Testing Concept - JEST

Lesson 01

Capgemini

Lesson Objectives

At the end of this module you will be able to:

- Unit Testing
- React test renderer
- ReactTestUtils
- Enzyme

Unit Testing

Unit Testing is testing method by which Individual source code will be determined whether they will fit or not (ie issues) into the software.

This is done during development phase.

Spots bugs with ease, and maintain a better development workflow.

Since React is a Component architecture, unit tests are a natural fit. They're also faster because you don't have to depend on a browser.

Unit tests help you think of each component in isolation and treat them as functions.

Jest Test Framework

JEST Framework is

1. Easy to setup
2. Zero configuaration
3. It's developed by Facebook

Jest is also faster than the rest because it uses a clever technique to parallelize test runs across workers. Each Test runs in a sandbox environment to avoid conflict between 2 successive tests.

If we create a project using create-react-app, it comes along with JEST. So, no need to install jest. But its mandate to install **react-test-renderer** as shown below.

npm install –save react-test-renderer

Setup with Create React App

We have an existing application you'll need to install a few packages to make everything work well together.

We are using the **babel-jest** package and the **react babel** preset to transform our code inside of the test environment.

Npm install --dev jest babel-jest @babel/preset-env @babel/preset-react react-test-renderer

And we have to specify the version of jest in devDependencies and also in scripts as shown below

"scripts": { "test": "jest" }

Add correct version of jest in packages .json file

Jest looks for tests to run using the following conventions:

Files with .test.js suffix.
Files with .spec.js suffix.
Files with .js suffix inside a folder named **tests**.
Other than .js files, it also automatically considers files and tests with the jsxextension.

Create a file inside the app/__tests__ folder, name it app.test.js and add the following to the file. Inside it write following code:

```
describe('App', () => {
it('should be able to run tests',
() => { expect(1 + 2).toEqual(3); });
 });
```

To create a test, you place its code inside an it() or test() block, including a label for the test. You can optionally wrap your tests inside describe() blocks for logical grouping.

Jest comes with a built-in expect() global function for making assertions

You can now run the added test with npm test and see the results in the Terminal.

Before doing above step ensure that in packages.json file we have added “test” : “jest”

Jest Matchers

Matchers are the most basic unit for testing in Jest. They are used to assert that an expected object is equal, close to, greater, or less than, or contains any other logical connection to another output object. We can create different constructions with these logical connections

```
test(‘object assignment’, () => {
const data = {one: 1};
data[‘two’] = 2;
expect(data).toEqual({one: 1, two: 2});
});

test(‘increasing a positive number is not zero’, () => {
let a = 1;
expect(a + 1).not.toBe(0);
});

test('adding floating point numbers', () => {
const value = 0.1 + 0.2;
// It fails because in JavaScript 0.2 + 0.1 = 0.30000000000000004 jejeje.
// expect(value).toBe(0.3);
// This works with a precision of 5.
expect(value).toBeCloseTo(0.3, 5);
});
```

Jest Mocks

Mocking is a useful way to isolate components and functions. To do this, we define the output of our function/dependency. A good mock is the one that mimics all corner cases. Another excellent Jest feature is the ease of creating mocking functions/dependencies.

```
test('computeList calls callback in array', () => {
function computeList(items, callback) {
for (let index = 0; index < items.length; index++) {
callback(items[index])
}
}

const mockCallback = jest.fn()
.mockReturnValue(32) computeList([0, 1], mockCallback) // The mock function is called twice
expect(mockCallback.mock.calls.length).toBe(2) // The first argument of the first call to the function was 0
expect(mockCallback.mock.calls[0][0]).toBe(0)
})
```

Setup and teardown

A good practice in Jest is organizing data in blocks. To do this, we can use describe function that receives two arguments: first the block name and then a function that will either have more organized blocks or test functions. To make it more modular, Jest provides a set of functions that will control code execution before and after we runtests.

```
describe('City Database', () => {
beforeAll(() => {
initEnvironment()
})

beforeEach(() => { initCityDB() })

afterEach(() => { clearCityDB() })

test('has Vienna', () => { expect(isCity('Vienna').toBeTruthy()) })

test('has San Juan', () => { expect(isCity('San Juan').toBeTruthy()) })
})
```

Demo

React-Jest-demo1
React-Jest-demo2
Jest-react- master
CountdownTimer_advanced with babel_wp

Snapshot Testing

➢Snapshot Testing is a useful testing tool in case you want to be sure user interface hasn’t changed.

➢ which automatically generates text snapshots of your components and saves them to disk so if the UI output changes later on, you will get notified without manually writing any assertions on the component output.

- test('function filters users', () => {
  expect(filterUsers('b').toEqual([
  { name: 'Becky' },
  { name: 'Bob' },
  { name: 'Bryan' },
  { name: 'Bryce' }
  ]))
  })

➢But, what happens if we add users or change schema? We have to change the test, copying and pasting the assertion to make the test pass. The logic didn’t change, only the expected output. This can happen multiple times in a project. Snapshots are the solution to making this repetitive process automatic and auto-generated.

➢test('filters items', () => {
expect(filterUsers('b')).toMatchSnapshot()
})

Enzyme API & JEST

- Instead of react-dom/test-utils , we can use the Enzyme package which was released by Airbnb.
  - ***Enzyme is a JavaScript Testing utility for React that makes it easier to assert, manipulate, and traverse your React Components' output.***
- Enzyme is a library that wraps packages like React TestUtils, JSDOM and CheerIO to create a simpler interface for writing unit tests.
- React TestUtils has methods to render a react component into a document and simulate an event
- Enzyme is **not** a unit testing framework.
- It does not have a test runner or an assertion library. It works with any test runner and assertion library.

○ Enzyme	○ Jest
1. Enzyme is a testing library	Jest is a unit testing framework and has a test runner, assertion library, and mocking support.
1. Jest can be used with non-react applications.	Enzyme works only with React. Jest

Enzyme.js

Enzyme.js is an open-source library maintained by Airbnb, and it's a great resource for React developers.

It uses the ReactTestUtils API underneath, but unlike ReactTestUtils, Enzyme offers a high-level API and easy-to-understand syntax.

The Enzyme API exports three types of rendering options:

- shallow rendering
- full DOM rendering
- static rendering

Enzyme can be installed through

Below is an example of Shallow rendering

```
import { shallow }  from 'enzyme';
import App from './App';

// More concrete example below.
 const component = shallow(<App/>);
```

ReactTestUtils

- Makes it easy to test React components in the testing framework
- We can import and use in program as below

```
import ReactTestUtils from 'react-dom/test-utils'; // ES6

var ReactTestUtils = require('react-dom/test-utils'); // ES5 with npm
```

- We have many methods in this API. Few of them are,
  - act()
  - mockComponent()
  - isElement()
  - isElementOfType()

- Take demo from https://reactjs.org/docs/test-utils.html

```
import { configure } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';configure({ adapter: new Adapter()
});

describe('App component', () => {
test('should shallow correctly', () => {
expect(shallow(
<App />
)).toMatchSnapshot()
})
test('should mount correctly', () => {
expect(mount(
<App />
)).toMatchSnapshot()
})
test('should render correctly', () => {
expect(render(
<App />
)).toMatchSnapshot()
})
```
})Enzyme has three methods for rendering React components. These methods give dif

Demo

React enzyme demo1
React-jest-test-clock

Defining Queue for Test coverage

- How to define the correct order of component testing in shared directory:

  - Independent
  - Auxilary
  - Simple
  - Complex
  - Field (generally forms)

  So order will be like, Forms, Utils, Widgets, Modals, HOC

  Things to be ignored while Test Coverage is

- ***Third-party libraries***
- ***Constants***
- ***Inline styles***
- ***Things not related to the tested component***

There are 2 ways to test the applications.

1. Snapshot Testing
2. Component logic testing

Summary

Review Questions

- Question 1:
- Question 2: