Jest (testing) (mocking specifically)

Wednesday, June 13, 2018 9:17 AM

Mocking files (reference)

The reference covers how to do this. The gotchas I've run into are as follows:

- I tried mocking "pixi.js" by creating __mocks__/pixi.js, but apparently it needed to be called __mocks__/pixi.js.js
- Let's say you have a module that you can't import because it will immediately try calling into something that's not allowed, e.g. pixi.js tries creating a white texture (which involves creating a canvas and getting its context), but that fails since there's no DOM. I couldn't write code like this in my __mocks__ folder:

  ```
  const mockedPixi = jest.genMockFromModule('pixi.js');
  module.exports = mockedPixi;
  ```

  I think that when you run into an issue like this, you've got to manually mock any functions that you're going to use (so in my case, I'd provide a module that has dummy functions for PIXI.Graphics and the likes). Alternatively, I could've installed canvas (which is what the error message suggests (Error: Not implemented: HTMLCanvasElement.prototype.getContext (without installing the canvas npm package))or used jest-canvas-mock.
- Mocked files are "immune" to "resetModules" (reference) since they'll just be required again from the mocked file.

Mocking a module that's used practically everywhere, e.g. a logger

I ran into a scenario where I had a Lerna repository that had many modules, and the tests from all of them needed to mock the logger so that it wasn't as noisy. I didn't want to pollute each test with specific code for this, so here's what I did instead:

Make the mocked module itself

In my case, I had a "logger" module, so I added a source file that looks like this:

```
const logger = require('./index');

logger.error = () => {};
logger.info = () => {};

module.exports = logger;
```

Make a jest.config.js to point to the mocked module (reference)

Note: in my case: the "src" folder gets published to the "lib" folder for modules, so that's why "lib" shows below:

```
module.exports = {
  moduleNameMapper: {
    logger: '<rootDir>/node_modules/@botland/logger/lib/mocklogger.js',
  },
};
```

That's it! The mock is automatic thanks to moduleNameMapper, and if you extend a base config, you only really have to define this mapping in one spot.

Making a new mocked function (that's not based on anything)

Just use "jest.fn(implementation)" (reference). Note that this is an alias for mockFn.mockImplementation.

HiDeoo: Note: you can also use jest.fn() with mockImplementationOnce, mockReturnValue[Once], etc. like you can do jest.fn().mockReturnValueOnce(3).mockReturnValueOnce(4).mockReturnValue(5), this would return 3 on first call, 4 on second and 5 after for all remaining calls.

Mocking dates

https://github.com/hustcc/jest-date-mock

Mocking exported functions

If the function is defined in a separate file from the dev code using it…

If you have something like this:

```
import depFunction from './foo';
function functionToTest() {
    depFunction(); // you want to mock this
}
```

In your test code

```
import * as depFunctionExports from './foo';

// In the test…
jest
  .spyOn(depFunctionExports, 'depFunction')
  .mockReturnValueOnce(5);
```

If the function is defined in the same file as the dev code using it…

Suppose you have a file like this:

```
function functionToTest(args) {
  functionToBeMocked(args); // imagine this is defined in this file too
}

module.exports = {
  functionToTest,
  functionToBeMocked, // exported just for the sake of testing
}
```

You want to test functionToTest while mocking functionToBeMocked. It is not possible to mock the functions from outside of this file in the way that you may want (look for "update 180204" on this page). You would either have to move the function to another file or do what's shown at the link:

```
function functionToTest(args) {
  lib.functionToBeMocked(args);
}

const lib = {
  functionToBeMocked // imagine this is defined in this file too
};

module.exports = {
  functionToTest,
  lib // exported just for the sake of testing
};
```

Mocking exported constants

For reference, here's the scenario that I have:

```
// file_to_be_tested.js
import * as constants from './constants';
export function functionToBeTested() {
  return constants.constantToBeMocked * 5;
}
```

I want to mock constantToBeMocked from my test code. Here's a way to do that which doesn't involve changing the code to be tested:

```
// In the test itself…
jest.mock('../src/constants', () => {
  const original = require.requireActual('../src/constants');

  return { ...original, constantToBeMocked: 4 };
});

// Need to require the function to be tested again. The reason it's
"require" and not "import" is so that it doesn't hoisted above where we
mock the file it depends on. The reason we need to re-require it in the
first place is so that the file to be tested pulls in the mocked value.
const { functionToBeTested } = require('../src/file_to_be_tested.js');
// Run tests on functionToBeTested


// Outside of the test…
beforeEach(() => {
  jest.resetModules();
});
```

Notes:

- You cannot use any out-of-scope values whatsoever in this. For example, I have the number 4 hard-coded above; I can't take the value from the line just before jest.mock since that's out-of-scope.
  - This means that modules like Lodash that you may want to use need to be required inside the mock function itself.
  - This also means that any transpiled code that requires out-of-scope code needs to be changed. For example, if you're not on a version of Node that allows the spread operator for objects, then it gets transpiled to use "_extend". To fix that, you either have to work around the transpilation by using pure ES5 (e.g. Object.assign in this case) or upgrade Node to use the >ES5 code natively (and then get rid of any Babel plug-ins trying to convert it).
  - If you want to use an out-of-scope value here, then you sort of need to reverse everything. For example, rather than say "const someVar = 5" outside of the scope and then mocking a constant to be "someVar", you first need to set the mocked constant to 5, then require the mocked constant just like the dev code will and set someVar based on the mocked constant's value. For example:

    ```
    jest.mock('../src/constants', () => {
      const original = require.requireActual('../src/constants');

      return { ...original, mockedConstant: 4 };
    });

    const ServerConstants = require('../src/constants');
    ```

```
            const someVar = ServerConstants.mockedConstant;
```

- Because of the hoisting mentioned above, you'll need to call "jest.resetModules();" in every one of your tests. This could be done in the "beforeEach". However, this will reset *all* local state of your modules. I ran into an issue originally where I imported my own custom logger, then called "logger.silenceAllLogs()", but then "resetModules" would make them noisy again. To fix *that*, I needed to mock the "logger" module entirely.
    - If you don't do this, then it doesn't matter whether you require or import at the top of the file; it's going to mess up some of your tests.

Specifically for Bot Land constants, here's how I mocked the constants since they're just one potential export from @botland/shared:

```
jest.mock('@botland/shared', () => {
  const original = require.requireActual('@botland/shared');

  return {
    ...original,
    constants: { ...original.constants, TOP_PLAYERS_TO_RANK: 4 },
  };
});
```

~~I found this on StackOverflow, and I suppose that could have worked if I mocked the whole file, but I didn't do that. Instead, I did the same thing that I'd do for mocked functions, which is to define a "lib" object:~~

```
~~// file_to_best_tested.js~~
~~import * as constants from './constants';~~
~~function functionToBeTested() {~~
~~  return lib.getConstantToBeMocked() * 5;~~
~~}~~
~~function getConstantToBeMocked() {~~
~~  return constants.constantToBeMocked;~~
~~}~~
~~export const lib = {~~
~~  getConstantToBeMocked,~~
~~};~~
```

~~Then, from the test code, I would do this:~~

```
~~import { lib as testFunctions } from '../src/functionToBeTested';~~

~~jest~~
~~  .spyOn(testFunctions, 'getConstantToBeMocked')~~
~~  .mockImplementationOnce(() => newValueOfConstant);~~
```

~~I don't really like this pattern because it results in every single constant that needs to be mocked being wrapped in a function for no reason.~~

Restoring a mock

If you mocked a function by using spyOn, then you can call "mockRestore", otherwise you have to handle restoring by yourself if you just assigned a "jest.fn()".

Ensuring middleware is called on a REST server

I had a situation like this:

```
// server.js
```

```
import {getUserMiddleware} from './middleware/getuser';

// This is where we set up routes
this.server.get(
  '/users',
  [getUserMiddleware],
  this.getUser.bind(this)
);

// middleware/getuser.js
export function getUserMiddleware(req, res, next) {
    // do something here that you don't want to be hit by tests
    next();
}
```

In this situation, I just wanted to make sure that "getUserMiddleware" gets hit, *not* that it does what it's supposed to. The reason for only checking *that* it gets hit is because I would have other tests ensure that the middleware works correctly, that way the unit test could focus on making sure getUser does the right thing.

Here are the steps I had to go through, and keep in mind that this is a combination of other notes in this OneNote:

1. Make sure that getUserMiddleware resides in its own file as shown above. If it doesn't, then relocate it to its own file.
2. In the test code, add this:

```
import * as getUserMiddlewareExports from './middleware/getuser';

beforeEach(() => {
  // This ensures that you don't have to do "mockImplementationOnce"
down below. If you don't want to explicitly state this in your test,
then you can modify jest.config.js to have restoreMocks: true and
clearMocks: true.
  jest.restoreAllMocks();
});

test('example call into REST API for middleware', async () => {
  const spy = jest
    .spyOn(getUserMiddlewareExports, 'getUser')
    .mockImplementation((req, res, next) => next());

  const request = supertest(yourServerHere);
  await request.get(`/users`).expect(200);
  expect(spy).toHaveBeenCalledTimes(1);
});
```

By specifying "mockImplementation", I make sure that the original function doesn't get called. This is really helpful when middleware would have called into the database or done something that's otherwise annoying to set up or takes a while to run.

Ensuring only a particular argument of spy/mock was specified

Let's say you have a function like this:

```
function respondFailure(res) {
    res.send(500, {error :'who cares'});
}
```

If you mock this function and only want to verify that 500 was specified rather than the specific error message, then you **cannot** do this:

```
expect(res.send).toHaveBeenCalledWith(500); // bad because it will fail
since the function was also called with an error
```

Instead, you must inspect the individual call:

```
const res = { send: jest.fn() };
respondFailure(res);

expect(res.send).toHaveBeenCalledTimes(1);
expect(res.send.mock.calls[0][0]).toBe(500); // examine the first
argument of the first call
```