JEST

Introduction

Sreekanth M. E.

Freelance Trainer & Consultant

http://www.SreekanthME.com

Test runner: Executes tests and writes test results.

Ex: Mocha, Jasmine, Jest

Assertion library: Verifies results of tests.

Ex. Chai, Should, Expect, Jest

Test Library: Special library to help test in specific programming language.

Ex. react-test-renderer (part of Jest), Enzyme

Although Jest may be assumed to be a React-specific test runner, in fact it is a universal testing platform, with the ability to adapt to any JavaScript library or framework.

Jest is already configured when you use create-react-app

Jest will automatically pickup tests placed in a __tests__ folder, or test files named with a .spec.js or .test.js extension.

Jest has a novel way to test react components: Snapshot testing.

With snapshot testing, the output of the current test run is compared with the snapshot of the previous test run. If the output matches the snapshot, the test passes.

Snapshot tests are a very useful tool whenever you want to make sure your UI does not change unexpectedly.

But not useful for TDD.

Jest automatically stores snapshots in a folder named __snapshots__

The snapshot for a file named Label.test.js is stored in a file named: Label.test.js.snap

Jest requires react-test-renderer library or enzyme library to test React components.

These libraries render React components so that Jest can run tests on them from command line without needing a browser.

Apart from Snapshot testing, Jest can traverse the output (rendered by test libraries) to find specific nodes and make assertions about them.

ENZYME

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.

JSDOM is a JavaScript implementation of the DOM.

CheerIO implements a subset of jQuery core and is used to query the DOM.

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 works only with React. It is a Test Library for React.

React components can be tested in the following modes:

- Shallow rendering
- Full DOM Rendering
- Static Rendered Markup

Enzyme supports all 3.

react-test-renderer supports only Shallow rendering.

SHALLOW RENDERING

Shallow rendering renders only the component itself without its children.

So if you change something in a child component it won't change shallow output of your component. Or a bug, introduced to a child component, won't break your component's test. It also doesn't require DOM.

FULL DOM RENDERING

Full DOM rendering actually mounts the component in the DOM.

Mounting is achieved by a library called jsdom which is essentially a headless browser implemented completely in JS.

STATIC RENDERED MARKUP

This mode renders React components to static HTML and analyze the resulting HTML structure.

Enzyme uses a third party HTML parsing and traversal library Cheerio.

END OF CHAPTER

APPENDIX

Jest API

The Jest API focusses more on the ability to define tests, make assertions, and create mocks.

describe: defines a test suite.

it: defines a test.

beforeEach: defines an entry hook before running each test.

expect: makes an assertion.

jest.fn(): creates a mock function.

Several methods are available for assertions.

toEqual: checks if two objects have the same value.

toBe: checks if two objects have the same value and type.

toBeDefined: checks if the object is defined.

toContain: checks if an item is present in a list.

toBeCalled: checks if a mock function is called.

There are useful methods on the mock.

mockImplementation provides an implementation for the mock function.

mockReturnValue returns a value when the mock function is called.

Enzyme API

Enzyme API focusses on rendering the react component and retrieving specific nodes from the rendered tree. There are several methods to retrieve nodes from the rendered component.

find: accepts a selector and retrieves nodes that match the selector.

findWhere: retrieve nodes selected by the predicate.

some: returns true if there is at-least one node matching the selector.

someWhere: returns true if there is at-least one node selected by the predicate.

first: returns the first node of a set.

at: returns the nth node of a set.

html: gets the HTML of the node.

text: gets the text representation of the node.

There are more methods to interact with the component and retrieve the component state.

simulate: simulates an event.

setProps: sets the props.

setState: sets the state.

setContext: sets the context.

prop(key): retrieves prop value corresponding to the provided key.

state(key): retrieves state corresponding to the provided key.

context(key): retrieves context value corresponding to the provided key.

A headless browser is a web browser without a graphical user interface. In other words it is a browser, a piece of software, that access web pages but doesn't show them to any human being. They're actually used to provide the content of web pages to other programs.

For example, a headless browser can be used by a computer program to access a web page and determine how wide that page (or any element on it) would appear to be by default for a user, or what colour text in any element would be, the font family used or even what the x/y coordinates of an object is.

This data is often used to test web pages en mass for quality control or to extract data.

The headless browser is significant because it understands web pages like a browser would – with the caveat that browsers all (annoyingly) behave slightly differently. Headless browsers, for example, should be able to parse JavaScript. They can click on links and even cope with downloads.

http://blog.arhg.net/2009/10/what-is-headless-browser.html