Centi TF (Testing Framework) 0.1alpha

Jaap Murre, University of Amsterdam, February 2020, jaap@murre.com

Version: 0.1alpha

Introduction

I have been a happy user of the Jest Testing Framework for a while now. As an intellectual exercise, I was wondering whether it is possible to create a much smaller framework that uses less verbose testing code, while still covering many useful test cases. Many projects are quite small and learning and setting up Jest or another framework then seems overkill for many programmers. This means that testing is conveniently 'forgotten'. To the detriment of the quality of the code base.

My objectives were the following:

- Small code base (100 lines of code)
- Each tests a short single without any repetition (e.g., avoid saying "Adding 2 + 2" followed by a function call "add(2+2)")
- Useable in Node.js and in the browser
- Small set of out-the-box matcher tests, covering many practical use-cases
- Very easily extendable with custom matcher tests
- Default support of (normal and typed) arrays, also for custom tests, without writing extra code
- Support for test groups
- Use in either string-based (eval'ed) mode or normal (verbose) mode

I committed the following sins while writing this framework:

- To avoid having to first write function calls and then also describing them, I use `eval()`
- To enable Node.js to access the context of the eval'd code, I use `with`
- I know

Usage

Setup

Using Centi TF in Node.js, `require` it so:

```
const expect = require("./centi_testing_framework");
```

In the browser do:

```
<script src="centi_testing_framework.js"></script>
```

This will send output to the console, but this can be redirected to the page with a simple bridging function (see below).

If you want to test Node.js code, you might define the following two functions:

```
expect.context.add = function(a, b) { return a + b; }
function subtract(a,b) { return a - b; }
```

This also works in the browser but there you have the option of forgoing adding the function to the context:

```
add = function(a, b) { return a + b; }
```

If you do not add a function to the expect.context in Node.js, like `subtract()`, you cannot use the string-based testing.

Writing tests

An example test:

```
expect("add(1,2)").toBe("3");
```

Because the code is specified as strings, which are than eval'ed, it is not necessary to provide any additional description. The response for this test is (with PASS in green):

```
PASS  Expect add(1,2) === 3
```

Because `subtract()` is not in the expect.context, its usage cannot be specified in a string when you are using Node.js (though it still works in the browser). The reason is that a Node.js module has no access to the scope in which the function is defined. Adding it to the expect.context remedies this. This also means that the system cannot discover the code of the call and the user must, therefore, provide an explicit message, like in Jest or other frameworks. For example:

```
expect(subtract(1,2),"subtract 1 from 2 to equal -1").toBe(-1);
```

which gives:

```
PASS  Expect subtract 1 from 2 to equal -1
```

The philosophy of Centi TF is to make use of the test code strings as much as possible for the generation of test results. It is possible to add a message to a string-based test, which is then shown above in the output:

```
expect("add(-4,-2)","Adding negative numbers").toBe("-6");
```

which outputs:

```
      Adding negative numbers:
PASS  Expect add(-4,-2) === -6
```

If you are using the string-based representation, it is necessary to use extra quotes so that the string is not eval'ed.

```
expect("'Hello' + ' ' + 'World'").toBe("'Hello World'");
```

Without the single quotes, this test would fail.

Matchers

The framework includes a baker's dozen of matchers:

- toBe: Uses and reports the === operator for comparison
- toBeEquivalent: Uses and reports the == operator for comparison
- toBeExactly: Uses `Object.is()` for comparison
- toBeCloseTo: Uses `Math.abs(assertion - exp) < expect.settings.tolerance` for comparison, where tolerance is 0.000001 by default. It is settable to another value like this: `expect.settings.tolerance = 0.05`.
- toBeGreaterThan: Uses and reports the > operator for comparison
- toBeGreaterThanOrEqual: Uses and reports the >= operator for comparison
- toBeLessThan: Uses and reports the < operator for comparison

- toBeLessThanOrEqual: Uses and reports the <= operator for comparison
- toBeTruthy: Uses `!!exp === true` to test
- toBeFalsy: Uses `!exp === true` to test
- toBeAnInstanceOf: Uses `exp instanceof assertion` to test
- toBeNaN: Uses `isNaN(exp)` to test
- toMatch: Uses `exp.search(assertion)` to test

Note that toBeTruthy() can also easily be achieved with the toBe() function, but we kept it anyway because it seems more expressive, conveying more clearly what it is that is being tested.

```
expect("add(1,1)").toBeTruthy();
```

is equivalent to writing

```
expect("!!add(1,1)").toBe("true");
```

For other things like string or array length, however, we opted not include special-purpose matchers, because it is both easy and expressive to write:

```
expect("'Hello'.length").toBe("5");
```

making an additional matcher unnecessary or even undesirable (more stuff to look through):

```
expect("'Hello'").toHaveLength("5"); // This matcher is not included!
```

You could easily include this matcher if you insist on having it. See below.

All of the above matchers may be negated by including .not, e.g.,

```
expect("resultString").not.toBe("'error'");
```

You can insert .not only once (the second time is ignored).

Automatic support of arrays

The matchers above, as well as any custom matchers you might add, will automatically support array-based comparison, where array is anything with a length property that is not a string or function. This means that TypedArrays are supported too. To communicate to the framework that you want the matcher to be applied to all elements on a element-by-element basis, use the .content property.

The following tests are useful when working with 32bit floating-point calculations and would pass:

```
expect("[10,20]").contents.toBeCloseTo("[10.00000001,20.00000001]");
expect("[100,200]").not.contents.toBeCloseTo("[10.00000001,20.00000001]");
expect("[100,200]").contents.not.toBeCloseTo("[10.00000001,20.00000001]");
```

As you can see the order of the .not and .contents is free.

Be default, all paired tests must pass for the whole test to pass. That is, there is an implied .all property. You may add it for clarity in the test but it is ignored in processing and reporting:

```
expect("[10,20]").all.contents.toBeCloseTo("[10.00000001,20.00000001]");
```

If it suffices that just one of the tests passes (or more, but not all), you can use the .some property. The tests are still done in a paired manner, so you must provide two arrays of equal length (else it will fail):

```
expect("[1,2,3,4,5]").contents.toBe("[1,1,1,1,1]"); // Fails
expect("[1,2,3,4,5]").some.contents.toBe("[1,1,1,1,1]"); // Passes
expect("[2,2,3,4,5]").some.contents.toBe("[1,1,1,1,1]"); // Fails because no
tests passes
```

Because there are many use cases where we are looking for a single needle in a haystack, we can also write

```
expect("[1,2,3,4,5]").some.contents.toBe("1"); // Passes
```

This usage of .some covers most of the array-contain cases, which do not need to be added separately.

Using a single element for comparison also works with the implied .all:

```
expect("[1,1,1,1,1]").all.contents.toBe("1"); // Passes
expect("[1,1,1,1,1]").contents.toBe("1"); // Passes
```

We can combine .some with .not:

```
expect("[1,2,3,4,5]").some.contents.not.toBe("1");  // Fails
```

The case where we have an array of comparison values is currently not supported. In case we have two arrays of unequal length the array comparison will fail:

```
expect("[1,2,3,4,5]").some.contents.toBe("[1,3]");  // Fails
```

Objects

Matching these has less support at the moment and must written out in more detail. Because of the terse nature of the tests, this may still be convenient enough. For example:

```
expect.context.house = { // Adding to the context is only required when
working with Node.js
    bath: true,
    bedrooms: 4,
    kitchen: {
        area: 20
    }
}

expect("house.bath").toBeTruthy();
expect("house.bath").not.toBe(undefined);
expect("house.pool").toBe(undefined);
expect("house.kitchen.area").toBeGreaterThan(15);
```

All four tests pass and show some possibilities.

Custom matchers

New matchers can be added directly to the expect.matchers object. They consist of an object with at the very least the the test function 'fun'. For example:

```
expect.matchers["toBeWithin"] = { fun: (exp,assertion) => exp > assertion[0]
&& exp < assertion[1] }
expect("add(1,3)").toBeWithin("[2.5,5]");
expect("add(1,3)").toBeWithin("[12.5,25]"); // Fails
```

The output is:

```
PASS  Expect add(1,3) to be within [2.5,5]
FAIL  Expect add(1,3) to be within [12.5,25]
```

Because we chose to provide the arguments to the assertion as an array this matcher cannot be combined with an array:

```
expect("[13,14,15]").toBeWithin("[12.5,25]"); // Fails
```

We can rewrite the matcher and use an object:

```
expect.matchers["toBeWithin"] = { fun: (exp,assertion) => exp > assertion.min
&& exp < assertion.max }
expect("[13,14,15,16]").contents.toBeWithin("({min:12.5,max:25})"); // Passes
```

A tricky point here is that whenever you specifiy an object literal, like here, to be eval'ed, you must wrap it in round brackets, or else eval() will think it is a block and choke. Another option is to allow a string:

```
expect.matchers["toBeWithin"] = { fun: (exp,assertion) => exp >
assertion.split(',')[0] && exp < assertion.split(',')[1] }
expect("[13,14,15,16]").contents.toBeWithin("'12.5,25'"); // Passes
```

This works and gives a useful output:

```
PASS  Expect contents of [13,14,15,16] to be within '12.5,25'
```

With matchers that take only a simple argument, this is more straightforward:

```
expect.matchers["toHaveLength"] = { fun: (exp,assertion) => exp.length ===
assertion }
expect("[13,14,15,16]").toHaveLength("4"); // Passes
```

Notice that we have dropped the .contents from the test because we are testing an array-level property. This gives output:

```
PASS  Expect [13,14,15,16] to have length 4
```

We can now also test an array of arrays:

```
expect("[[1,2],[3,4]]").contents.toHaveLength("2") // Passes
```

with output:

```
PASS  Expect contents of [[1,2],[3,4]] to have length 2
```

If we do not like how Centi TF transforms the test name 'toHaveLength' into a reported test phrase, we can add our own by specifying the .test property (which is optional). For example:

```
expect.matchers["toHaveLength"] = { fun: (exp,assertion) => exp.length ===
assertion, test: "to be of length" }
```

Now

```
expect("[13,14,15,16]").toHaveLength("4"); // Passes
```

will give output:

```
PASS  Expect [13,14,15,16] to be of length 4
```

But we could have achieved the same by naming the matcher toBeOfLength as the camel-case conversion would transfer this to 'to be of length'.

Test Groups

By writing

```
expect.end()
```

a summary is output, such as:

```
----------- END TEST ------------

    Passed 40 out of 55 tests
```

You can also precede a number of tests with:

```
expect.start("Array tests");
```

This will output:

```
START Array tests
```

then you could have, say, 13 array tests and then end with:

```
expect.end("Array tests");
```

This will then output a test-group-based summary, while still also counting total passes and total tests:

```
END Array tests
    Passed 8 out of 13 tests
```

Using expect.start() without arguments, will reset the global counters of passes and total tested and output a reset message.

Usage in the browser

If you want to make use of testing on a webpage, this will work fine if you open the console (with F12 on Windows); you can then inspect the messages. If you want, you can also redirect the console.log function to the webpage itself as follows:

```
<!DOCTYPE html>
<html>
<head>
<title>Centi Testing Framework</title>
<script src="centi_testing_framework.js"></script>

</head>
<body>
<h1>Testing the Centi Testing Framework</h1>
<pre id="mylog" style="font-size:1.2em;"></pre>



<script>

    const originalConsoleLog = console.log;
    console.log = (...args) => {
        let s = "<b>";
        args.map(arg => s += arg + " ");
        s = s.replace("\x1b[42m","<span style='background-
color:lightgreen;'>");
        s = s.replace("\x1b[41m","<span style='background-color:#fbb;'>");
        s = s.replace("\x1b[0m","</span>");
        s = s.replace(/\n/g,"<br />");
        document.querySelector("#mylog").innerHTML += s + "</b><br />";
    }

    // Here will be all your scripts
    console.log = originalConsoleLog; // Redirect console.log to the original
one

</script>

</body>
</html>
```

The output of console.log is written to a pre-element so that the layout (based on spaces) is preserved. The console.log escape codes for color are converted to styles and the newline symbol \n to the br-element (break).

If you still need to write to the actual console but cannot because of the redirection you may consider using console.debug() (or console.info()), which still works as usual and is identical to console.log().