Node Testing

Thursday, 22 February 2024 10:33 AM

Tests has to be in __tests__, which is as per the paypalize standard and their scripts are reading this.
And the file name should be name.test.js so it can be run using **npm run test**

The golden rule is Test the interface and not the implementation.

For unit test cases in server / client can be placed inside :

- Src
- Controller / screens

By default its in watch mode

- Run -> npm run test
- Press "w"
- It should list of options like a, f, q, P
- Press "P" for enabling only server or client or both

Writing test cases using jest: https://jestjs.io/docs/

- Npm install --save-dev jest
- Syntax:
  - Import {exported const} from …
  - Describe('suite name', () => {
  - Test('test name', () =>{
  - Expect(const).toEqual({success: true});
  - })
  - })
- For async and await function, we have to use async() and make it await.. E.g(let txns = await txnService.getTransactions('1754370741915445701');)
- Other than equal, we have toBe, toEqual, toNull, toMatch, toContain, toHaveLength found in the jest

Creating a custom jest matcher:

- expect.extend({
  toMatchTransaction(received, expected) {
  // put your custom logic here
  },
  })

  Returning pass or fail in return { pass: false, message: () => 'received is empty' }

Mocking:

Just has a function called jest.fn() which gives mockfunction for controller, instead of calling again the service layer, you can use this fn() and see if it have been called and whether it gives success response..

- Const jsonspy = jest.fn();
- Const dummydata = [1,2,3,4]
- courseService.getCourses = jsonspy.mockResolvedValue(dummydata);
- Const expected = {courses:dummydata};
- Await courseController({}, {json:jsonSpy})
- Expect(jsonSpy).toHaveBeenCalledWith(expected)

.MockResolvedValue for async MockReturnedValue for sync funtions to assign the dummy data as mock.

Timer Mocks :: If our service is having settimeout goals, where it takes 500ms or 2000ms. You can use faketimer instead of waiting for that time using
jest.useFakeTimers();

You can give time like jest.advanceTimersByTime(500);

For cleaning up the timer in the afterEach(), we have to specify as afterEach(() => jest.useRealTimers());

Jest.mock() is also a function available =? Its for creating mock for entire module
Jest.fn() => make a mock for a particular service

Mocking Modules :: For mocking a complete module, keep it inside __mocks__ location
We have service code calling some downstream services, so we mock the complete service core using this mock. -> https://github.paypal.com/GlobalTechEd/GlobalTechEd-Node/blob/master/080-testing-debugging/lab3/instructions.md

We have to use these 2 lines:

- Servicecore.mock.serviceStubs.bankreferenceserv = {
  - Request: jest.fn(() => Promise.resolve(dummyBody))
- `const calls = workdayMock.mock.calls;`

Const dummyBody = { body: {holiday: true}};
Servicecore.mock.serviceStubs.bankreferenceserv = {
Request: jest.fn(() => Promise.resolve(dummyBody))
}

Test('test name', asyn () =>{
Let req = {
Params: {date: '2020-01-01',
Locality: {country: 'US'}
}}
`res = { json: jest.fn() }`
})

Coverage of test cases: npm run test -- --coverage

Debugging :: On browser use this chrome://inspect => Open dedicated DevTools for Node -> sources -> workspace -> Add folder -> select the project controller location -> In the popup give Allow

Now you can place debug points in the chrome and do the debug.

If we need to do in vscode itself we need to add launch.json inside .vscode which need to be in root path of the project. If not present we can create that folder and keep that file

In file:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Attach",
      "port": 9229,
      "request": "attach",
      "skipFiles": [
        "<node_internals>/**"
      ],
```

```
      "type": "pwa-node"
    }
  ]
}
```

And save it.

In leftside panel in vscode there is run and debug will have top dropdown as Attach, click the playbutton the logs will show which application its listening and then you can place debug points.

If you want to disconnect the debugger then you can select disconnect button below Attach.

Functional Testing:

Nemo and Cypress are used for functional end to end testing.

Cypress is more easy to start than traditional Nemo. By default the paypalscript support Nemo, but if we need to change to Cypress we need to do some steps. All the test results are stored in go/console -> Test Data Platform.

Steps:

- npm install cypress --save-dev
- Running a functional test : npm run test:functional
- In paypal how to use cypress / nemo - cypress :
- https://github.paypal.com/pages/webplatformengineering/docs/guides/functional-testing/
- To open cypress and run in the UI :: Npx cypress open
- "test:functional": "cypress run",
- Apart from this we need to set cypress to use for FT and configuration for FT

Test case:

- Here we use it instead of test
- We can use cy. To call any functions for e,g:
  - Cy.login(uname, pwd)
  - Cy.visit('/')
  - Cy.contains('button').click();

Errors:

- If facing issue while running in chrome, then from top right corner we can switch from chrome to Electron

Creating Test env:

- From terminal
  - Install testenv
  - Testenv login
  - Have a testenv.yaml file with following configs:

```yaml
applications:
- name: gtekrakennodeweb
  version: gtekrakennodeweb-2.10.4_20230609090816802
  features:
   - name: l7routing
     configs:
     - name: router
       value: NATIVE
  instance_count: 1
backed_by: msmaster.qa.paypal.com
label: sugnanasekaran
operators:
- sugnanasekaran
```

  - testenv create 30d --env-name te-sugnanasekaran --spec-file