Unit Testing in Typescript/NodeJS

Unit testing is a vital part of writing stable backend software. It's even more important in a dynamic language environment like NodeJS, as it doesn't have a strict type system to catch a lot of seemingly simple errors.

A number of different testing frameworks exist, and each have their own quirks.

Basic Structure of Jest Tests

Jest is a test and mocking framework built in and for NodeJS. Typescript compatibility has been built over the top of it, but a lot of the legacy documentation has been written for older Js in mind first.

The basic intent, as with all unit testing, is to take some public element of your codebase (a unit if you will), and ensure that it works the way you expect under all input conditions. This can help us detect the cause of bugs early while building, and when a change has created a problem later on.

Jest has a series of different key functions we can use to organise and run our tests. Here's the main few:

```
describe('group description', () => {});
// Describe is used to group logically similar test cases. You can
nest these as many times as needed. Expect() statements do not go
into describe() blocks directly.

it('test description', () => {});
test('test description', () => {});
// It and Test are aliases of the same thing. No preference for
either, apart from how you phrase your test descriptions.
// it('Returns a positive result when adding 2 positive numbers', ()
=> {})
// is preferred over
// test('If it returns a positibe result when adding 2 positive
numbers', () => {});
// As it's more obvious what the test is expecting when it fails.

expect();
// expect is the assertion clause that Jest uses. It has 2 basic
forms:
// 1: Matching a result with an expectation:
expect(myFunction('Hello', 'World!')).toBe('Hello World!');
// 2: Checking for an error
expect(() => { myFunction('Hellow', 'World') }).not.toThowError();
```

When put together, normal, synchronous testing might have a structure like this:

```
describe('testModule/Class', () => {
  describe('public function being tested', () => {
    it('specific behaviour being tested 1 - Eg "Adding 2 positive
integers gives the positive sum"', () => {
      expect(add(2,2)).toBe(4);
    });

    it('specific behaviour being tested 2 - Eg "Adding a positive and
negative integer gives the signed difference between their absolute
values"', () => {
      expect(add(2,-2)).toBe(0);
    });

    it('specific behaviour being tested 3 - Eg "Adding two negaive
integers gives the negative sum"', () => {
      expect(add(-2, -2)).toBe(-4);
    });
  });
});
```

Generally speaking, less assertions per-test (ie, per `it()` statement) will give more reliable results. Jest has been known to ignore failing `expect()` statements if several (>5 maybe?) are stacked.

Writing good unit tests with descriptive labels also has the advantage of acting as documentation for a specific service. Comments and requirements docs may fail to be updated at some point, but tests must pass to be deployed, so it is helpful to write more, smaller tests.

A more explicit form of Arrange/Act/Assert might look like so:

```
describe('testModule/Class', () => {
  describe('public function being tested', () => {
    it('specific behaviour being tested 1 - Eg "Adding 2 positive
integers gives the positive sum"', () => {
      const inputDataA = 2;
      const inputDataB = 2;
      const expectedOutputData = 4;

      const actualResult = add(inputDataA, inputDataB);

      expect(actualResult).toBe(expectedOutputData);

    });
  });
});
```

Obviously this is a fairly contrived example, but your data may be a lot more complex - ie data produced by faker or a very long string.

Jest Matchers

Basic Matchers

Jest has a fairly extensive list of basic matchers, found here: Expect · Jest (jestjs.io)

A few main tripping points that you'll probably forget:

```
expect().toBe() // Checks if 2 objects are the same *instance*
expect().toEqual() // Checks if 2 objects have equal properties
(preferable I think)

expect().toBeCloseTo() // use this for floating-point equality
testing.
```

More advanced matchers

In some cases, we may want to use more advanced matching techniques when asserting certain responses.

For example, it may be beneficial to ignore the exact timestamp generated by a service response, but to simply assert that there is one.

We can used more advanced matchers (detailed here: The hidden power of Jest matchers | by Boris | Medium) to test the interesting elements of the response.

Examples:

```
// Expecting a date, but without any care for the contents
const comment = createComment('test content', 'author@me.com');
expect(comment).toEqual({
  createdAt: expect.any(Date),
  content: 'test content',
  author: 'author@me.com'
});
```

```
// Partial match an object on specific keys
const user = prepareUserInfo('test-user');
// user object is something like ...
// {
//   id: 123,
//   name: 'test-user',
//   profile: {...},
//   passwordHash: '*****',
//   whatever: {...}
// }
// ... but for the test we are interested only in name and id
expect(user).toEqual(expect.objectContaining({
  id: 123,
  name: 'test-user'
}));
```

```
// Checking a mocked service was called, formatting an input argument
exactly.
test('Sends an email with all the listed items present, separated by
newlines', async () => {
    const sendMail = jest.fn();
    const testLines = [
        `This is item 1`,
        `This is another item`,
        `Oh wow, yet another item!`,
    ];
    await manage.sendLinesEmail(testLines);

    expect(sendMail).toHaveBeenCalledTimes(1);
    expect(sendMail).toHaveBeenCalledWith({
        from: expect.any(string),
        to: expect.any(string),
        subject: expect.stringContaining('Preconfigured subject
line!'),
        text: expect.stringContaining(`\n${testLines.join('\n')}`),
    });
});
```

Repeated Tests with `it.each()`

Sometimes you will have a set of tests that all look very oddly similar, but use ever so slightly different data and results. Not to worry, we can write the test code once and insert our own data layout using the `it.each()` function. This will also let us use the values in the test name.

Let's say we are testing the following function, since it's fairly simple.

```
// src/palindrome.ts
function isPalindrome(word: string): boolean {
    return word.toLowerCase() ===
word.toLowerCase().split('').reverse().join('');
}
```

Form 0: Individual Tests

The default way to write a test suite is to use individual `it()` statements for each test.

```
// test/palindrome.test.ts
describe('isPalindrome - Form 0', () => {
    it('isPalindrome("Racecar") => true', () => {
        expect(isPalindrome('Racecar')).toEqual(true);
    });
    it('isPalindrome("Typewriter") => false', () => {
        expect(isPalindrome('Typewriter')).toEqual(false);
    });
    it('isPalindrome("rotor") =>', () => {
        expect(isPalindrome('rotor')).toEqual(true);
    });
});
```

This certainly works, but it violates the `DRY` principle - Don't Repeat Yourself. We can do better!

Form 1: Input data as an array of Arrays

You can specify a test data suite as an "Array of Arrays" if you like, where each inner array is equal to one test data set. Example:

```
// test/palindrome.test.ts
describe('isPalindrome - Form 1', () => {
    it.each([
        ['Racecar', true],
        ['TypeWriter', false],
        ['rotor', true],
    ])('isPalindrome("%s") => %s', (testWord, expectedResult) => {
        expect(isPalindrome(testWord)).toEqual(expectedResult);
    });
});
```

This gives us a few nice qualities, and has some downsides.

Firstly, we write the test code once, and yet it runs multiple times!

We also can inject the parameter values from the test into the test name (using `%s` formatting)!

It also means that adding more test cases is as easy as adding a new line to the array.

Unfortunately, it has the downside of not really labelling the parameters. If you're not so familiar with JS, it might not be obvious that `'Racecar'` is being inserted as `testWord`, but that's not a huge deal.

We do have another option:

Form 2: Input Data as Tagged Template Literals

Tagged Template literals is a concept in JS of using the backtick characters (`` ` ``) to format strings.

In this case we get to create a testing table as below:

```
// test/palindrome.test.ts
describe('isPalindrome - Form 2', () => {
    it.each`
        testWord        | expectedResult
        ${'Racecar'}    | ${true}
        ${'TypeWriter'} | ${false}
        ${'rotor'}      | ${true}
    `('isPalindrome("$testWord") => $expectedResult', ({ testWord,
expectedResult }) => {
        expect(isPalindrome(testWord)).toEqual(expectedResult);
    });
});
```

Note the differences between this and `Form 1`

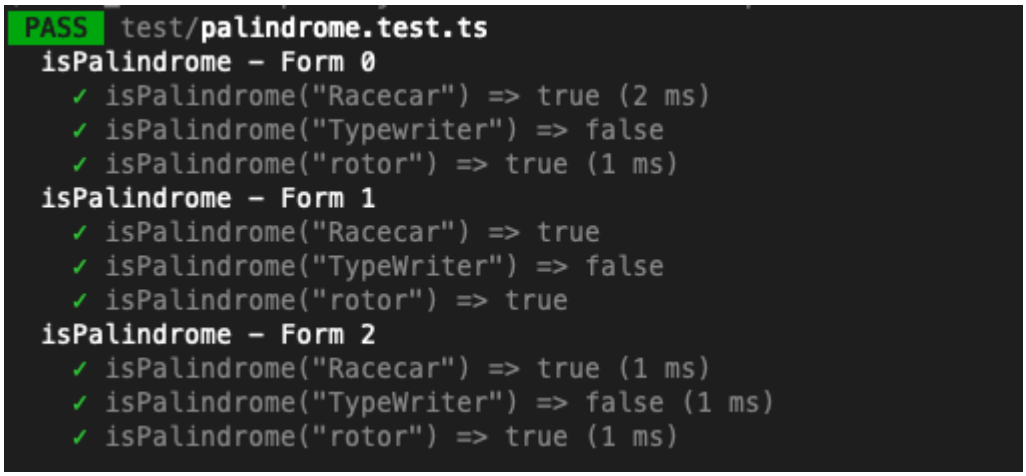
- `it.each([])('', ()={})` is now called like `it.each``('', () => {})`
- The parameters in the test name are now labelled, instead of using `%s`
- The arguments to the test are now using `({ testWord, expectedResult }) => {}` instead of `(testWord, expectedResult) => {}`
- The columns have labels!
- Test values are wrapped in `${}` . This is required, even for string arguments.

If you're using ES Lint and Prettier, it will even clean up the table formatting for you, so all of your columns are well aligned!

The final advantage is that the column order doesn't have to follow the argument order in the test parameters - the column names just have to match. ie: the following will also work!

```
describe('isPalindrome - Form 2', () => {
    it.each`
        expectedResult | testWord
        ${true}        | ${'Racecar'}
        ${false}       | ${'TypeWriter'}
        ${true}        | ${'rotor'}
    `('isPalindrome("$testWord") => $expectedResult', ({ testWord,
expectedResult }) => {
        expect(isPalindrome(testWord)).toEqual(expectedResult);
    });
});
```

In any case, the results will give you tests that print out like this when you run them:



Using `it.each()` may not be applicable for all test input data, for example if sometimes the test should throw an `Error`.

Testing Services with Jest

Mock vs Stub vs Spy

Most testing reference documents will discuss some combination of Mocking, Stubbing and Spying as a tool for testing services. However, terminology has become muddled as more and more digital ink has been spilled, and frameworks have been introduced.

Let's briefly define the above terms.

Mocking

Mocking is when you replace the implementation of a function, service, or other coroutine with a no-op operation, and return null or 0. Often you will use this to remove some external dependency like a HTTP call, or a send to a logging framework. It can also be used to replace some expensive call with a no-op.

Technically speaking, mocks don't do any calculation or return any real implementation/data. They're the dumbest possible unit, and we use them to replace side-effects.

Stubbing

Stubbing is like mocking, but your replacement function might have a tiny amount of brains to it, and might return actual, useful data. It simulates some behaviour.

This article will use Mock and Stub interchangeably, as `jest` doesn't draw any real distinction between them (apart from mocks are not given implementations by default).

Spying

Spying, unlike mocking or stubbing, doesn't replace the default implementation of a piece of code (at least, unless you tell it to). Generally you would use spying to check that a method is being called with the right parameters, and the correct number of times.

A Mock/Stub Example

Why would you want to mock or stub a function? In essence it comes down to the structure of your unit test.

Typically, each test should be looking to verify one specific piece of behaviour, eg: `The SMS Service sends one SMS to each unique number in the recipient list`.

This description implies 2 things:

- The service is responsible for de-duplicating the recipient list
- This service needs to call its SMS Provider once for each list item after de-duplication.

Realistically, we don't care who the SMS provider is, or how they are told to send an SMS, those can be handled downstream or in integration testing.

We probably don't want to be sending real SMSes out in this case though! So let's mock/stub the SMS Provider.

We may or may not care, in this tests case, what the response from the SMS Provider call is. If we do, we would stub the function to return some reasonable response, and if we don't, a mock is sufficient.

It can also be used to stand in for long running compute functions that we don't necessarily want to run every time we run `yarn test`.

For example, if we have a file like this:

```
// /app/service/send-sms.ts
import { sendSms } from 'my_sms_library';

/**
 * Returns true when we successfully send an sms message
 */
export function sendSmsFromMe(destNumber: string, message: string):
boolean {
  // Maybe do some validation on the destinationNumber

  try {
    const smsResponse = sendSms({
      sendingNumber: 'myNumber',
      destinationNumber: destNumber,
      message: message,
      apiKey: '<some precofigured thing>'
    });

    if(smsResponse.code === 200) {
      return true;
    }
  } catch {
    return false;
  }
  return false;
}
```

We may want to pretend to call `sendSms` instead of actually calling it (so we don't send a real sms out or spam the sms provider with junk).

However, Good Unit Testing principles dictate that we should make sure that `sendSmsFromMe` returns the correct response when `sendSms` gives us an error or something, so let's just pretend we own `sendSms` instead.

Below is the code for the minimal mock of the function where we tell it what we expect (I haven't tested this though whoops):

```
// /test/service/send-sms.test.ts
import { sendSms } from 'my_sms_library';
import { sendSmsFromMe } from '/app/service/send-sms.ts';

jest.mock('my_sms_library');
// You can also specify the mock type, but I haven't here.
const mockedSendSms = sendSms as jest.Mock;

describe('send-sms', () => {
  describe('sendSmsFromMe', () => {

    afterEach(() => {
      // Remove any lingering metadata about the mock
      mockedSendSms.clearMock();
    });

    it('Returns true when the service returns a 200 code', () => {
      mockedSendSms.mockResolvedValue({ code: 200, messageId:
'123445315' });
      expect(sendSmsFromMe('<my number>', 'Hello
stranger!')).toBe(true);
    });

    it('Returns false when the service returns something other than a
200 code', () => {
      mockedSendSms.mockResolvedValue({ code: 400, error: `That
didn't work!` });
      expect(sendSmsFromMe('<my number>', 'Hello
stranger!')).toBe(false);
    });

    it('Returns false when the service throws an error', () => {
      mockedSendSms.mockImplementation(() => {throw new Error(`Hey! I
don't like that!`)});
      expect(sendSmsFromMe('<my number>', 'Hello
stranger!')).toBe(false);
    });
  });
});
```

We can also interrogate the mock to see how many times it was called, and with what arguments.

It'd not be good for say, an SMS Api to accidentally get called multiple times to send 1 SMS, so this can be helpful to test.

```
// /test/service/send-sms.test.ts
import { sendSms } from 'my_sms_library';
import { sendSmsFromMe } from '/app/service/send-sms.ts';

jest.mock('my_sms_library');
// You can also specify the mock type, but I haven't here.
const mockedSendSms = sendSms as jest.Mock;

describe('send-sms', () => {
  describe('sendSmsFromMe', () => {

    afterEach(() => {
      // Remove any lingering metadata about the mock
      mockedSendSms.clearMock();
    });

    it('Returns true when the service returns a 200 code', () => {
      mockedSendSms.mockResolvedValue({ code: 200, messageId:
'123445315' });
      expect(sendSmsFromMe('<my number>', 'Hello
stranger!')).toBe(true);
    });

    it('Only calls the Sms Library once per sms request', () => {
      mockedSendSms.mockResolvedValue({ code: 200, messageId:
'123445315' });
      sendSmsFromMe('<my number>', 'Hello stranger!');
      expect(mockedSendSms).toBeCalledTimes(1);
    });

    // You'd probably pre-configure these in some kind of config
file, and you might keep a dummy copy in your repo for this
particular test.
    it('Adds the api key and my number as the source!', () => {
      mockedSendSms.mockResolvedValue({ code: 200, messageId:
'123445315' });
      sendSmsFromMe('<my number>', 'Hello stranger!');
      expect(mockedSendSms).toBeCalledWith({
        sendingNumber: 'myNumber',
        destinationNumber: '<my number>',
        message: 'Hello stranger!',
        apiKey: '<some precofigured thing>'
      });
    });

    // ... other tests redacted
```

```
  });
});
```

Don't forget you can use the scope of your containing block to reduce the number of strings you have to retype, or the number of similar mocks you need to set up!

Mocking Singletons with Jest

Singletons are a common design pattern in a reasonable amount of software. The typical case is that you have some configuration data that you want to use in your service that you don't want to retrieve or parse each time the service is created.

Unfortunately, since singletons give you a weird call pattern, mocking their functions is a bit of a pain.

One solution is to refactor the singleton service to be provided via dependency injection, but this does add some amount of structural overhead to your application, may require learning a framework, and otherwise requires justifying the work to actually implement.

We can, however, mock the singleton functions using Jest Spies.

Wait! I just said that spies don't change their target's behaviour!

Well... They can in Jest! Probably because they can in other mocking frameworks, and Jest thinks it wants to be the testing framework to end all the rest.

It's a good thing too, since the way that regular jest mocking works isn't really conducive to dealing with class instances.

Take for example the following service and consumer, and let's say we want to test the consumer works properly.

```
// src/service/db-service.ts
export class DbService {
    private static instance: DbService;
    public static getInstance(): DbService {
        if (this.instance == null) {
            this.instance = new DbService();
        }
        return this.instance;
    }

    private connectionString: string;

    private constructor() {
        if (!process.env.DB_CONNECTION_STRING) {
            throw new Error('process.env.DB_CONNECTION_STRING not
set');
        }
        this.connectionString = process.env.DB_CONNECTION_STRING;
    }

    /** Retrieves the record with the given ID from the database */
    async getRecord(recordId: string): Promise<{ id: string; value:
string }> {
        // Actual implementation removed because we don't care, let's
return some random string for now.
        return { id: recordId, value: 'Hello World!' };
    }
}
```

```
// src/service/email-service.ts

/** Retrieves a given record from the DB and renders it as HTML */
export async function renderRecordAsHtml(recordId: string):
Promise<string> {
    try {
        const record = await
DbService.getInstance().getRecord(recordId);
        return `<!Doctype html><html><body><h1>${record.id}</h1>
<p>${record.value}</p></body></html>`;
    } catch (e) {
        throw new Error('Something went horribly wrong');
    }
}
```

```
// test/service/email-service.ts

describe('renderRecordAsHtml', () => {
    const mockRecordValue1 = 'Our own special record value!';
    let dbServiceInstance: DbService;
    let dbServiceSpy: jest.SpyInstance;

    beforeAll(() => {
        process.env.DB_CONNECTION_STRING = 'blah!';
        // Get the service *before* running the suite, as we need to
use the same class instance to mock it.
        dbServiceInstance = DbService.getInstance();

        // Now let's mock it
        dbServiceSpy = jest.spyOn(dbServiceInstance, 'getRecord');
        dbServiceSpy.mockImplementation((recordId: string) => {
            return { id: recordId, value: mockRecordValue1 };
        });
    });

    afterEach(() => {
        dbServiceSpy.mockClear();
    });

    afterAll(() => {
        delete process.env.DB_CONNECTION_STRING;
        jest.resetAllMocks();
    });

    // We use our default (set in beforeAll()) if we don't override
it
    it('Inserts the record ID into a H1 tag', async () => {
        const recordAsHtml = await renderRecordAsHtml('314');

expect(recordAsHtml).toEqual(expect.stringContaining(`<h1>314</h1>`));

    });

    it('Inserts the value inside a paragraph tag', async () => {
        const recordAsHtml = await renderRecordAsHtml('314');

expect(recordAsHtml).toEqual(expect.stringContaining(`<p>${mockRecordV
</p>`));
    });

    // Or we can override it for this specific test set!
    it('Inserts the record ID into a H1 tag after overriding the
```

```
implementation', async () => {
        const mockIdOnce = 'This is a fake ID!';
        const mockRecordOnce = 'This is our own record :p';
        dbServiceSpy.mockImplementationOnce(() => {
            return { id: mockIdOnce, value: mockRecordOnce };
        });

        const recordAsHtml = await renderRecordAsHtml('314');

expect(recordAsHtml).toEqual(expect.stringContaining(`<h1>${mockIdOnce
</h1>`));
    });

    // But this one still uses the default implementation!
    it('Inserts the value inside a paragraph tag', async () => {
        const recordAsHtml = await renderRecordAsHtml('314');

expect(recordAsHtml).toEqual(expect.stringContaining(`<p>${mockRecordV
</p>`));
    });

    // We can even make it throw an error!
    it('Throws ~Something went horribly wrong~ if the record is not
found', async () => {
        dbServiceSpy.mockImplementationOnce(() => {
            throw new Error('Heck!');
        });

        try {
            const recordAsHtml = await renderRecordAsHtml('314');
            fail('Expected to throw error "Something went horribly
wrong", but did not');
        } catch (e) {
            expect(e).not.toBeNull();
            expect(e).toEqual(new Error('Something went horribly
wrong'));
            expect(e).not.toEqual(new Error('Heck!'));
        }
    });
});
```

The above will correctly mock the implementation of the `DbService`, but we still need to know the implementation details of `DbService.getInstance()`.

In particular, if you are testing multiple functions inside `email-service.ts`, you will need to allow a singleton reset so you aren't sharing state between tests, or structure your tests such that all the mocking is done before all the suites run. This is pretty annoying IMO.

Spying on functions with Jest

Okay, so we can see the obvious use case for mocking - we don't want to call an AWS Service for example, but maybe we are testing something that handles some data it might return.

But when would we use spying? And how do we call it in Jest?

Generally, spying is used when we don't want to change the implementation of a class, but we do care about how it's being used.

Here's some generic examples where you'd maybe want to use a spy over a mock:

- I need to test my http client service is only sending one request when I call it, and is not meaningfully modifying the responses. I'd use a Spy + a tool like Nock to create generic test cases.

When testing the consumers of this service, I will write a stub for this service.

- I have some publisher/subscriber model, and I really need to make sure the subscriber is receiving notifications in specific circumstances (ie: I have a checkbox checked).

I don't want to mock this subscriber particularly, but I need to verify that the subscription and unsubscription is actually working as I intended.

Mocking RestAPIs with Nock

nock/nock: HTTP server mocking and expectations library for Node.js (github.com)

Nock is a framework that allows the tester to mock restAPI responses, in HTTP format even.

This allows us to test our code against diverse responses, without the need for an integration server.

It can also let us get a little creative with our validation code, and ensure that we are handling unexpected responses correctly.

For example, the `axios` library provides async http calls to other services, but itself as a service can be hard to mock correctly, as its error handling can be somewhat subtle.

`nock` allows us to verify our code against what axios would actually do in a specific reaponse case (say, a 404 error), as opposed to us guessing based off their documents.

Nock automatically converts header keys in requests to lower case. In most cases, API gateways should also be case insensitive, but be sure to check this before relying on Nock to test your request headers.

For example, the following header arrangements should be equivalent on most HTTP servers:

```
{
    "Authorization":"MyAuthKey",
    "accept":"application/json",
    "deviceId":"12345"
}

{
    "authorization":"MyAuthKey",
    "accept":"application/json",
    "deviceid":"12345"
}
```

However some specific server implementations may not be the case. Using Nock to test your header arrangements are cased correctly may not be a suitable test in this case.

The following describes a possible test pattern using Nock to control axios calls:

```
import nock from 'nock';
import * as AuthService from '/app/service/authentication-service';
import { CustomError } from '/app/service/error';
import axios from 'axios';

// Make sure Axios will work with Nock
axios.defaults.adapter = require('axios/lib/adapters/http');

describe('authenticate', () => {
    const authDomainRoot = 'https://example.com';
    const authUrlPath = '/auth';

    afterEach(() => {
        // Don't forget to reset any Nock mocks after each test
(since we are mocking per-test)
        nock.cleanAll();
    });

    it('Throws a specific error if the Auth API returns 400', async
() => {
        const scope =
nock(authDomainRoot).post(authUrlPath).reply(400, {});

        try {
            const response = await
AuthService.authenticate('thisIsAnAuthCode', 'thisIsNotAJwt');
            console.log(response);
            fail('Expected to throw CustomError but did not');
        } catch (e) {
            if (!(e instanceof CustomError)) {
                fail('Expected to throw CustomError, but threw
something else');
            } else {
                const specificCustomErr = new CustomError(400, '07',
`Something went wrong, please try again`);

expect(e.errorCode).toEqual(specificCustomErr.errorCode);
                expect(e.message).toEqual(specificCustomErr.message);

expect(e.httpCode).toEqual(specificCustomErr.httpCode);
            }
        }
    });

    it('Throws a specific error if the Auth API returns 500', async
() => {
        const scope =
```

```
nock(authDomainRoot).post(authUrlPath).reply(500, {});

        try {
            const response = await
AuthService.authenticate('thisIsAnAuthCode', 'thisIsNotAJwt');
            console.log(response);
            fail('Expected to throw CustomError but did not');
        } catch (e) {
            if (!(e instanceof CustomError)) {
                fail('Expected to throw CustomError, but threw
something else');
            } else {
                const specificCustomErr = new CustomError(500, '10',
`Something went wrong, please try again`);

expect(e.errorCode).toEqual(specificCustomErr.errorCode);
                expect(e.message).toEqual(specificCustomErr.message);

expect(e.httpCode).toEqual(specificCustomErr.httpCode);
            }
        }
    });

    it('Returns the same tokens provided by the Auth API', async ()
=> {
        const mockAccessToken = 'ThisIsAnAccessToken';
        const mockRefreshToken = 'ThisIsARefreshToken';
        const scope = nock(authDomainRoot)
            .post(authUrlPath)
            .reply(200, { access_token: mockAccessToken,
refresh_token: mockRefreshToken });

        const response = await
AuthService.authenticate('thisIsAnAuthCode', 'thisIsNotAJwt');
        expect(response.accessToken).toEqual(mockAccessToken);
        expect(response.refreshToken).toEqual(mockRefreshToken);
    });

    it('The provided authCode is present in the body', async () => {
        const mockAccessToken = 'ThisIsAnAccessToken';
        const mockRefreshToken = 'ThisIsARefreshToken';
        const mockAuthCode = 'thisIsAnAuthCode';
        let caughtBody;
        const scope = nock(authDomainRoot)
            .post(authUrlPath)
            .reply(200, function (uri, reqBody) {
                caughtBody = reqBody;
                return { access_token: mockAccessToken,
```

```
refresh_token: mockRefreshToken };
            });

        await AuthService.authenticate(mockAuthCode,
'thisIsNotAJwt');
        const bodyAsParams = new URLSearchParams(caughtBody);
        expect(bodyAsParams.get('code')).toEqual(mockAuthCode);

expect(bodyAsParams.get('grant_type')).toEqual('authorization_code');
    });

    it('The provided clientAssertion is present in the body', async
() => {
        const mockAccessToken = 'ThisIsAnAccessToken';
        const mockRefreshToken = 'ThisIsARefreshToken';
        const mockJwt = 'thisIsNotAJwt';
        let caughtBody;
        const scope = nock(authDomainRoot)
            .post(authUrlPath)
            .reply(200, function (uri, reqBody) {
                caughtBody = reqBody;
                return { access_token: mockAccessToken,
refresh_token: mockRefreshToken };
            });

        await AuthService.authenticate('thisIsAnAuthCode', mockJwt);
        const bodyAsParams = new URLSearchParams(caughtBody);

expect(bodyAsParams.get('client_assertion')).toEqual(mockJwt);
        expect(bodyAsParams.get('client_assertion_type')).toEqual(
            'urn:ietf:params:oauth:client-assertion-type:jwt-bearer'
        );
    });
});
```

Mocking AWS Services with `aws-sdk-mock`

`aws-sdk-mock` gives the user a little more control over mocking the Aws SDK. Specifically, it overrides specific functions of the SDK with your own implementation, preserving the type contract.

Note: `aws-sdk-mock` is built for the 'v2' version of the `aws-sdk`.
For v3 (ie the one that uses `@aws-sdk/client-`), use https://github.com/m-radzikowski/aws-sdk-client-mock

The standard pattern for mocking with `aws-sdk-mock` and `jest` looks like so:

```
import AWSMock from 'aws-sdk-mock';
import AWS from 'aws-sdk';
import { getFile } from 'app/service/s3-service'; // This is our
wrapper service for S3

import type { GetObjectOutput, GetObjectRequest } from 'aws-
sdk/clients/s3';

describe('s3-service', () => {
    describe('getFile', () => {
        // Create a Jest Mock Function that we can query about calls
later.
        let mockGetObject: jest.Mock<GetObjectOutput, [req:
GetObjectRequest]>;
        beforeAll(() => {
            // Write an implementation for our Jest Mock
            mockGetObject = jest.fn((req: GetObjectRequest):
GetObjectOutput => {
                return { Body: 'Test Body', VersionId: 'V1' };
            });


            // Note that we need to set the AWS SDK instance before
we try to mock it.
            AWSMock.setSDKInstance(AWS);
            // Overwrite S3.getObject() with AWSMock.
            // Note that we use the `callback()` function, leaving
the Error side undefined.
            AWSMock.mock('S3', 'getObject', (params, callback) => {
                callback(undefined, mockGetObject(params));
            });
        });

        afterAll(() => {
            // Restore S3's normal functionality
            AWSMock.restore('S3', 'getObject');
            // Clear out the Jest Mock as well, just to be safe
            jest.resetAllMocks();
        });

        afterEach(() => {
            // Clear data from the mock, ie how many times it was
called, etc
            mockGetObject.mockClear();
        });

        // Test that our wrapper is returning what we told it to from
```

```
the mock
        it('Returns the requested file if the file exists', async ()
=> {
            const awsResp = await getFile('testBucket',
'testFolder/testFile.txt');

            expect(awsResp).toEqual({ Body: 'Test Body', VersionId:
'V1' });
        });

        // Test that we aren't calling S3 more than we have to
        it('Calls the getObject function once per request', async ()
=> {
            await getFile('testBucket', 'testFolder/testFile.txt');

            expect(mockGetObject).toHaveBeenCalledTimes(1);
        });
    });
});
```

This pattern allows us to modify the return of an AWS SDK function individually, but also use a standard return in most cases.

Note that the AWS SDK class and method are described by strings, which are equivalent to the name of the function you are overriding in your code. `aws-sdk-mock` should warn you if you are overriding it incorrectly.

Mocking AWS Errors without `aws-sdk-mock`

When testing unit behaviour in the event of an error, it can be helpful to mock when AWS returns some kind of error. The `aws-sdk` typically throws an `AWSError` object when it fails (as a promise rejection).

Unfortunately, this data type is not something `jest` will typically handle in its `expect().toThrow()` matching hook.

This pattern is known to work if there is a wrapper typescript service around the AWS SDK service (in the below example as `/app/service/s3-service.ts`), but could potentially work with `aws-sdk-mock` as well.

We can use the following pattern to mock an error response. This sample uses S3.

```
// File: /app/service/s3-service.ts
import { S3 } from 'aws-sdk';

/**
 * Retrieves a file from an S3 Bucket as an S3 Object
 * @param Bucket The S3 bucket to fetch from
 * @param Key The name + path of the file to fetch
 * @returns The file as an S3 Object Promise
 */
export const getFile = async (Bucket: string, Key: string) =>
    new S3()
        .getObject({
            Bucket,
            Key,
        })
        .promise();
```

```ts
// File: /app/service/s3-wrapper-service.ts
import * as s3 from './s3-service';
import type { PromiseResult } from 'aws-sdk/lib/request';
import type { AWSError, S3 } from 'aws-sdk';

/**
 * Fetches the contents of a file from an S3 bucket as a string.
Returns an empty string if the file doesn't exist. Throws an {@link
AWSError} on any other failure.
 * @param bucketName The name of the S3 bucket to read from
 * @param key The file name + path to read.
 * @returns The body of the file as a string. If the file doesn't
exist, returns an empty string. If there is another error, throws it.
 */
export async function fetchFromS3(bucketName: string, key: string):
Promise<string> {
    let file: PromiseResult<S3.GetObjectOutput, AWSError>;

    try {
        file = await s3.getFile(bucketName, key);
    } catch (e) {
        const error = e as AWSError;

        // If no files exists - just treat it as an empty file
        if (error.code === 'NoSuchKey') {
            return '';
        }
        throw e;
    }
    if (!file.Body) {
        // If the file doesn't exist - just treat it as an empty one
        return '';
    }
    return file.Body.toString();
}
```

```
// File: /test/s3-wrapper-service.test.ts
import * as s3 from 'app/service/s3-service';
import { fetchFromS3 } from 'app/service/s3-wrapper-service';
import type { AWSError } from 'aws-sdk';

jest.mock('app/service/s3-service');

describe('fetchFromS3', () => {
    it('Returns an empty string if S3 returned `NoSuchKey` as an
error', async () => {
        const noSuchKeyError: AWSError = {
            code: 'NoSuchKey',
            message: 'NoSuchKey error',
            name: 'No Such Key error or something',
            time: new Date(),
        };
        (s3.getFile as jest.Mock).mockRejectedValue(noSuchKeyError);
        const result = await
fetchFromS3(process.env.DH_MIRROR_BUCKET!, 'testfile.csv');
        expect(result).toEqual('');
    });

    it("Throws S3 errors that aren't `NoSuchKey`", async () => {
      const otherError: AWSError = {
          code: 'ServiceUnavailable',
          message: 'need more kfc 21 piece buckets',
          name: 'ServiceUnavailable',
          time: new Date(),
      };
      (s3.getFile as jest.Mock).mockRejectedValue(otherError);

      try {
          await fetchFromS3('test-bucket', 'current.json');
          fail('Expected to throw but did not');
      } catch (e) {
          expect(s3.getFile).toBeCalledWith('test-bucket',
'current.json');
          expect(e).toBe(otherError);
      }
    });
});
```

Note the `try/fail/catch` block in the second `it()` statement.

Mocking AWS PromiseResult

Occasionally, there is the use case in the AWS SDK to inspect the raw `$response` object that AWS returns inside some SDK Result. This contains errors and the raw HTTP response object.

Unfortunately, doing this in application code means that you also need to mock it in your AWS Mocks.

Below is an example of this working correctly for KMS:

```ts
// /test/mocks/aws-result.ts
import AWS from 'aws-sdk';
import { PromiseResult } from 'aws-sdk/lib/request';

/** This produces a valid AWS PromiseResult that a mock funtion could
return */
export function awsSuccessPromiseResult<T>(resp: T): PromiseResult<T,
AWS.AWSError> {
    return {
        ...resp,
        $response: {
            data: resp,
            hasNextPage: () => {
                return false;
            },
            requestId: '',
            redirectCount: 0,
            retryCount: 0,
            nextPage: () => {},
            error: undefined,
            httpResponse: {
                body: '',
                headers: {},
                statusCode: 200,
                statusMessage: 'OK',
                streaming: false,
                createUnbufferedStream: () => {
                    return {};
                },
            },
        },
    };
}

/** This produces a valid AWS Error PromiseResult response */
export function awsErrorPromiseResult(): PromiseResult<any,
AWS.AWSError> {
    const otherError: AWS.AWSError = {
        code: 'ServiceUnavailable',
        message: 'need more kfc 21 piece buckets',
        name: 'ServiceUnavailable',
        time: new Date(),
    };
    return {
        $response: {
            data: undefined,
            hasNextPage: () => {
```

```
                return false;
            },
            requestId: '',
            redirectCount: 0,
            retryCount: 0,
            nextPage: () => {},
            error: otherError,
            httpResponse: {
                body: '',
                headers: {},
                statusCode: 200,
                statusMessage: 'OK',
                streaming: false,
                createUnbufferedStream: () => {
                    return {};
                },
            },
        },
    };
}
```

```
// /test/jwt-service.test.ts
import jwt from 'jsonwebtoken';
import AWSMock from 'aws-sdk-mock';
import AWS from 'aws-sdk';
import KMS from 'aws-sdk/clients/kms';
import { PromiseResult } from 'aws-sdk/lib/request';
import { awsSuccessPromiseResult } from './mocks/aws-results';

describe('JwtService', () => {
    let mockSign;
    beforeAll(() => {
        mockSign = jest.fn((req: KMS.SignRequest):
PromiseResult<KMS.SignResponse, AWS.AWSError> => {
            const resp = {
                KeyId: req.KeyId,
                SigningAlgorithm: req.SigningAlgorithm,
                Signature:
'ThisSignatureVerifiesThatTheJwtIsToooootallyLegit',
            };
            return awsSuccessPromiseResult(resp);
        });

        // Overwriting KMS.sign()
        AWSMock.setSDKInstance(AWS);
        AWSMock.mock('KMS', 'sign', (params, callback) => {
            callback(undefined, mockSign(params));
        });
    });

    afterAll(() => {
        // Restore KMS
        AWSMock.restore('KMS', 'sign');
        jest.resetAllMocks();
    });

    describe('signJwt', () => {
        afterEach(() => {
            mockSign.mockClear();
        });

        it('Mocks the signature correctly', async () => {
            const response = await new KMS().sign({
                Message: '{"recordA":"A very transgender value", ...
<other JWT stuff>}',
                KeyId: '99999',
                SigningAlgorithm: 'RSASSA_PKCS1_V1_5_SHA_256',
                MessageType: 'RAW',
```

```
            })
            .promise();
            console.log(response);
            console.log(JSON.stringify(jwt.decode(response)));
            expect(true).toBeTruthy();
        });
    });
});
```

Using Jest with VSCode

The main Jest plugin for VSCode is here: Jest - Visual Studio Marketplace

There is a config setting to turn off running tests automatically on save - add the following line to your `settings.json` file:

```
"jest.autoRun": "off",
```

The plugin is known to not work so well with yarn workspaces. In this case, it may be easier to simply use the CLI commands:

Jest CLI Options · Jest (jestjs.io)

Hacks

Boolean logic for Expect statements

We can write a fairly comprehensive test by using multiple `expect` statements in one test. ie:

```
it('Does 2 things', () => {
  expect(thingOne).toBeTruthy();
  expect(thingTwo).toBeTruthy();
});
```

This gives us the equivalent of a logical AND statement - if anything fails, the whole test fails.

Jest, however, doesn't support logical OR on first inspection, ie:

```
it('Does one of 2 things', () => {
  expect(result).toBe(thingOne jest.or thingTwo);
});
```

This makes sense somewhat, a unit really shouldn't have 2 equally valid outputs for 1 input.

However occasionally we might test something where it is helpful - like testing a mock where the call order doesn't matter.

In this case we can wrap the `expect` in a `try/catch` , and it will take either option. Example:

```
it('Calls the mock service with one of 2 arg sets', () => {
  try{
    expect(mockService).toBeCalledWith({ argA: 'myArgA', argB:
'myArgB' });
  } catch {
    expect(mockService).toBeCalledWith({ argA: 'yourArgA', argB:
'yourArgB' });
  }
});
```

In this case, only 1 of the expect statements is required for the test to pass.

Samples

Expressiveness

Here is a very expressive sample for a date/time validation function.

Using such detailed samples allows us to find the exact places and edge cases that a service might fail if adjusted.

In reality, I'd probably use an `it.each()` expression to write this, but it gives us a very well-described function anyhow.

```
describe('validateDateTimeString', () => {
    describe('Date strings', () => {
        it('Accepts a well formatted string', () => {
            expect(validateDateTimeString('01/01/2021',
'dd/MM/yyyy')).toBeTruthy();
        });

        it("Rejects a string that requires leading zeroes, but
doesn't supply them", () => {
            expect(validateDateTimeString('1/1/2021',
'dd/MM/yyyy')).toBeFalsy();
        });

        it('Rejects a date where the year position is not where it is
expected', () => {
            expect(validateDateTimeString('2021/01/01',
'dd/MM/yyyy')).toBeFalsy();
        });

        it('Rejects a date where the day value is larger than can be
in any month', () => {
            expect(validateDateTimeString('90/01/2021',
'dd/MM/yyyy')).toBeFalsy();
        });

        it('Rejects a date where the month value is larger than 12',
() => {
            expect(validateDateTimeString('01/90/2021',
'dd/MM/yyyy')).toBeFalsy();
        });

        it('Rejects a date where the year value is longer than the
required four digits', () => {
            expect(validateDateTimeString('01/01/12021',
'dd/MM/yyyy')).toBeFalsy();
        });

        it('Rejects a date where the day value (40) is larger than
can be for a given month', () => {
            expect(validateDateTimeString('40/08/2021',
'dd/MM/yyyy')).toBeFalsy();
        });

        it('Rejects a date where the day value is in the months
column (ie - American formatting)', () => {
            expect(validateDateTimeString('04/30/2021',
'dd/MM/yyyy')).toBeFalsy();
```

```
        });

        it('Accepts a date where the day value (30) is within the
given month (April)', () => {
            expect(validateDateTimeString('30/04/2021',
'dd/MM/yyyy')).toBeTruthy();
        });

        it('Rejects a date where the day value (31) is larger than
can be for a given month (April)', () => {
            expect(validateDateTimeString('31/04/2021',
'dd/MM/yyyy')).toBeFalsy();
        });

        it('Accepts a date where the day value (31) is within the
given month (January)', () => {
            expect(validateDateTimeString('31/01/2021',
'dd/MM/yyyy')).toBeTruthy();
        });

        it('Rejects a date where the day value (32) is larger than
can be for a given month (January)', () => {
            expect(validateDateTimeString('32/01/2021',
'dd/MM/yyyy')).toBeFalsy();
        });

        it('Accepts a date where the day value (28) exists for the
given February (2022)', () => {
            expect(validateDateTimeString('28/02/2022',
'dd/MM/yyyy')).toBeTruthy();
        });

        it('Rejects a date where the day value (29) does not exist
for the given February (2022)', () => {
            expect(validateDateTimeString('29/02/2022',
'dd/MM/yyyy')).toBeFalsy();
        });

        it('Accepts a date where the day value (29) exists for the
given February (2024)', () => {
            expect(validateDateTimeString('29/02/2024',
'dd/MM/yyyy')).toBeTruthy();
        });

        it('Rejects a date where the day value (30) does not exist
for the given February (2024)', () => {
            expect(validateDateTimeString('30/02/2024',
'dd/MM/yyyy')).toBeFalsy();
```

```
        });

        it('Accepts a date where the day value (28) exists for the
given February (2100)', () => {
            expect(validateDateTimeString('28/02/2100',
'dd/MM/yyyy')).toBeTruthy();
        });

        it('Rejects a date where the day value (29) does not exist
for the given February (2100)', () => {
            expect(validateDateTimeString('29/02/2100',
'dd/MM/yyyy')).toBeFalsy();
        });

        it('Rejects a date that is given in English instead of the
required date string', () => {
            expect(validateDateTimeString('1st of January 2021',
'dd/MM/yyyy')).toBeFalsy();
        });
    });

    describe('Time strings', () => {
        it('Accepts a time string that is formatted correctly for the
given format', () => {
            expect(validateDateTimeString('01:01:01',
'HH:mm:ss')).toBeTruthy();
        });

        it('Rejects a time with an hour value that is too high', ()
=> {
            expect(validateDateTimeString('60:01:01',
'HH:mm:ss')).toBeFalsy();
        });

        it('Rejects a time with a minute value that is too high', ()
=> {
            expect(validateDateTimeString('01:90:01',
'HH:mm:ss')).toBeFalsy();
        });

        it('Rejects a time with a second value that is too high', ()
=> {
            expect(validateDateTimeString('01:01:90',
'HH:mm:ss')).toBeFalsy();
        });

        it('Rejects a time with a value written in english, instead
of the required date format', () => {
```

```
            expect(validateDateTimeString('midnight',
'HH:mm:ss')).toBeFalsy();
        });
    });
});
```