نکات پیشرفته ی ری اکت

Testing

ابتدا یک پروژه را به صورت سمپل بسازیم

برای یک تست ساده فایل App.js مثلا



برای ران کردن هم باید npm run test را بزنیم.

داخل Expect باید تست مورد نیازمون رو بنویسیم.

برای تک تک کارهایی که میخواهیم انجام دهیم می توانیم تست بنویسیم. مثلا رندر شدن کامپوننتی یا کلیک و...

می توانیم دو نوع پوشه بندی تست کنیم.jest تمامی فایل test را اجرا میکند.

می توانیم کنار هر فایل. فایل تست هم داشته باشیم وهم می توانیم یک directory جدا داشته باشیم برای تست ها به نام \_\_test\_\_ و داخلش فایل های تست را بزاریم

It(“name of test, test for botton click”,()=>{

This function is test code.

})

قطعه کد بالا فقط متن داخل یک div را چک میکند به این منظور این div که ما ساختیم به عنوان یک div مجازی وجود دارد(شبیه سازی) و داخل div کامپوننت اصلی را شبیه سازی می کنیم بعدش چک می کنیم که فقط این متن داخل این کامپوننت هست یا نه یعنی بودن یا نبودن کامپوننت را بررسی نمی کند و بعد از تست کردن باید unmount کند.

Expect(the thing that we want to verify).matcherStatement(value we want to see)

برای تست کردن وجود یک کامپوننت باید از enzyme استفاده کنیم.

نصب enzyme تنها در محیط development نصب می شود. Enzyme یک نسخه از نسخه ری‌اکت کمتره. باید خود enzyme و enzyme adapter react را باید نصب کنیم

باید داخل پوشه src یک فایل به نام setupTests.js ایجاد می کنیم و باید config ، enzyme را انجام دهیم.

Import Enzyme from “enzyme”

Import Adapter from “@wojtekmaj/enzyme-adapter-react-17”

Enzyme.configure({adapter : new Adapter()})

کد بالا باید در فایل setupTests.js نوشته شود.

Jest قبل از همه کار هاش میاد تو پوشه src و دنبال فایل setupTests.js می گرده و ما از این ویژگی برای کانفیگور کردن enzyme استفاده می کنیم

Enzyme api

* Static:render given component and return plain HTML
* Shallow : render just the given component none of its children
* Full Dom : render the component and all its children + let us mmodify it afterwards

برای خواسته بالا باید از shallow استفاده کنیم

Import {Shallow} from “enzyme”

It(“show component”,()=>

{

Const wrapper = shallow(<Wrapper\_Component/>);

Expect(wrapper.find(<child\_Comp>).length).toEqual(1);

})

اگر هر کدام از تست ها (می توانیم 500 تا از این مدل it را زیر هم داشته باشیم ) قسمت یکسانی داشته باشد می توانیم از beforeEach استفاده کنیم.

Let Wrapper

BeforeEach(()=>{

Wrapper = shallow(<App />;

});

اگر از روش full rendring استفاده می کنیم حتما باید اخرش unmount کنیم چرا که full rendring بر خلاف Shallow و static بر روی DOM اجرا می شود که اگر unmount را فراخوانی نکنیم ممکن است به سایر اجزایی که در DOM هستند خللی ایجاد کند.

برای استفاده از full rendering باید mount را از enzyme ایمپورت کنیم

Import {mount} from “enzyme”

Mount همانند Shallow می باشد (طریقه ی استفاده کردن)

برای اینکه بفهمیم چه المانی در صفحه است باید از full rendering و mount استفاده کنیم و از متد find مثل مثال زیر.

It(“show textarea and button”,()=> {

Const wrapper = mount(<CommentBox />);

Expect(wrapper.find(“textarea”).length).toEqual(1);

Expect(wrapper.find(“button”).length).toEqual(1);

})

AfterEach : در داخل این قسمت (که همانند beforeEach است ) کدی نوشته می شود که قرار است بعد از هر تست اجرا شود، مثلا در full rendering باید بعد از هر تست wrapper را unmount کنیم.

AfterEach(()=>

{

Wrapper.unmount();

})

برای یک تست کردن عملکرد یک textarea مبنی بر اینکه onchange صدا زده می شود یا نه و اینکه استیت اپدیت می شود و صفحه به روز می شود یا نه باید به مدل زیر عمل کرد.

برای پیدا کردن textarea کار راحتی داریم.

برای فانکش change باید یک تابع را به صورت fake ، simlate کنیم که در قسمت

Simulate(“change”) اورده شده است. حالا باید یک event را بهش پاس بدیم که دقیقا یک event فیک را ساختیم که هم تارگت و هم value دارد.

برای اپدیت کردن هم باید از متد update استفاده کرد. حال باید چک کنیم که مقدار value در textarea تغییر کرده است یا نه. برای این کار از متد prop(key) استفاده کنیم که طرز استفاده به صورت زیر است که در اینجا به جای key باید از value استفاده کنیم چون ما میخواهیم value ی textarea را بررسی کنیم

It(“find textarea”,()=>

{

Expect(wrapper.find(“textarea”).length).toEqual(1);

})

It(“textarea function”,()=>{

Wrapper.find(“textarea”).simulate(“change”,{

Target:{value: ” this is fake event”}

})

Wrapper.update();

Expect(wrapper.find(“textarea”).prop(“value”)(.toEqual(“this is fake event”);

})

Test 2

برای تست اینکه در زمان ران کردن در حالت test watch mode بماند باید از npm test استفاده می کنیم که منتظر تغییر در فایل های تست است.

Const wrapper = shallow (<App />);

Expect(wrapper).toBeTruthy()

این یک تابعی است که در صورت وجود این wrapper مقدار true را برمی گرداند.

برای بررسی وجود یا عدم وجود یک کامپوننت باید از shallow استفاده کرد.

Const wrapper = shallow (<App />);

Shallow یک متد هایی داره برای سلکت کردن المان ها.

یک اتریبیوتی که برای تست نوشتن برای المان ها نیاز می شود اتریبیوت data-test = “string” است. به این صورت

<div className = “ App “ data-test = “app component” />

Data-test اتریبیوتی است که فقط برای تست کردن استفاده می شود و جز تست هیچ کارایی دیگری ندارد.

Const selectedElement = wrapper.find(“[date-test=‘app component ’]”)

Expect(selectedElement.length).toBe(1);

در قسمت “[data-test=’app component‘]” از زدن space اضافی پرهیز کنیم. البته در این قسمت می توانیم به جای استفاده از data-test از className و id هم استفاده کرد.

چک کردن و تست کردن ورودی های یک کامپوننت:

برای انجام این کار باید از propType استفاده کنیم که برای نصب این کتابخانه باید دستور زیر را به کار ببریم.

npm install –save prop-types

نحوه ی عملکرد به این صورت است که در تابعی که برای return کردن wrapper داریم باید props={} را بهش پاس بدیم

Const setup = (props={})=>

{

Return shallow(<App {…props} /> );

}

حال برای چک کردن ورودی ها (اینکه اصلا فرم ورودی ها درست است یا نه باید از کتابخانه ی prop-types استفاده کنیم. به این صورت که باید از به فرم زیر عمل کرد

Import PropTypes from “prop-types”

Const propError = PropTypes.checkPropTypes(myComponent.PropTyps, expectedProps, “prop” ,myComponent.name);

Ecpect (propError).toBeUndefined();

اگر ورودی های ما درست باشد این ارور برابر با undefined می شود و تست ما پاس می شود.

فرم expectedProps :

Const expectedProps = {success : true}

This means that we have one component that we expect it’s props should be like expectedProps.

بعد از تست نوشتن باید در فایل component مان ابتدا PropTypes را import کنیم و در اخر کامپوننت قبل از export ، component.PropTypes را تعریف کنیم.

Component.PropTypes = {

Success : PropTypes.bool.isRequired

{

// is required is optional

Export default …

Describe :

برای اینکه کد های تست را جدا کنیم از decribe استفاده می کنیم. مثلا یک کامپوننت دو حالت دارد با توجه به شرایطی، به همین دلیل در فایل تست هم این دو تکه را از هم جدا می کنیم پس از describe استفاده می کنیم و همچنین اینکه تست های موجود در describe های جداگونه از هم مستقل هستند.

فانکشن ها:

اگر یک فانکشنی که نوشتیم را بخواهیم تست کنیم می بایست چند تست را برای ان بنویسیم تا شرایط و حالات گوناگون را تست کنیم و خروجی های مختلفی را تست کنیم.

ریداکس:

اگر به پروژه ریداکس را اضافه کردیم در کنار هر فایل action یا reducer می بایست یک فایل تست بنویسیم.

مثلا برای action ها ببینیم که ایا تایپ درستی را بر می گراند یا خیر. برای اینکار باید در فایل تست به این صورت عمل کنیم

Import myAction from “./action”

Describe(“testing actions”,()=>

{

Test(“does it return true action”,()=>

{

Var sampleAction = myAction();

Expect(sampleAction).toStricEqual({

Type: ”action\_Type”

})

})

})

نکته‌ی بسیار مهمی که باید به ان توجه کرد این است که امکان تست کردن استیت ها در فانکشنال کامپوننت نیست بلکه برای تست کردن استیت ها باید از کلاس کامپوننت استفاده کنیم

Import React , {Component} from “react”

Import {connect} from “react-redux”

Class Comp extend Componnet {

Render(){

Return <div> this is Comp </div>

}

}

Const mapStateToProps=(state)=>

{

// this section will receive state and convert in to props

};

Export default connect(mapStateToProps)(Comp)

زمانی که برای تست کردن استیت ها اقدام میکنیم و از class component استفاده می کنیم در تابع setup زمانی که component را shallow میکنیم باید به store دسترسی داشته باشد. (به دلیل وجود connect چون از ری اکت ریداکس import شده). به همین دلیل برای تست شدن این تست ها باید در هنگام shallow کردن باید Store به کامپوننت پاس داده شود یعنی به فرم زیر

Const wrapper = shallow(<Input store={Store} />)

که چون در فایل تست به Store دسترسی نداریم باید از متد createStore استفاده کنیم که هم می توانیم دقیقا در تابع setup ، store جدید را بسازیم و یا می توانیم در آن فایلی که همه توابع مشترکمان بود بسازیم و بعد import کنیم فقط در هنگام پاس store علاوه بر reducer یک initialState هم پاس می دهیم. این initialState به عنوان ورودی تابع خواهد بود و به createStore پاس داده می شود و این initialState همان state موجود در ورودی setup است.

راه دوم برای هندل کردن Store این است که خود Store را از فایل Store.js ، import کنیم.

زمانی که کامپوننت ما دارای store است ، wrapper مثل قبل نیست. یعنی آن wrapper کلی که فقط کامپوننت اصلی ما داخلش باشد نیست، بلکه اگر wrapper.debug() را cnosole.log کنیم عبارت زیر را مشاهده می کنیم.

<ContextProvider value={{…}}>

<Comp store={{…}} dispatch={[Function: dispatch]} />

</ContextProvider>

که contextProvider یک high order component است و Comp اصلی ما به عنوان child است.

اما برای component ای که store نداشته باشد از این contextProvider ها ندارد و دقیقا از container آن کامپوننت را console میکند پس این نکته بسیار مهم است و ان wrapper بالا که contextProvider دارد به درد تست کردن نمی خورد و باید به خود اون child (کامپوننت اصلی) دسترسی داشته باشیم که برای این کار باید از ابزاری به نام dive استفاده کنیم. به این صورت <=

Const warpper = shallow(<Comp store={Store} />).dive();

حال اگر این wrapper را چاپ کنیم خط زیر نمایش داده می شود

<Comp store={{…}} dispatch={[Function: dispatch]} />

که باز هم به درد ما نمی خورد و مجددا باید dive کنیم تا به داخل متد render کامپوننت ما برود

Const warpper = shallow(<Comp store={Store} />).dive().dive();

خروجی console بالا برابر با <div>this is Comp</div> می شود که این برای تست ما مناسب است.

Redux thunk :

ابزاری است که انعطاف پذیری بیشتری را به action های ما می دهد و اجازه میدهد که چندتا action را dispatch کنیم و همچنین این امکان را می دهد که به مقدار state ها هم دسترسی داشته باشیم.

برای اینکار ابتدا باید redux thunk را نصب کنیم برای اینکار از دستور

Npm install redux-thunk استفاده می کنیم و در فایل configureStore باید از applymiddleware ها استفاده کنیم و دیگر نیازی به createStore نداریم.

ConfigureStore.js = >

Import {applyMiddleware ,createStore} from “redux”

Import ReduxThunk from “redux-thunk”

Import RootReducer from “#your reducer file path”

Export const middlewares = [ReduxThunk];

Const Store=createStore(RootReducer,applyMiddleware(…middlewares));

Export default Store;

هرجایی که از createStore استفاده کردیم باید به جاش از خطوط کد بالا استفاده کرد.

Const Store=createStore(RootReducer,applyMiddleware(…middlewares),initialState);

Return Store;

این خط با خط زیر برابری می کند.

Const Store = applyMiddleware(…middleware)(createStore);

Return Store(RootReducer,initialState)

برای تست کردن صحت عملکرد action و reducer نباید تست های جداگانه (unit testing) طراحی کرد بلکه باید به صورت integration test انجام داد چون به هم ارتباط دارند. (الان تمامی تست های ما unit test بوده اما برای action , reducer باید integration باشه. پس یک دایرکتوری جداگانه درست میکنیم.

تست نوشتن برای فراخوانی های رابط API

تست نویسی برای axios :

همانطور که قبلا بررسی کردیم زمانی که یک کامپوننت را به صورت مجازی shallow میکنیم. فقط و فقط به اجزای داخل آن دسترسی داریم و دیگر به اجزای مختلف برنامه دسترسی نداریم و زمانی که رابطی را فراخوانی می کنیم یعنی به سرور ریکوئست می زنیم که با این مقوله ی تست ما تناقض دارد به همین دلیل از ابزاری دیگر باید استفاده کنیم یعنی ان ابزار ها فرایند ریکوئست زدن را شبیه سازی می کند و یک fake request می زند و همچنین یک fake server هم می سازد تا به ما response دهد. نام ان ابزار moxios است. برای نصب ان ابتدا باید خود axios را نصب کنیم و سپس moxios را نصب کنیم. توجه داشته باشیم که نباید این دو را با هم نصب کرد چون محیط نصب این دو جداست.

Axios in production environment and Moxios in development environment

Npm install axios

Npm install - - save-dev moxios

در فایل action.test.js که به تست کردن moxios پرداختیم باید در describe یک beforeEach و یک afterEach داشته باشیم که در before باید moxios.install() و در after moxios.uninstall() را وارد کنیم.

برای تست کردن اینکه از رابط فراخوانی کنیم باید از خود axios استفاده کنیم ولی باید علاوه بر axios باید کد moxios را هم بنویسیم به این صورت:

Moxios makes fake server and fake response with real request that requested by axios and our duty is make axios's response.

moxios.wait(()=>

{

Const request = moxios.requests.mostRecent();

به این معنا که moxios صبر کن و اخرین ریکویستی که اومد را با این پاسخ جواب بده.

request.respondWith({

Status : 200,

Response : myAnswer

});

});

حال در این قسمت می خواهیم به تست کردن خود ریداکس بپردازیم یعنی اینکه ایا کامپوننت ما به استیت های مورد نیازش دسترسی دارد یا نه و همچنین علاوه بر استیت های مورد نیاز به اکشن های مورد نیازش هم دسترسی دارد یا خیر برای اینکار باید از wrapper.instance().props استفاده کنیم همانند کد زیر

Test("does my component have access to props",()=>

{

Const success = true;

Const wrapper = setup({success});

Const successProps = wrapper.instance().props.success ;

Expect(successprops).toBe(success)

})

قطعه کد بالا برای تست کردن اینکه ایا به استیت ها دسترسی دارد یا خیر حال می خواهیم تستی بزنیم که بررسی کنیم ایا در ورودی هایش action creator وجود دارد یا نه.

Test("has access to guessWord action creator as a function props",()=>

{

Const wrapper = setup();

Const actionPeops = wrapper.instance().props.guessWord;

Expect(actionProps).toBeInstanceOf(Function)

});

این به این معناست که ایا در ورودی های کامپوننت guessWord داریم که از جنس function باشد یا نه.

حال برای اینکه مثلا اکشن getSecretWord را امتحان کنیم (که از رابطی مقداری دیتا را دریافت می کند و state ای را اپدیت میکند) که ایا یکبار فراخوانده می شود یا چند بار باید کلا روند تست کردن را تغییر داد یعنی قاعده و قانون خاصی ندارد. مرحله ی اول این است که ان کامپوننتی که این اکشن به عنوان props در mapStateToProps آن قرار دارد یک نسخه ی کپی بگیریم که به Store متصل نباشد یعنی به connect متصل نباشد(مثلا اسم کامپوننت : unConnected App ) و کپی از اکشن را به عنوان Props به component جدیدمان می دهیم که به آن Mock می گویند(Mock function). از ویژگی های بسیار مهم و خوب این Mock function این است که می توانیم ببینیم چند بار ران شده است. علاوه بر این کار ها می خواهیم که توابع lifeCycle کامپوننت ها را هم disable کنیم چونکه می خواهیم هر وقت که خودمون خواستیم این متد ها فراخوانده شوند. به همین دلیل در داخل فایل setupTests.js باید بعد از instanciate کردن از adapter باید داشته باشیم

disableLifecycleMethodes : true;

فرایند تست کردن این کار را در پایین ذکر کردم.

1. We have to add disableLifecycleMethods : true; in setupTests.js
2. Change Name of App component to UnconnectedApp and export this like bellow

Export class UnconnectedApp extends Components {

ComponentDidMount(){

// here should run getSecretWord that it is coming from Props of UnconnectedApp in test file.

}

.

.

.

.

Export default connect(mapStateToProps , {getSecretWord})(unConnectedApp)

}

تنها تغییر ما در این قسمت فقط تغییر نام کامپوننت بود و همچنین بعد از disable کردن lifeCyclemethod ها خودمان ComponentDidMount را می نویسیم.

در فایل تست

Test(“getSecretWord run on App Mount”,()=>

{

Const getSecretWordMock = jest.fn();

Const wrapper = shallow(<UnconnectedAppgetSecretWord=getSecretWordMock }/>)

این خط به این معنا است که تابع جدید و فیک را به عنوان ورودی به آن تابع پاس می دهیم و نیازی به این نیست که از state بگیرد.

Wrapper.instance().componentDidMount();

به این معناست که componentDidMount را اجرا کن(که در بالا هم در داخل آن تابع گفتم که باید getSecretWord ورودی اجرا شود یعنی داخل componentDidMount باید به فرم زیر باشد.

this.props.getSecretWord();

یعنی getSecretWord ای که از ورودی گرفتی را اجرا کن که همان Mock است.

Const getSecretWordCallCount = getSecretWordMock.mock.calls.length;

Expect(getSecretWordCallCount).toBe(1);

})

پس به طور کلی تست به صورت زیر است:

const getSecretWordMock = jest.fn();

const wrapper = shallow(<UnconnectedApp getSecretWord={getSecretWordMock} />);

wrapper.instance().componentDidMount();

const getSecretWordCallCount = getSecretWordMock.mock.calls.length;

expect(getSecretWordCallCount).toBe(1);