JavaScript: Debugging and Testing

medium.com/@csg.riskgame/javascript-debugging-and-testing-5cb2c6aae425

Introduction:

In this article, we will discuss the concepts of debugging, testing support and unit testing frameworks in the context of JavaScript, as well as a video demonstration on how to perform simple debugging in JavaScript. To start, we will define and explore debugging.

Debugging:

Since errors can happen syntactically, logically, or at any point in the code, it is imperative for a developer to be able to easily detect and correct errors. Debugging itself may not be simple, but the tools available for debugging in JavaScript are simple.

Most browsers have debugging tools built into them. Browsers such as Chrome, Firefox, Opera, Safari, and Internet Explorer allow the user to access the debugging tool by pressing *F12* or by following a series of menu options.

Chrome's DevTools, in particular, is a great option for testing JavaScript client-side scripting applications, such as web pages. It is a simple and powerful tool that can be accessed by clicking the *"inspect element button"* in the Chrome browser. If you would like to implement more advanced unit testing or testing for server-side scripts, such as Node.js, refer to the last section of this article about unit testing frameworks.

Using the DevTools, we can utilize crucial debugging techniques such as setting breakpoints, stepping through code, logging to console to check variables, benchmark time complexity, perform a stack trace, watch specific function or argument calls.

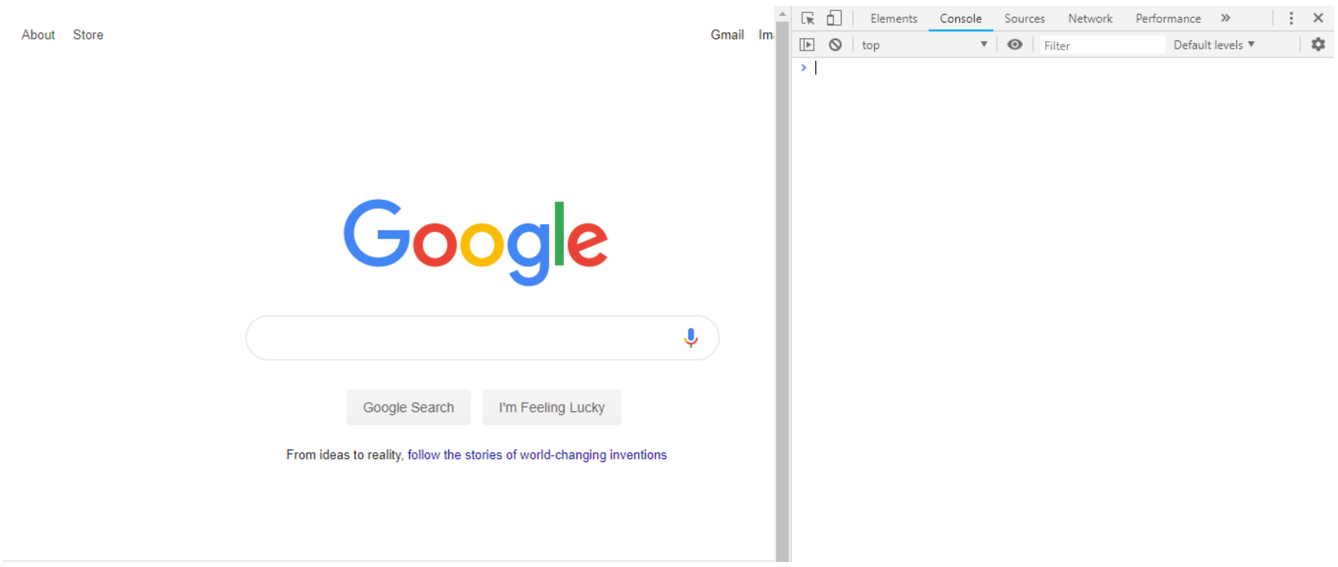
Debugging Demonstration:

To further explain debugging in JavaScript and showcase the use of Chrome's DevTools, we have created a video linked here:

Testing Support:

Chrome’s Devtools Console is an important tool when it comes to testing JavaScript code. The Console allows you to view diagnostic information about a page and runs the actual JavaScript code.

Press *Command+Option+J* in Mac or *Control+Shift+J* in Windows and Linux to open the Console



Example-opening the Console in Google’s homepage

The Console API allows users to log messages from their JavaScript code to the Console. This is particularly helpful when testing buttons and other interactive features. To write a message to the Console, use *console.log(‘your desired message’)* in your JavaScript code. Once the action that triggers the console log function is executed, the message will immediately appear in the Console.

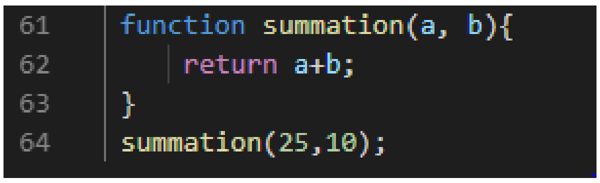
It is helpful to click *“Clear Console”* if the Console becomes too cluttered from previous executions. Or use the *filter* text-box provided by the Console to see particular messages that contain specific text chosen by the user.

The browser can also log error messages to the Console. This is helpful for pointing out flaws that may not be caught by the debugging process. For example, logging a *TypeError* or a *404 (Not Found)* error. These errors indicate a user is trying to access a property or file that does not exist.

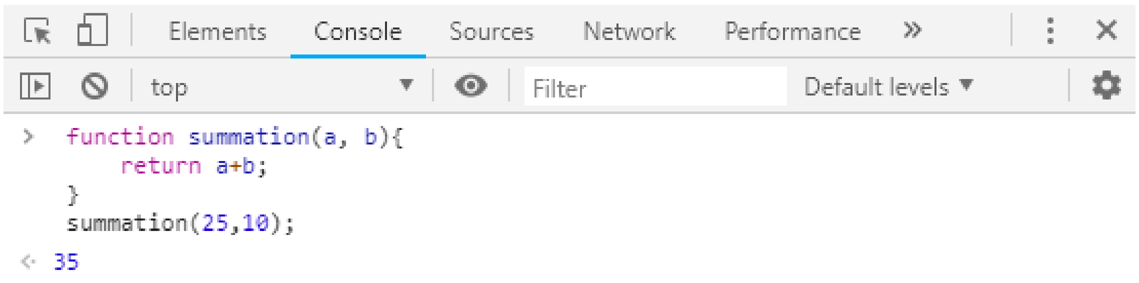
You can also test specific portions of JavaScript code by typing them directly into the Console.

Say, you want to just test the summation function shown below.

You can type the code directly into the Console and run it by hitting enter.



```
function summation(a, b){
    return a+b;
}
summation(25,10);
```



```
> function summation(a, b){
      return a+b;
  }
  summation(25,10);
< 35
```

While testing, you may want to change how a page runs or looks. You can do this directly in the Console. For example, if you would like to change the text of a certain element you could type into the Console the following.

document.getElementById('elementName').textContent = 'desired modified text'

It is important to note that you can only modify items within the same execution context. This means that you cannot modify embedded items since they are in a different execution context. An example of this is an iframe, a common ad-distribution method. To switch the execution context, highlight the property to be changed within the web page to the left of the Console and click *inspect*, then *escape*. The Console is now in that properties execution context.

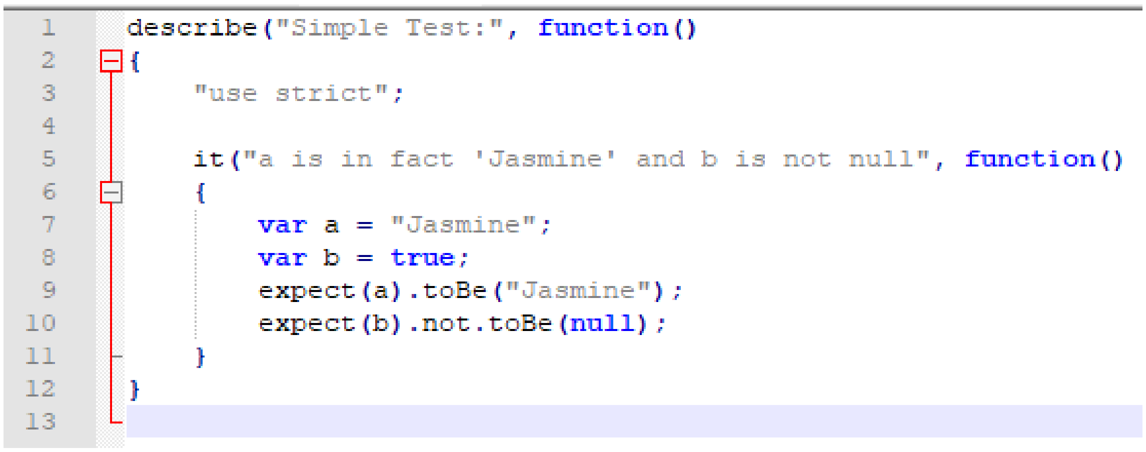
Unit Testing Framework:

Testing is an important step in the development process that helps to ensure the software behaves correctly under different conditions. Unit testing is the primary level of testing which enables the developer to test small individual units of the program as they are developed to ensure they behave as desired. JavaScript is usually used for front end development, therefore JS unit tests are mostly in the browser (front end) which may lead to some challenges. To confront the challenges of creating unit tests to cover as many aspects of the program as possible, developers use unit test frameworks. Almost every language is equipped with at least one, unit testing framework. For example, JUnit is a unit testing framework for Java.

As mentioned earlier, writing unit tests for JavaScript may be somewhat challenging since it involves the front-end. Therefore, JavaScript is equipped with multiple unit testing frameworks, each of these testing tools have some characteristics that could make one more beneficial to use over the other depending on the program to be tested. The remaining of this section of the article will introduce some of the popular unit testing frameworks in JavaScript and their characteristics.

1. **Jasmine:**

Jasmine has multiple advantages. Jasmine is independent of browser, platform, framework and it can be used for both test-driven development and behavioral driven development. Jasmine provides command line utility, also the syntax is very easy. Jasmine provides assertions, spies, and mocks and it allows the developer to add any library based on the need.



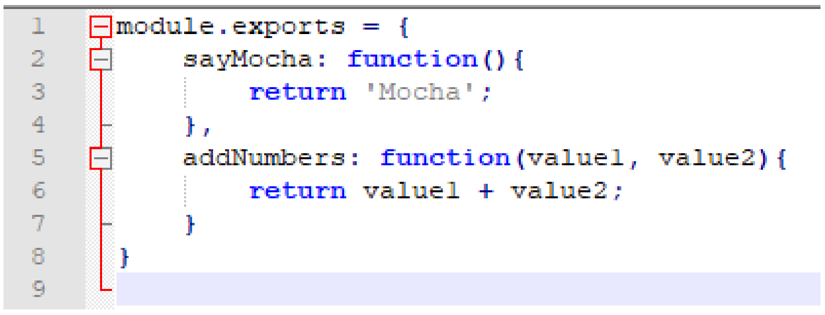
```
describe("Simple Test:", function()
{
    "use strict";

    it("a is in fact 'Jasmine' and b is not null", function()
    {
        var a = "Jasmine";
        var b = true;
        expect(a).toBe("Jasmine");
        expect(b).not.toBe(null);
    }
}
```

A simple unit test using Jasmine

2. **Mocha:**

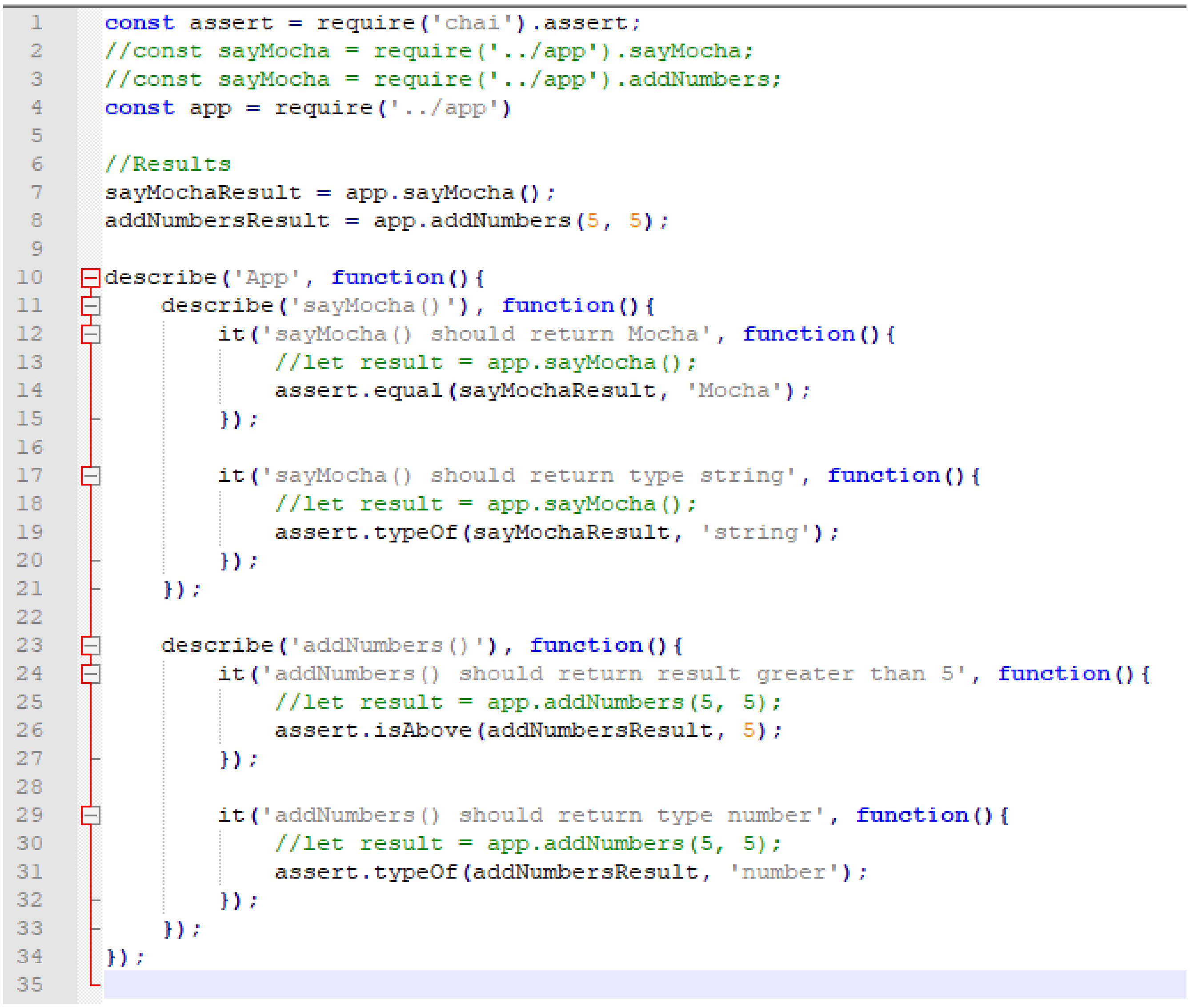
Mocha which provisions for asynchronous testing, supports Node.JS programs, generates test coverage reports and runs in the browser.

Mocha is liked by many React developers. Mocha's popularity is due to the flexibility it provides, it allows access to assertions, spices, mocks through adding libraries such as Chai and express.js assertion libraries, on the other hand, the ability to add additional assertion libraries can be a disadvantage for the complexity it can cause. Also, Mocha allows the developers to write their own testing libraries. Another disadvantage of Mocha is, not allowing the tests to run in any order, as well as not supporting auto mocking or snapshot tests. Consider a JavaScript code like the one shown below:



```
module.exports = {
    sayMocha: function(){
        return 'Mocha';
    },
    addNumbers: function(value1, value2){
        return value1 + value2;
    }
}

```

app.js

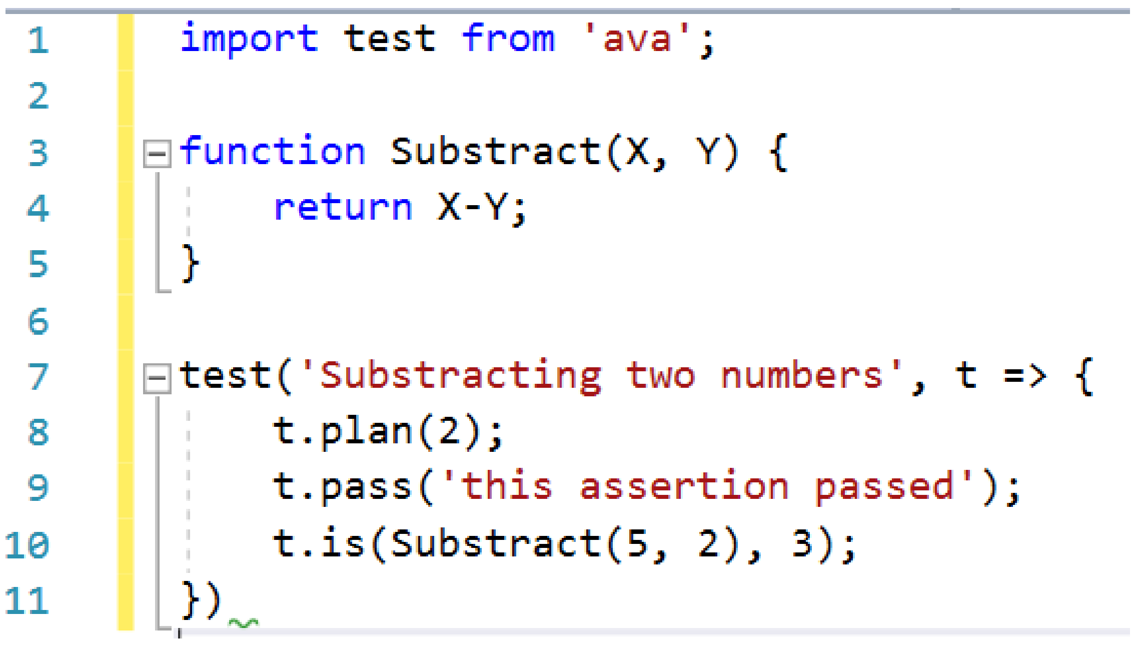


```
const assert = require('chai').assert;
//const sayMocha = require('../app').sayMocha;
//const sayMocha = require('../app').addNumbers;
const app = require('../app')

//Results
sayMochaResult = app.sayMocha();
addNumbersResult = app.addNumbers(5, 5);

describe('App', function(){
    describe('sayMocha()', function(){
        it('sayMocha() should return Mocha', function(){
            //let result = app.sayMocha();
            assert.equal(sayMochaResult, 'Mocha');
        });

        it('sayMocha() should return type string', function(){
            //let result = app.sayMocha();
            assert.typeOf(sayMochaResult, 'string');
        });
    });

    describe('addNumbers()', function(){
        it('addNumbers() should return result greater than 5', function(){
            //let result = app.addNumbers(5, 5);
            assert.isAbove(addNumbersResult, 5);
        });

        it('addNumbers() should return type number', function(){
            //let result = app.addNumbers(5, 5);
            assert.typeOf(addNumbersResult, 'number');
        });
    });
});
```

appTest.js

3. **AVA:**

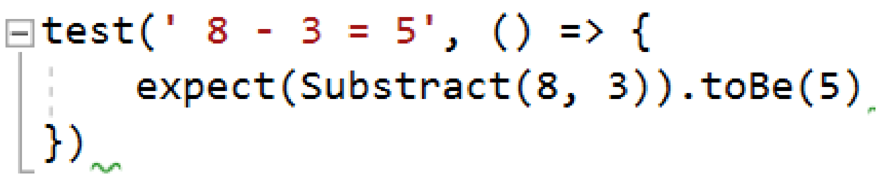
Another widely used testing framework for JavaScript is AVA, which can be easily installed using the JavaScript package manager. AVA is very easy to use, has simplex syntax, and supports asynchronous testing. Unfortunately, AVA doesn't have built-in mock support, we would need additional libraries such as Sinon.js to be able to use spies, stubs, and mocks. AVA can be used for front-end and back-end of an application developed in JavaScript, and it only allows writing atomic tests. Overall, AVA is a nice framework to use, its concise API and minimalist approach provide very good performance. Here is a simple test for validating a function that performs subtraction, implemented using AVA framework.



```
import test from 'ava';

function Substract(X, Y) {
    return X-Y;
}

test('Substracting two numbers', t => {
    t.plan(2);
    t.pass('this assertion passed');
    t.is(Substract(5, 2), 3);
})
```

4. **Jest:**

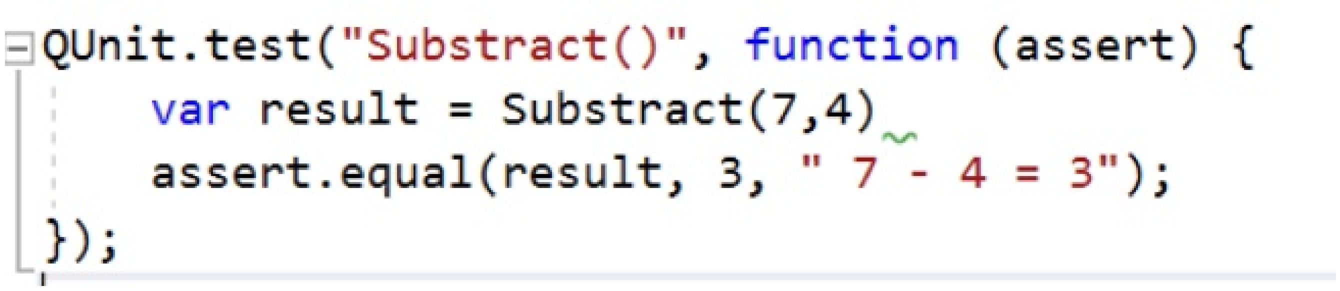
Jest is an open source framework created by Facebook. Although Jest is known as the best testing framework for React applications, it is not limited to React framework. Jest can be used with Angular, Node.js, Babel and many other frameworks. Jest has a well-supported API that provides almost everything a developer needs such as mocks, assertion libraries, snapshot testing, generates code coverage reports, and supports parallel testing. Jest can be easily installed using the package manager and is a great testing framework to use. Here is a unit test to verify the same subtraction function from above, using Jest:



```
test(' 8 - 3 = 5', () => {
    expect(Substract(8, 3)).toBe(5)
})
```

5. **QUnit:**

QUnit is an easy-to-use testing framework which is commonly used for JQuery applications. QUnit can test Node.js programs as well as supporting in browser tests, essentially both client and server side can be supported using this framework. QUnit can test any JavaScript code even itself and supports synchronous and asynchronous testing. Besides having the assertion libraries, it allows you to make custom assertions. There are also many plugins available to better coordinate this framework with the needs of the developer. Here is an example of a unit test using QUnit for the same subtraction function introduced earlier in this section:



```
QUnit.test("Substract()", function (assert) {
    var result = Substract(7,4)
    assert.equal(result, 3, " 7 - 4 = 3");
});
```