

## Unit testing for React JS

### Setup

Module Name	Version	Function
Jest	(inbuilt)	Basic testing functions
@testing-library/react	12.1.2	Useful tools to handle find components and wait for specific components to update
enzyme	3.11.0	A simulator to render React
enzyme-adapter-react-16	1.15.6	Adapter between React v.16 and enzyme.
sinon	13.0.1	Useful tools for spies and mocks

```
npm install --save enzyme enzyme-adapter-react-16
```

```
npm install --save @testing-library/react
```

```
npm install --save sinon
```

### Introduction

Automated testing not only reduces human error and time required for testing, introducing the tests also records the bugs encountered throughout the history. Say if we found a bug in our system, we can make a test case for it. And if there are other changes made upon this code in the future, we do not need to re-discover the bug again. The automated testing would pick out the error before it crashed upon rare unexpected cases.

### Getting started

For this example, we are using this component as an example

On mount, it fetches data from api, calls a function foo, and switches rendering page from loading to a material table

```
import React from 'react'
import FetchApi from "../Service/FetchApi";
import MaterialTable from "material-table";

class MyComponent extends React.Component{
  constructor(props) {
```

```

super(props);
this.state = {
  TableCol: [
    /*0*/ { title: "Id", field: "Id" },
    /*1*/ { title: "Name", field: "Name" },
  ],
  isLoading: false,
  Name: null,
}
}

async PostFunction() {
  console.log("PostFunction start");

  let body = { data: "data" };
  var api = FetchApi("url");
  await api.post(body).then((response) => {
    console.log("response", response);
    this.setState({
      Name: response,
    },
    this.Foo );
  })
  .catch((exception) => {
    console.log("The function is not mocked");
  });

  if (this.state.Name !== null) this.setState({ isLoading: true });

  console.log("PostFunction end");
}

Foo = () => {
  console.log("An inner function is called")
}

componentDidMount() {
  this.PostFunction();
}

render() {
  if (this.state.isLoading) {
    return (
      <div>
        <h1>Some title</h1>

```



```

        <MaterialTable
          columns={this.state.TableCol}
          data={this.state.Name}
          title={"Table"}
        ></MaterialTable>
      </div>
    )
  }
  else{
    return(
      <div>
        <h1>Loading...</h1>
      </div>
    )
  }
}
}

export default MyComponent

```

1. Create a file with .test.js suffix. We often name it the same as our file to test.

 **MyComponent.jsx**  
 **MyComponent.test.js**

2. Import modules and setup the adapter

```

import React from 'react';
import {shallow, configure, mount} from 'enzyme'
import Adapter from 'enzyme-adapter-react-16'
import sinon from 'sinon'
// Set up an adapter to interface enzyme and react
configure({ adapter: new Adapter() });

```

3. Render the component

```
const component = mount(<MyComponent />);
```

4. Create a simple test on the title

```

it('Page loading', ()=>{
  // Despite the isLoading state is true, the page did not update
  expect(component.find('h1').text()).toEqual("Loading...");
})

```

## 5. Run testing command in the terminal

```
User$ npm test MyComponent
```

## Selector

Selectors are used in find functions to locate desired components. It can search by class name, attributes and [more](#).

- Class : “.MyClass”
- Element tag name : “div”
- Id : “#foo”
- Attribute : “[href=“google.com”]”
- Universal : \*
- React component : Button (needs to be imported)

Determine test success

[expect\(\)](#) are used for determining test success/fail. However, we don't use it directly, but with the matchers inside the expect.

- Match value  
expect.toBe(value)
- Not operator  
expect.not.toBe(value)
- Compare  
expect.toBeGreaterThan(number)

The expects can be combined to generate more flexible test cases

E.g. The test succeed if the input is called with a format {x,y}, but the values are only required to be numbers.

```
it('onPress gets called with the right thing', () => {  
  const onPress = jest.fn();  
  simulatePresses(onPress);  
  expect(onPress).toBeCalledWith(  
    expect.objectContaining({  
      x: expect.any(Number),  
      y: expect.any(Number),  
    }),  
  );  
});
```

## Simulate events

Events can be simulated by Enzyme. First, we need to locate the component, and then apply the simulation.

```
const button = component.find(Button).find({className:"ButtonName"}).at(0);
```

Sometimes the find function returns more than one node. If you know which one is your target, use `.at({index})` to locate it

- Mouse Click

```
button.simulate('click');
```

- Type in a text field

```
textfield.find('input').simulate('change', { target: { value: 'New value' } });
```

Note that the change event handler may be in the children of the textfield component.

- ★ Note that in order to test in between simulations, we need to add `done()` to the tests so that jest completes the test before moving on.

```
it('Before call, (done)=>{
  expect(Foo).not.toHaveBeenCalled();
  done();
})
it('sample test', (done)=>{
  component.find('Button').simulate('click');
  expect(Foo).toHaveBeenCalled();
  done();
})
```

## Not updating?

When state change does not reflect on the component change, re-find the component before passing it to expect.

You can also try `wrapper.update()` on your mounted component.

## Async testing

When there is an async function, the test may be run before the function is resolved, so we need to await all promises to be resolved.

```
it('is at the profile page after change password success', async(done)=>{
  const submitPWButt = component.find(Button).simulate('click');
  await Promise.resolve();
  expect(Foo).toHaveBeenCalled();
  done()
})
```

## Mock functions

The [mock functions](#), aka spies, behave like functions, but armed with multiple functionalities including tracking, and replacing the outputs. You can create a new mock function with :

```
const mockFoo = jest.fn();
```

When creating a mock function, you can also define its behaviour :

The example is a simple incrementor

```
const mockIncrement = jest.fn(x=>x+1);
```

In order to make our mock function behave like a spy, it needs to track one of our functions.

```
const spy = jest.spyOn(MyComponent.prototype, "PostFunction")
```

Note that it searches from “prototype”. That is to say arrow functions are not visible to spies, because those functions would not be present in the object’s prototype.

## Spying on functions

jest.spyOn creates a mock function that also tracks calls to that object[methodname]

1. Check called

```
expect(spy).toHaveBeenCalled();
```

2. Check results

```
expect(spy).toHaveReturned(value)
```

3. Check input

```
expect(spy).toHaveBeenCalledWith(arg1, arg2, ...)
```

## Replacing functions

For testing, we often need to create a fixed input to validate our system’s output. However, tests become inconsistent if the input relies on external sources, i.e. servers, networks, etc. , besides, waiting for servers to respond also slows down the testing, so it’s better to swap out these functions and inject our controlled input instead.

1. Locate component
2. Mock implementation

```
jest.fn().mockImplementation(()=>{value : jest.fn()})
```

If you want the function to return different values on on each call, use mockImplementationOnce instead.

```
jest.fn().mockImplementationOnce(()=>{value : First Call})  
  .mockImplementationOnce(()=>{value : Second Call})  
  .mockImplementationOnce(()=>{value : Third Call})  
  .mockImplementationOnce(()=>{value : Fourth Call})
```

## Examples

1. **Mocking a module**

Example : mock the FetchApi module

```
import React from 'react';  
import {shallow, configure, mount} from 'enzyme'
```

```

import Adapter from 'enzyme-adapter-react-16'
import sinon from 'sinon'
import { waitFor } from '@testing-library/react';
// Set up an adapter to interface enzyme and react
configure({ adapter: new Adapter() });
import MyComponent from './MyComponent'
import FetchApi from '../Service/FetchApi';
/* v--- Fake functions and datas ---v */
const mockResponse = [
  {Id:0,
    Name:"Data1"},
  {Id:1,
    Name:"Data2"},
];
// The fake post method we are going to override in Fetch api
const mockPost = () => new Promise((resolve, reject) => {
  console.log("Fake Post!")
  // Resolve it with a mock Data
  resolve(mockResponse);
});
// Tell jest what we want to mock
// Note that the mock prefix in mockPost is mandatory
jest.mock('../Service/FetchApi', () => {
  // mock out the return in this module : default FetchApi
  return jest.fn().mockImplementation(()=>{
    return{
      // mock out the function inside FetchApi
      post : mockPost
    }
  })
});
/* ^--- Fake functions and datas ---^ */
//Begin test suite
describe('MyComponent', ()=>{
  beforeEach(function() {
    // What to do before each it()
    // Set a fake timer for async functions to operate
    jest.useFakeTimers()
  });
  afterEach(function() {
    // What to do after each it()

```

```

});
// Create a spy that watches this function
// Note that the function cannot be an arrow function,
// because it does not show inside prototype, hence invisible in testing
const spy_fetch = sinon.spy(FetchApi);
// Simulate the component by mounting it
const component = mount(<MyComponent />);
// tests
// use syntax 'it' or 'test'. They are mostly the same
// each test passes only when all expects are passed
it('fetch called', () => {
  expect(spy_fetch).toBeCalled();
});
it('isLoading', ()=>{
  expect(component.state('isLoading')).toEqual(true);
})
it('Page loading', ()=>{
  // Despite the isLoading state is true, the page did not update
  expect(component.find('h1').text()).toEqual("Loading...");
})
it('Page loaded', async()=>{
  // The update must be explicitly called for it to take effect
  // In cases when the setState is inside a mocked function, the update is not
  required
  component.update();
  await waitFor(() =>{
    expect(component.find('h1').text()).toEqual("Some title");
  });
})
it('Material table loaded the mockData', ()=>{
  expect(component.find('MaterialTable').prop('data')[0].Name)
    .toEqual("Data1");
})
})

```

## 2. Mocking function inside Component

Note that the function must not be an arrow function, because spies are not able to identify it. An arrow function is generated on execution, so it does not show up inside an object's prototype, hence invisible to spies.

```
import React from 'react';
```



```

import {shallow, configure, mount} from 'enzyme'
import Adapter from 'enzyme-adapter-react-16'
import sinon from 'sinon'
import { waitFor } from '@testing-library/react';
// Set up an adapter to interface enzyme and react
configure({ adapter: new Adapter() });
import MyComponent from "../MyComponent"

import FetchApi from "../Service/FetchApi";
jest.mock("../Service/FetchApi");
/* v--- Fake functions and datas ---v */
const mockResponse = [
  {Id:0,
    Name:"Data1"},
  {Id:1,
    Name:"Data2"},
];

// The mocked version of the target function
async function mockPostFunction(){
  console.log("Fake function!");
  this.setState({
    Name : mockResponse,
  },
    this.Foo );
  this.setState({isLoading : true});
};

// Use spyOn to find the function and mock it with mockImplementation
const spy = jest.spyOn(MyComponent.prototype,
"PostFunction").mockImplementation(mockPostFunction);
/* ^--- Fake functions and datas ---^ */
//Begin test suite
describe('MyComponent', ()=>{
  beforeEach(function() {
    // What to do before each it()
    // Set a fake timer for async functions to operate
    jest.useFakeTimers()
  });
  afterEach(function() {
    // What to do after each it()
  });
});

```

```

// Create a spy that watches this function
const spy_fetch = sinon.spy(FetchApi);
const spy_getSummary = jest.spyOn(MyComponent.prototype, "PostFunction");
// Simulate the component by mounting it
const component = mount(<MyComponent/>);

// tests
it('fetch called', () => {
  //since we mocked the getSummaryAll, this fetch would not be called
  expect(spy_fetch).not.toBeCalled();
});

it("getSummaryAll called", ()=>{
  expect(spy_getSummary).toBeCalled();
  // remember to restore the mocked function so that it does not persist into
  next suite
  spy_getSummary.mockRestore();
})

it('isLoading', ()=>{
  expect(component.state('isLoading')).toEqual(true);
})

it('Page loading', ()=>{
  // This test will fail
  expect(component.find('h1').text()).toEqual("Loading...");
})

it('Page loaded', ()=>{
  expect(component.find('h1').text()).toEqual("Some title");
})

it('Material table loaded the mockData', ()=>{
  expect(component.find('MaterialTable').prop('data')[0].Name)
    .toEqual("Data1");
})
})

```

## Dealing with WithStyles

With style wraps your component. There are [several ways](#) to bypass it.  
This is the one I have succeeded.

1. Add export prefix to your component.

```
export class MyComponent extends React.Component
```

2. Use named import to import you class without being wrapped by withStyles

```
import {MyComponent} from "../MyComponent"
```

3. Mount the component with mocked classes used in withStyles.

Since the with style is not present now, calling this.props to get the styles will generate an error.

4. Proceed your testing

## Material table interaction

Since there does not seem to be an easy way to access the material table's components, we will have to do it through a web scraping manner.

Tips : If you can't find a certain element, try use find by attribute with elements you know, or data you injected. And use .parent() or .children() to navigate in the tree. Use .props() to see the element's attributes.

While checking values have been successful, making changes such as search, filter has not been successful. So I'd suggest that we avoid it in the test, and do it manually instead if needed.

- Find row

1. Locate the row in Inspector > Elements

```
▼ <div>
  ▼ <div style="overflow-y: auto;">
    ▼ <div>
      ▼ <table class="MuiTable-root" style="table-layout: auto;">
        ▶ <thead class="MuiTableHead-root">...</thead>
        ▼ <tbody class="MuiTableBody-root">
          ▶ <tr class="MuiTableRow-root" style="height: 10px;">...</tr>
          ▼ <tr class="MuiTableRow-root" index="0" level="0" path="0"
            style="transition: all 300ms ease 0s;">
```

2. Find the row by its attributes

```
var row = component.find({className : 'MuiTableRow-root', index : 0});
```

3. Use its children to retrieve their values. 0 is the left most cell.

```
console.log(a.children().at(0).props().value);
```

- Search

Not successful so far. The shown name can be different from what we see in code. Though I have located the search field, the simulation of onClick does not bring any change on its value ;<

```
const searchField = component.find({className:
'MTableToolbar-searchField-38'}) .at(0);
```

```
▼<div class="MuiInputBase-root MuiInput-root MuiInput-underline MuiInputBase-formControl MuiInput-formControl MuiInputBase-adornedStart MuiInputBase-adornedEnd"> flex
  ::before
  ▼<div class="MuiInputAdornment-root MuiInputAdornment-positionStart"> flex
    <span class="material-icons MuiIcon-root MuiIcon-fontSizeSmall" aria-hidden="true">search</span>
  </div>
  <input aria-invalid="false" placeholder="Search" type="text" class="MuiInputBase-input MuiInput-input MuiInputBase-inputAdornedStart MuiInputBase-inputAdornedEnd" value="sssa">
  ►<div class="MuiInputAdornment-root MuiInputAdornment-positionEnd">...</div>
  flex
  ::after
</div>
```