describe, It, and Hooks. `describe()` is merely for grouping, which you can nest as deep as necessary to clearly ‘group’ your tests into suites. `it()` is a test case, there are usually several it statements within a describe statement. `before()`, `beforeEach()`, `after()`, `afterEach()` are hooks to run e.g. ‘before()’ is run before all tests.

Living Documentation

Test plans and test strategies are often written, read once, then forgotten. A better approach is to transform your test cases into living documentation. Simply put, living documentation is the definition of your features providing a common support dialogue for your business and tech teams.

Take this basic example of using nested describes for consideration:

php.spec.js

```
describe("PrizmDoc - PHP Sample Viewer", function() { // main describe

  before(() => alert("Testing started – before all tests"));
  after(() => alert("Testing finished – after all tests"));

  beforeEach(() => alert("Before a test – enter a test"));
  afterEach(() => alert("After a test – exit a test"));

    describe("Redactions", function() { // nested describe
      it('should allow text redactions', () => // test code);
      it('should allow retangular redactions', () => // test code);
    });
    describe("Annotations", function() { // nested describe
      it('should allow drawing of a circle annotation', () => // test code );
      it('should allow drawing of a rectangular annotation, () => // test code);
    });
});
```

This example covers a small amount of features. However, if you know your product, it should be clear that you have inadequate testing by simply reading. I have nested describes within a main describe statement. This allows grouping of tests providing clear organization of your tests. Since there are multiple Samples, PHP, JSP etc., I will create different javascript spec files accordingly: php.spec.js, jsp.spec.js, etc.

For each sample, there are many test scenarios that can be performed on content, e.g. redactions, annotations, search, etc. Grouping or nesting provides the developer with an easy way to locate the test suite allowing for easy troubleshooting and/or adding additional tests. From a reporting perspective, it provides stakeholders with enough information to understand there is appropriate test coverage for the feature.

Best Practice - Use nested describe statements and describe/it statements that are human readable. Ubiquitously, the test case should be understood regardless of your understanding of code. For example, if you write a single describe statement like this:

describe("sum", function()

It's ambiguous - leaving the question "the sum of what?" - which causes the viewer to dive into more details of the test. This is potentially a waste of time as it might not be the test case the person is looking for. Regardless of the skillset of the person looking at the test case, they should have a basic understanding of 'what' and 'why' but not necessarily the 'how'.

Chai provides an easily readable assertion library and pairs nicely with the Mocha test framework. There are three options: should, expect, and assert. I particularly like expect.

```
var expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(tea).to.have.property('flavors')
  .with.lengthOf(3);
```

For me, these examples are very easy to read; the intent is clear. However, using expect, should, or assert is really preference. I feel expect allows chaining of words which is closer to natural language and promotes the most readability.

Using Mocha with Chai is a great way of providing living documentation for the specifications of the features of your products. Remember, test cases should be easy to read and understand without knowing the specifics of the test code.

Simplification

Test code should be simple but promote reuse. Minimal abstraction and/or levels of indirection should be considered allowing for developers at all levels to write effective test cases.

Best Practice - Use Page Object Pattern. Page Object Pattern is a design pattern that separates specification from implementation. In other words, it allows us to write tests in a

javascript file which calls functions that are abstracted in a separate javascript file. This greatly simplifies the test, making it easier to read and maintain. The complexities of the test are abstracted out into an object which encapsulates the page functionality.

Using the example from webdriver.io:

```
//page.js
export default class Page {
    constructor() {
        this.title = 'My Page';
    }

    open(path) {
        browser.url(path);
    }
}
```

This is the base page that all other pages extend from. This allows you to write functions like open(path) which are necessary for each page object. Therefore, if you are writing a function that all page objects will need to use, write it in this class:

```
// login.page.js
import Page from './page';

class LoginPage extends Page {

    get username() { return $('#username'); }
    get password() { return $('#password'); }
    get submitBtn() { return $('form button[type="submit"]'); }
    get flash() { return $('#flash'); }
    get headerLinks() { return $$('#header a'); }

    open() {
        super.open('login');
    }

    submit() {
        this.submitBtn.click();
    }

}

export default new LoginPage()
```

Notice that we first start by importing the base page. Then our class LoginPage needs to extend Page. Finally, we use this by exposing open function to our test specs by calling super.open('login'). Because the page object is called Login and the path is login, we can abstract this information hiding it from the test case. The test case doesn't need to know the path. It simply needs to use LoginPage and call open().

```
// login.spec.js
import { expect } from 'chai';
import LoginPage from '../pageobjects/login.page';

describe('login form', () => {
    it('should deny access with wrong creds', () => {
        LoginPage.open();
        LoginPage.username.setValue('foo');
        LoginPage.password.setValue('bar');
        LoginPage.submit();

        expect(LoginPage.flash.getText()).to.contain('Your username is invalid!');
    });

    it('should allow access with correct creds', () => {
        LoginPage.open();
        LoginPage.username.setValue('tomsmith');
        LoginPage.password.setValue('SuperSecretPassword!');
        LoginPage.submit();

        expect(LoginPage.flash.getText()).to.contain('You logged into a secure area!');
    });
});
```

The most important thing here is that there is no logic to 'how' we are setting the values. This logic is in the page object class LoginPage.js e.g. get username() { return $('#username'); } This assumes that the HTML has an element with the id 'username'. The page object exposes functions that are available to the test case by importing LoginPage. All we need to do is use LoginPage's functions to perform the test. Notice that even the assertion uses the page object. Also, the example above is not importing Page from the test case.

Page object design is a simple pattern that provides enough abstraction to promote reuse of functions throughout your test framework. It particularly works well with end-to-end tests a.k.a. journey tests.

Return on Investment (ROI)

It is imperative that we consider time as a factor when calculating ROI for our automation suites. Test frameworks should be optimized to provide return on investment where applicable.

Best Practice - Use tags. There could be cases where we might want to run a subset of tests. It is helpful to divide our test suites into smaller groups of tests that can be easily run independently of each other. A strategy for running tests at different times during your release cycle could help decrease the amount of time it takes to complete your testing effort. One solution is to mark tests with tags:

```
it('should respond with the login form @fast', function() {
```

Notice here that this test is tagged with '@fast'. The point of tagging this test as 'fast' would be to run these during development e.g. running tests daily or on merges to a master branch. This strategy would provide the developer with a quick understanding if they broke anything.

```
it('should respond with the file @slow', function() {
```

Notice here that this test is tagged with '@slow'. The point of tagging this test as 'slow' would be to run these tests only before release as these tests take much longer to run and would disrupt the development process.

To run these use --grep @fast as a parameter option on your scripts.

Best Practice - Use linters. Linters analyze code for potential errors. Using specific linters geared for testing can also help improve your ROI in your test framework. Eslint-plugin-mocha has lots of rules that can be applied. Most notably, when tests are written at the global level or when tests are skipped. Eslint-plugin-jest can be used to find tests that contain no assertions. This will save the developers time by warning them before committing bad tests. Linters can help save time and improve the robustness of your test suites ultimately providing more confidence that your tests are not full of common errors.

In summary, these best practices should help make testing more enjoyable. Treating your test code as living documentation is a great way to reduce your documentation efforts and provides non-developers the ability to collaborate on testing. Simplification of your test framework is important because it should be easy to maintain and easy to contribute. ROI is always something that we should keep in mind when developing frameworks. Tagging and Linting can certainly contribute to a higher return on investment when implemented in your test framework.