Node.js - Guide

section 5 - Testing - Assertion Testing

- Dr Nick Hayward

A brief outline of testing Node.js apps with assertions.

Contents

intro

An *assertion* library helps us make assertions about values within our tests. This might include assertions about their types, the value itself, if an array contains a given element, and so on.

There are many examples of assertion libraries, including the popular library called **expect**. We can use this library to pass some values and then make some assertions.

e.g. pass a number value, and assert that it needs to equal 22.

URL is as follows,

- https://github.com/mjackson/expect

setup

We can install this assertion library using NPM,

```
npm install expect --save-dev
```

and then start to integrate with our app's testing.

testing - using *expect* assertion library

We can start by requiring the assertion library in our app's test file, e.g.

```
const expect = require('expect');
```

We can then start to use the `expect()` method with a passed value, which then allows us to test various assertions. e.g.

```
// call expect function and pass value - then assert against this passed value
expect(total).toBe(33);
```

The ***expect*** method `toBe()` simply asserts that one passed value is equal to another value (using strong equals, `===`). i.e. the value passed to `expect()` is equal to the value passed to `toBe()`. This method expects a ***truthy*** return for this assertion.

The `expect()` library with an assertion will also return meaningful error message, where applicable.

We can use this assertion as part of the `it()` testing with Mocha. It simply becomes the logic for testing in Mocha's `it()` method.

testing - chaining assertions

We can also chain assertions, such as testing for a values type whilst checking the actual value. e.g.

```
expect(total).toBe(33).toBeA('number');
```

So, our testing is now ***asserting*** that the total should be `33` and a `number` type.

We can also use method to check for inequality, e.g.

```
expect(total).toNotBe(17);
```

testing - checking arrays, objects &c.

Methods such as `toBe` are not great at checking values and structure for arrays, objects, &c.

For objects and arrays, the problem occurs because `toBe()` is using the strict equals comparison operator. This operator is checking type and value, so two separate objects/arrays will not be the same, even if we're comparing two similar name:value pairs in objects. e.g.

```
expect({total: 33}).toBe({total: 33});
```

These would appear to be equal. However, `expect` does not see them as equal because they're different objects.

To check object values, we can use `toEqual()` instead with `expect`, e.g.

```
expect({total: 33}).toEqual({total: 33});
```

This will now work as expected for the comparison. Likewise, we can perform an inequality check for objects,

```
expect({total: 35}).toNotEqual({total: 33});
```

Both of these equality tests will pass successfully.

We can also check if an array or object includes something, e.g.

```
expect([2,4,6]).toInclude(4);
```

This will return successfully, obviously as `4` is in this passed array. We can also check for exclusion, e.g.

```
expect([2,4,6]).toExclude(3);
```

which will likewise return successfully as `3` is not in the passed array. We can also use this structure with objects, e.g.

```
expect({
  title: "first book",
  author: "new author",
  year: 1997
}).toInclude({
  year: 1997
})
```

In this example, we're passing an object with three name:value pairs. However, we're only interested in testing for a single name:value pair, `year`, and the value of `title` and `author` are not important to the success of this assertion. In this example, the assertion test will complete successfully.

`expect()` also supports lots of different mathematical operations and comparison operators.

testing - async checks and testing

We can update the `utilities.js` file to simulate some async calls for testing purposes. e.g. to simulate some delays - such as a database or remote service request

```
module.exports.asyncAdder = (a, b, callback) => {
  setTimeout(() => {
    callback(a + b);
```

```
  }, 1000); // > 2 secs & mocha will assume there's a problem with the
request...
};
```

In this `asyncAdder()` function, we might use a standard call to a timeout to simulate a delay from a remote source, includng a DB, remote service &c.

n.b. we need to keep the timeout less than 2 seconds (2000 ms) to ensure that Mocha doesn't default to an error. If the timeout is > 2 secs, Mocha assumes there's a problem and returns an error.

Then, we need to update the `utilities.test.js` test file, e.g.

```
it('should asynchronously add two passed numbers', (done) => {
  // call async test function - pass two numbers and callback function
  utils.asyncAdder(3, 7, (sum) => {
    // use an assertion to test return and type
    expect(sum).toBe(10).toBeA('number');
    done();
  });
});
```

For the return value from the test `asyncAdder()` function, we can then run a chained assertion for the return value and type.

n.b. to make this Mocha test and Expect assertion work together correctly for an asynchronous call, we start by adding `done` as an argument value to the initial Mocha `it()` test. This tells Mocha that the test is asynchronous. Then, the `asyncAdder()` function is called, 1000 ms is waited, and the return will come back. The assertion is then run and checked, and we can now tell Mocha that the async test is complete by calling `done()`.

testing - test an Express web app

To be able to test Express web apps, we can still use Mocha and Expect with additional testing libraries for http requests &c.

So, we need to add a basic server to our existing test app, e.g.

```
npm i express --save
```

As expected, this will install the Express web framework module for the current project.

We can then setup a basic Express server in a new directory, `server` in a `server.js` file. This is the server we can the use for testing Express and HTTP requests, e.g.

```
const express = require('express');

var app = express();
```

```
app.get()
```

For testing purposes, we can use the ***Supertest*** library, which was created by the original developers of Express. As expected, there's good support for Express testing. Further info,

- https://github.com/visionmedia/supertest/

```
npm install supertest --save-dev
```

Once installed, we can now create a `server.test.js` file for Supertest in the same directory as `server.js`.

We'll then need to require supertest in this test file,

```
const request = require('supertest');
```

and create a new export for the server app in `server.js`,

```
module.exports.app = app;
```

This exports means we can now access the server from other files, including any test files, e.g.

```
// require exported server app from server.js
var app = require('./server').app;
```

testing - test http request from server.js

We can use the Supertest library to test a basic http request from our server, e.g.

```
it('should return a http response from server...', (done) => {
  request(app) // pass server app to supertest
    .get('/') // make http request - e.g. get, put, post or delete
    .expect('test http server...') // assertion - expected return
response, e.g. a string from the server...
    .end(done); // call supertest's `end()` method passing done from
Mocha...done is handled by supertest
});
```

As we're using Mocha with an async request, we still need to pass `done` to inform Mocha that it needs to wait for the response.

Then, we call the Supertest library, passing the required server, `app`. Then, we can call any of the required http requests, such as `get`, `post`, `put` or `delete`, passing the route to call for testing.

The assertion will checking the expected return response from the http query, in this example a simple string response `test http server...`.

To finish this test, we then call supertest's `end()` method, and pass `done` from Mocha. Supertest will handle the actual closing and finishing of the test.

We can now run this test,

```
npm run test-mon
```

which will test our previous utilities tests, and our new http requests from the server. The testing responses from Mocha will also include some useful timings for these tests in ms.

We can then add further test assertions, such as a check to the server for the return status code,

```
it('should return a http response from server...', (done) => {
  request(app) // pass server app to supertest
    .get('/') // make http request - e.g. get, put, post or delete
    .expect('test http server...') // assertion - expected return
response, e.g. a string from the server...
    .expect(200) // checl return http status code
    .end(done); // call supertest's `end()` method passing done from
Mocha...done is handled by supertest
});
```

So, we can use Supertest to test many different assertions for http request responses, status codes, return bodies (e.g. JSON or not? &c.). We might use this type of testing to check assertions for return objects, perhaps from JSON-based API streams.