Team 9 - ERP Testing Plan

Version 1.0
02/03/2021

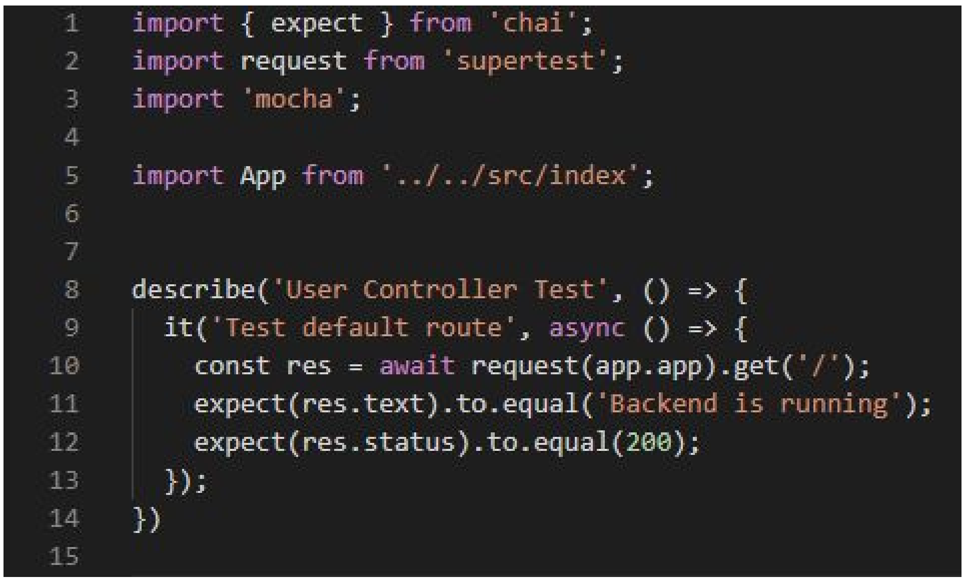
1. Testing Procedure

1.1 Unit Testing

Backend

For the backend, our team has decided to go with the Mocha Framework, which is a JavaScript testing framework for Node.js. Mocha was chosen for its flexibility allowing us to use other libraries for assertion and mocking. As such for the backend, SinonJS will be used for the mocking and stubbing and Chai will be used for assertion as it is a very popular library used with mocha. To test http requests, the team has opted to use the library SuperTest for its simplicity and ease of use. Another reason why the team has decided to go with mocha is its maturity which makes it easier to use as there is a vast amount of documentation and tutorials on how to implement the tests. Using Node's package manager "npm" the framework was installed in our project with the latest distribution which is 8.2.1. For SinonJS, the version 9.2.4 is installed, for Chai, the version 4.2.0 and finally for SuperTest, the version 6.1.11 . As of today these are the versions that the team is using. However this may change as the packages are set up to always install the latest versions of each library.

Below is an example of a dummy test using these libraries.



```
import { expect } from 'chai';
import request from 'supertest';
import 'mocha';

import App from '../../src/index';


describe('User Controller Test', () => {
  it('Test default route', async () => {
    const res = await request(app.app).get('/');
    expect(res.text).to.equal('Backend is running');
    expect(res.status).to.equal(200);
  });
})

```

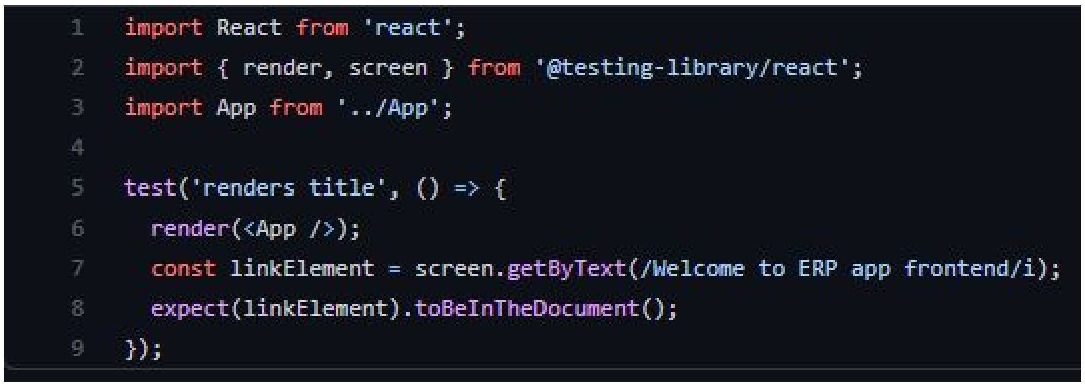
Fig 1. Dummy Test for the backend

Finally, the team is running these tests using a ts-node version of Mocha which allows us to take advantage of using TypeScript which can increase code quality and simplify coding since its points outs errors before run-time.

Frontend

For the frontend, the team has decided to go with the default testing library that comes with React which is React Testing Library. The advantages of using RTL is that the test better represents what the user sees and hides the implementation details which the team has found more appropriate for the frontend. This library tests the frontend in the eyes of the user. Another reason why RTL was chosen is it's ease of use. Since most of the implementation are hidden, the tests are simply a matter of rendering a component and verifying if it displays what was intended correctly. This also makes testing a lot faster. This library was integrated by default using the create-react-app command.

Below is an example of a dummy test using this library.



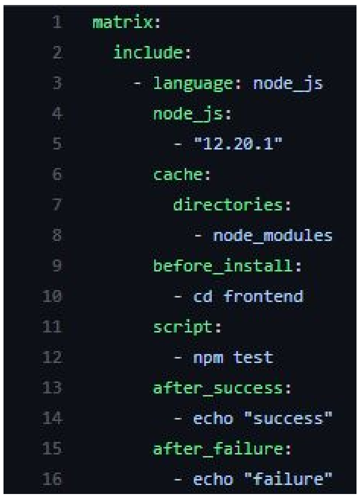
```
import React from 'react';
import { render, screen } from '@testing-library/react';
import App from '../App';

test('renders title', () => {
  render(<App />);
  const linkElement = screen.getByText(/Welcome to ERP app frontend/i);
  expect(linkElement).toBeInTheDocument();
});
```

Fig 2. Dummy Test for the frontend

1.2 Test Execution on Pull Requests

The team will be using Travis CI to build the application and run tests with each pull request. Travis CI is a continuous integration tool which is highly configurable through the ".travis.yml" file. It is well integrated with GitHub and prevents the merging of pull requests if the tests do not pass or the application fails to build.

Below is an example of the Travis CI configuration file.



```
matrix:
  include:
    - language: node_js
      node_js:
        - "12.20.1"
      cache:
        directories:
          - node_modules
      before_install:
        - cd frontend
      script:
        - npm test
      after_success:
        - echo "success"
      after_failure:
        - echo "failure"
```

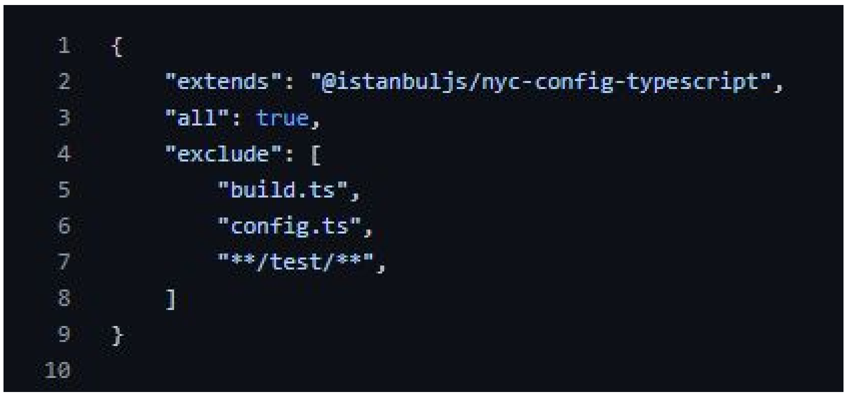
Fig 3. Travis CI configuration file

As mentioned previously, Travis will check if the application can build and whether or not every test passes. Also, tests will fail if they do not meet the coverage requirements. The current coverage is fairly low but as the project progresses more tests will be written which will increase the code coverage. The current pipeline is testing and building the frontend and the backend at the same time.

1.3 Code Coverage

The code coverage is currently only calculated in the backend. The nyc code coverage library will be used in order to calculate the code coverage of our backend. It is a highly popular library which makes documentation and tutorial widely available and it works well with our testing framework, Mocha. Also, it provides a wide array of report formats. It is also extendable with TypeScript integration which is the language the team is using.

Below is an example of the nyc configuration.



```
{
    "extends": "@istanbuljs/nyc-config-typescript",
    "all": true,
    "exclude": [
        "build.ts",
        "config.ts",
        "**/test/**",
    ]
}

```

Fig 4. Nyc configuration example

The configuration can be found in the file called ".nycrc.json". Currently the report generated has been set up to be in a text format. The report is also shown in the console when running the coverage test.

2. Weekly Activities

2.1 Weekly Tests

I. The tests are run through docker and every test should pass. This is done to maintain consistency.

II. Weekly review of bugs:

- The team will review the application after each bug fix to ensure that it did not generate new bugs and that it is fixed.

- The team planner will set an urgency level for each bug ranging from low to high priority in order to prioritise some bugs over other works if it is deemed necessary.

- Each bug will have an incident report.