LESSON 4: TESTING YOUR CODE – PART 2

ENCRYPTION

In web development, encryption means using an algorithm to turn a string into a series of characters that is (hopefully) undecipherable as the original string. I say 'hopefully' because a quick Google search will reveal online applications for decrypting encoded strings. For developers, our role in providing online security means utilising encryption as one of our methods of defence and not relying on it as our only means.

There are many ways to "hash" a string into encrypted code. Two of the most popular algorithms are MD5 and SHA1.

What's important to consider is that these methods aren't merely character replacement (e.g. "A" = "H"), but sophisticated mathematical operations that turn any string into a set number of characters. So, "a" and five paragraphs of Lorum Ipsum will both generate a string the same number of characters long (32 in the case of MD5, and 40 in the case of SHA1).

Another strategy to help fight decryption is to add a "salt" - an extra string that gets mixed in with the encryption. It serves as a key to unlocking the encryption. You cannot decrypt the string without knowing the salt.

PHP has several built in methods for encrypting strings. JavaScript, however, must rely on external libraries or services such as CryptoJS.
https://code.google.com/archive/p/crypto-js/

JAVASCRIPT TESTING WITH LIBRARIES/FRAMEWORKS

As laborious as Unit Testing is, any code meant for production demands it, and developers are expected to be able to test their code. Test-driven-development goes so far as to create the tests for the code, before creating the code.

Fortunately, there are numerous testing libraries and frameworks that make Unit Test creation and execution quite simple.

A testing library or framework lets you create tests for your functions and markup, and provides the results of the test. An individual test is referred to as a Test Case, and many Test Cases make a Test Suite.

In many instances, the syntax looks like a sentence in English. This follows the **Behaviour-driven-development** (BDD) technique that…

> *"…focuses on writing descriptive tests from the business perspective. BDD extends TDD by writing test cases that test the software behaviour (requirements) in a natural language that anyone (not necessarily a programmer) can read and understand. The names of the unit tests are sentences that usually start with the word "should" and they are written in the order of their business value." (Saleh, p. 31)*

For example, a test name might be, **`"The login validator should display an error message if the password is an empty string."`**

The testing of the code can also look like a sentence in English. Commonly, it is referred to as an **assertion**. You assert the condition, and what you expect the result will be: **`expect(2+2).toEqual(4)`**
In the Jasmine example above, this test asserts that 2 plus 2 will be 4.

Besides Jasmine, some of the most popular libraries are:

- QUnit (used by jQuery)
- Mocha (hugely powerful but not beginner-friendly)
- YUI Test (part of the Yahoo! User Interface JavaScript Library)
- JsTestDriver (JSTD) (Can also run other testing frameworks)

JASMINE

As previously mentioned, Jasmine is one of most popular testing frameworks because much of it is written in plain English, and both the tests and results can be read by non-technical people. (Saleh, p. 31)
Beyond its ease-of-use, it is also extremely versatile and powerful. You can test functions, chunks of HTML and even asynchronous operations. It integrates with AngularJS and other popular frameworks and script runners.

To incorporate Jasmine, go here:
https://github.com/jasmine/jasmine/releases

and download a stable release of **jasmine-standalone-*x.x.x*.zip.**

After unzipping, you will find three folders and one file:

- **MIT.LICENSE** – the license agreement for usage
- **SpecRunner.html** – this is the file you run in a browser to execute your tests
- **lib** – this is where the Jasmine code is, no need to touch it.
- **spec** – this is where the js files with your Jasmine tests will go. Jasmine refers to each test case as a 'spec', while a group of specs is a 'suite'.
- **src** - this is where the js files that hold your functions will go

To run a Jasmine test

1. Add your js file to the src folder
2. Add a <script> reference to your **source** file in SpecRunner.html
3. Add a <script> reference to your **spec** file in SpecRunner.html
4. Create a spec js file and add to the spec folder.
5. Create the suites of specs for your functions in the spec file.
6. Run SpecRunner.html in a browser to see the results.

Matchers

In testing your code, there are several different built-in methods to compare your expectation with the result. Jasmine refers to these methods as, 'matchers'. In the example above, we used the toEqual() matcher. Other matchers include:

- **toBe** (similar to **toEqual**, but not as thorough- it can't check arrays)
- **toBeUndefined** / **toBeNull** / **toBeTruthy** (checks for type)
- **toContain** (checks a string or an array for a desired substring)
- **toBeLessThan** / **toBeGreaterThan** (as you would expect)
- **toMatch** (compares a result to a string or regular expression)
- **toThrow** (expects an error to be thrown)

Note, most matchers have opposites (toBeTruthy has toBeFalsy), or you can use the .not method to check for the opposite:

```
expect(true).not.toEqual(false).
```

(Saleh, pp. 39-43)

Reference

Saleh, H. (2013). *JavaScript Unit Testing.* Birmingham UK: Packt Publishing.