Testing and Code Coverage

Table of Contents

Node.js package dependencies:

Framework/package	**Install command**	**Documentation**
Mocha	`$ npm install -g mocha --save`	documentation link
Chai	`$ npm install -g chai --save`	documentation link
Istanbul	`$ npm install -g istanbul --save`	documentation link

Test Resources test/ directory:

The Date-A-Dog web app test folder can be found in `Date-A-Dog/api/test` directory. This folder contains all the Mocha JavaScript test files which will be executed when we run the "`npm test`" command. Future added functionality tests can be added in this directory and should adhere to the following file-naming convention; `test_[modulename].js`

Writing new tests:

Our web app uses the Mocha JavaScript framework for testing. Each JavaScript file in `Date-A-Dog/api/www/js` has a corresponding Mocha test file in the `Date-A-Dog/api/test` directory, with test cases targeting functionality for a single JavaScript module. Therefore, two cases arise when writing new tests:

1. Implementing new functionality for an existing JavaScript module, i.e. `js/[modulename].js`
   - Add test cases to existing mocha test file inside the test directory

2. Implementing a new JavaScript module, i.e. `js/[newmodulename].js`
   - Create new test file `test/test_[newmodulename].js`

Note: Refer to next section on how to structure mocha test files.

Mocha test file structure:

Test files start with *require* statements which import any dependencies needed to conduct testing.

```
var assert = require("assert");
var expect  = require("chai").expect;
var module = require("../www/js/modulename");
```

The outermost `describe()` function takes two parameters: the first parameter is a string description of the module targeted by this test, the second is an anonymous function which contains one or more calls to nested `describe()` functions, each intended to test a specific function of the module. Example:

```
describe("[modulename]", function(){
  // testing foo
  describe('Testing modulename.foo()', function () {
    // test cases here
    });
    … More test cases …

  });

  // test bar()
  describe('Testing modulename.bar()', function () {
    // Test cases here
  });

  … More function tests…
});  // end of modulename test
```

As shown above, nested `describe()` functions also take two parameters: the first a string description of the function being tested, and the second an anonymous `function()`. Test cases are performed by calling `it()` inside the anonymous function passed to the nested `describe()`.

The `it()` function takes two parameters: first a string description of the test case, and second an anonymous function containing all assert statements to be evaluated during testing. Example;

```
    // Test case
    it("0. Test case for modulename.foo()" function() {
      assert.Equal([result], [expected]);
    });
```

Our final mocha test file ends up looking something like this;

```
var assert = require("assert");
var expect  = require("chai").expect;
var module = require("../www/js/modulename");

describe("[modulename]", function(){
  // testing foo
  describe('Testing modulename.foo()', function () {
    // Test case
    it("0. Test case for modulename.foo()" function() {
      assert.Equal([result], [expected]);
    });
    … More test cases here…

  });

  // test for bar()
  describe('Testing modulename.bar()', function () {
    // Test case
    it("0. Test case for modulname.bar()" function() {
      assert.Equal([result], [expected]);
    });
    … More test cases here…

  });
  … More function tests here…
});
```

Android Tests

The Android test suite uses JUnit. To create a new test method, simply add the "@Test" annotation before the test method's declaration. Test classes containing any test methods should add the following imports to use the JUnit framework.

```
import org.junit.Test;

import static org.junit.Assert.*;
```

How to run test suite:

Tests must be run from the `Date-A-Dog/api/` directory by executing the following linux command:

```
$ npm test
```

This will execute all mocha test files found inside `Date-A-Dog/api/test` directory, and output the test results to console.

Android tests can be run through Android Studio (right click tests and select Run Test), or by running the following from the project root directory:

```
$ ./gradlew test
```

How to run coverage analysis:

Our web app uses the Istanbul test coverage tool. To execute the coverage analysis, run the following command from `Date-A-Dog/api/` directory;

```
$ istanbul cover _mocha -- -R spec
```

This will first run the tests, print test results, followed a summary of the test coverage to the console.

To run coverage analysis on the Android app, you can use Android Studio. First, select the test classes you want to include in the analysis. Then right click them and select Run with Coverage. The tests will be run and coverage analysis for lines, paths, etc. will be displayed within the Android Studio GUI.