Priscila-Rojas

REact Hooks y MERN

**Repaso JS**

**Funciones de Flecha:**

* Se puede escribir en solo un reglón.

*const* *saludar* = (*nombre*)*=>* `Hola, ${nombre}`;

* Para retornar un objeto lo debo poner entre (), así se omite el return

*const* *getUser* = ()*=>* ({

*// Los parentesis me indica que voy a devolver el objeto que se encuentre dentro.*

    uid: 'ABC123',

    username: 'El\_Papi1502'

  })

**Desestructuración de Objetos/ Asignación Desestructurante**:

Se puede extraer las propiedades de un objeto creando variable por cada una de ellas, es importante que en el caso de los objetos los nombres de las variables tengan el mismo nombre que las propiedades.

*const* persona = {

  nombre: 'Tony',

  edad: 45,

  clave: 'Ironman'

}

*//              asigno un Nuevo nombre a la variable*

*const* { nombre, edad:year, clave} = persona

Se suele usar mucho la desestructuración en los parámetros de una función, que espera recibir un objeto.

*//asigno un valor predeterminado*

*const* *useContext* = ({*nombre*, *edad*, *clave*, *rango* = 'Capitán'}) *=>* {

  return {

    nombreClave: clave,

    anios: edad,

    latlng: {

      lat: 14.1232,

      lng: -12.3232

    }

  }

}

**Desestructuración Objetos anidados**

Cuando queremos desestructurar un objeto dentro de otro, hacemos mención al nombre de ese objeto anidado y solicitamos sus propiedades a desestructurar con su respectivo nombre, creando así las variables con los datos dentro del objeto anidado.

*const* *useContext* = ({*nombre*, *edad*, *clave*, *rango* = 'Capitán'}) *=>* {

  return {

    nombreClave: clave,

    anios: edad,

    latlng: {

      lat: 14.1232,

      lng: -12.3232

    }

  }

}

*const* {nombreClave, anios, latlng: { lat , lng}} = *useContext*( persona );

**Desestructuración de Arrays**

A diferencia de la desestructuración de objetos, esta se hace entre [ ], puedo nombrar cada variable con el nombre que yo quiera. El orden de desestructuración es exactamente mente que en el que están en el array, en caso que no quiera algún elemento del array simplemente dejo el espacio en blanco seguido de una coma y salta ese elemento.

*const* personajes = [ 'Goku', 'Vegeta', 'Trunks'];

*// Las comas indican que ignore esas posiciones*

*const* [ , ,personaje3 ] = personajes;

**Import, Export y Export Default**

**Export:** Nos permite trabajar con la información/ funciones disponibles en este archivo en otros archivos externos sin necesidad de copiarla, se exporta un objeto el cual debo desestructurar al momento de importarlo.

export *const* heroes = [{

        id: 1,

        name: "Batman",

        owner: "DC",

    },

{

        id: 2,

        name: "Spiderman",

        owner: "Marvel",

    }]

Podemos exportar varios objetos, funciones a la vez colocando el Export al final del archivo

export {

    heroes,

    owners

}

**Export Default:** Nos permite trabajar con la información/ funciones disponibles en este archivo en otros archivos externos sin necesidad de copiarla. Se importa directamente sin desestructuración, podemos asignarle el valor que deseemos al momento de importarlo. Es recomendable exportarlo al final del archivo.

export default heroes

Dentro de un export puedo exportar algún elemento en especial con un export default.

export {

    heroes as default,

    owners

}

**Import:** Nos permite traer información o funciones de otro archivo externo (previamente exportado) al cual estoy trabajando. Los export default se importan sin desestructuración y los export con desestructuración.

import owner { heroes } from "./data/heroes";

**Promesas**

Las promesas son asíncronas por lo que primero se va a ejecutar todo el código síncrono y as promesas quedan en la pila de ejecución ejecutándose al final.

Reciben 2 parámetros: resolve y reject, el resolve se ejecuta cuando la promesa es resuelta y el reject cuando la promesa es rechazada.

*const* promesa = new *Promise* ((*resolve*, *reject*)*=>* {

*setTimeout*(() *=>* {

*const* hero = *getHeroeById*(3)

    hero? *resolve*(hero): *reject*('no found');

  }, 2000);

});

**.then:** nos permite capturar la resolución de la promesa y poder hacer lo que deseemos con lo que nos regresó. Pasándole como parámetro lo que la promesa nos devuelve, dicho para metro puede tomar el nombre que yo elija no necesariamente resolve.

promesa.*then*( (*resolve*)*=>*{

  console.*log*(resolve)

})

Para poder manejar la información debemos parsear la información y convertirla a un objeto manejable.

El método . Json devuelve una promesa la cual con otro .then podemos majear como queramos. El primer .then del siguiente ejemplo toma lo que devuelve petición y lo transforma a objeto, el 2do .then toma ese objeto que nos devuelve el .json y lo trabaja como nosotros le digamos.

peticion

  .*then*( *resp* *=>* resp.*json*())

  .*then*( ({ *data* }) *=>* {

*const* { url } = data.images.original;

*const* img = document.*createElement*('img');

    img.src = url;

    document.body.*append*( img );

  })

**.catch**: Permite capturar el error o motivo por lo cual la promesa no pudo ser resuelta.

.*catch*(*reject=>*{

    console.*warn*(reject)

})

**Async, Await y Try{} Catch{}**

La función **async()** retorna una promesa. El Await nos ayuda a trabajar nuestro código y/o función como código síncrono. El **await** debe ir siempre dentro de una función async().

Todo el código que queramos ejecutar en la función va dentro del **try** en caso de que surja algún error se coloca dentro del **Catch**, sería como si el try fuera el .then y el catch él .catch de las promesas. La siguiente función hacemos lo mismo que en el ejemplo del .then pero son función asunc().

*const* *getImagen* = async() *=>* {

  try{

*const* apiKey = 'zROSwXHym6UaM8CszCpL4s9Lfv3WDzR0';

*const* peticion = await *fetch*(`https://api.giphy.com/v1/gifs/random?api\_key=${apiKey}`)

*const* { data } = await peticion.*json*();

*const* { url } = data.images.original;

*const* img = document.*createElement*('img');

        img.src = url;

        document.body.*append*( img );

  }catch(e){

    console.*log*(e)

  }

}

**Operador condicional ternario**

EL operador ternario nos permite simplificar el condicional if else. Colocando primero la condición seguro de un signo ? para indicar que código ejecutar en caso de que la condición se verdadera y : en caso de la condición sea falso. El operador ternario requiere siempre de un else.

*const* activo = true;

*//            Condicion   Verdadero  Falso*

*const* mensaje = activo ? ' Antivo' : 'Inactivo';

*const* mensaje = activo ? ' Antivo' : null;

También tenemos la forma reducida de ejecutar un if sin el else:

*//           condicion    Si es verdadero ejecuta*

*const* mensaje = activo && 'Activo'

**REACT**

Para crear un proyecto con React tenemos varias opciones algunas de ellas son:

* Escribimos en la terminal de comandos ubicándonos en donde queremos crear nuestro proyecto: npx créate-react- app nombre-de-mi-app
* con Vite: escribimos en el mismo lugar: npm créate vite
  + se instala vite
  + me pide le nombre del proyecto
  + me pregunta que tecnologías voy a utilizar
  + ingreso a la app creada y escribo en la consola: npm install para que se instalen las dependencias necesarias.

Componentes

Es un pequeño trozo de código encapsulado que realiza un trabajo en específico que puede tener un estado o no.

Estado

Es como se encuentra la información del componente en un punto determinado del tiempo.

Componentes

Para crear un componente Funcional de React debemos siempre importar primero React dentro del componente y el React Dom. Se recomienda nombrar los componentes siempre en CamellCase.

import React from 'react'

import ReactDom from 'react-dom/client';

Los componentes siempre deben retornar un único elemento html, el cual en su interior puede contener lo que yo desee.

Para poder usar un componente en otro componente o renderizarlo primero debo exportarlo

export *const* *App* = ()*=>*{

  return (<h1>Hi!!!</h1>);

}

Luego para poder usarlo dentro de otro componente de importarlo.

import { App } from './firstApp'

lo uso como si fuera una etiqueta Html común.

<*App*/>

Imprimir variables en Html

Para poder colocar variables o código JavaScript mesclado con el código html debo colocarlo entre { }.

*const* newMessage = 'Priscilla';

*const* *FirstApp* = () *=>* {

  return (

      <h2>{ newMessage }</h2>

  )}

Si quiero imprimir un objeto tal cual el código, primero debo serializarlo para que se pueda imprimir tal cual:

*const* newMessage = {

  mesagge: 'Hi worl',

  date: 'message'

};

return (

    <>

      <h2>{ JSON.*stringify*( newMessage )}</h2>

    </>

  )

Con la etiqueta <code> se ve mejor.

<>

      <code>{ JSON.*stringify*( newMessage )}</code>

    </>

*Reglas de React*

* “Sí tengo una variable la cual contiene un valor estático que no se modifica y no tiene ninguna relación con ningún Hooks o algo por el estilo, se recomienda colocarlo fuera del Funtional Component”

## Estilos Css

Para poder aplicar estilos globales a toda la aplicación debo importar la hoja css el el archivo principal por ejemplo main.jsx.

import './styles.css'

## Componentes y Props

Las Props son objetos que contienen información la cual se envía desde un componente (componente padre) a otro componente (componente hijo). Las Props (properties), se envían al momento de invocar al componente. Si deseo enviar números debo hacerlo entre {} para que los tome como números u no como string.

export *const* *App* = ()*=>*{

  return (

    <>

      <*First* title="I’m a title" number={123} *First*>

    </> ); }

En caso de que queramos establecer un valor por defecto en caso de que no nos envíen las Props debemos establecerlo de la siguiente manera. En el componente que va a recibir las Props.

*const* *FirstApp* = ( {*title* = 'i am a default value'} ) *=>* { … }

## Props Types

Nos sirve para poder definir qué tipo de Props esperamos recibir (tipar, sí trabajamos con JS), aceptando estrictamente solo lo que definimos, y también a definir cuando una Props es obligatoria o no.

Sí realizamos nuestro proyecto con CreateReactApp ya viene incorporado, sí lo hacemos con Vite debemos instalarlas manualmente.

Desde la terminal de comando colocándome en la carpeta de mi proyecto coloco

**Yarn:** yarn add prop-types

**Npm**: npm install prop-types

Luego las importo en mi componente:

import PropTypes from 'prop-types';

Las definimos de la siguiente manera, generalmente se definen luego del componente, es decir al final del mismo:

FirstApp.propTypes = {

  title: PropTypes.string.isRequired

}

Tip: Sí mando una Prop de la siguiente manera es igual a enviar un booleano en verdadero.

<*First* title ></*First*> 🡺 <*First* title={true} ></*First*>

## Default Props

Podemos definir los valores por defecto de las Props como mencionamos anteriormente

*const* *FirstApp* = ( {*title* = 'i am a default value'} ) *=>* { … }

o de la siguiente manera:

FirstApp.defaultProps = {

  title: 'no hay titulo'

}

## Eventos:

### Clic:

Para poder usar el evento clic se debe usar como un método que llama una función:

<button onClick={ *function*(){ } } > </button>

Debemos tratar de evitar escribir un código complicado de leer y trabajar con código limpio por lo que en el caso de los eventos es preferible crear la cunfion fuera del código html

*const* *handleAdd* = (*event*) *=>* {

  console.*log*( event )

}

export *const* *CounterApp* = ({*value*}) *=>* {

  return (

    <>

      <h1> CounterApp </h1>

      <h2> { value } </h2>

      <button onClick={ (*event*) *=>* *handleAdd*(event) }> + 1 </button>

    </>

  )

}

Podemos sintetizar el evento de la siguiente manera

<button onClick={ handleAdd }> + 1 </button>

Al llamar a la función handleAdd de esta forma se pasa automáticamente el evento como parámetro y se ejecuta, salvo que debamos pasarle a la función algún otro parámetro ajeno al evento en sí, en cuyo caso sí debería escribirse de la siguiente manera.

<button onClick={ (*event*) *=>* *handleAdd* (event, value)}> + 1 </button>

# Hook: useState

En los componentes de React lo primero que se debe importar son los Hooks.

import { useState } from 'react';

Los hooks son funciones especiales, las cuales inician con la palabra “use” para que cuando las miremos podamos reconocer automáticamente que es un Hook, por eso mismo cuando creemos nuestros propios Hooks su nombre debe iniciar con la palabra “use”, no es obligatorio, pero es lo recomendado.

El useState se desestructura con 2 variables: el primero es el nombre del estado que vamos a crear por ejemplo counter y el 2do es el nombre de la función que va a permitirnos modificar ese estado. Cuando se inicializa el estado usualmente se le debe pasar cual es el valor inicial del estado:

*const* [ counter, setCounter] = *useState*( 10 );

para modificar el estado debemos usar la función setCounter e indicarle dentro de los parámetros como queremos que la modifique.

*const* *handleAdd* = (*event*) *=>* {

*setCounter*( counter + 1)

  }

Cuando llamamos al setCounter le indicamos a React que el estado del componente se modificó por lo que se debe volver a renderizar dicho componente.

Desde la versión 18 de React, la renderización se ejecuta únicamente cuando se terminó de modificar el estado, por ejemplo, cuando se terminó de ejecutar, en este caso, handleAdd. Ta que dentro de la función handleAdd podemos tener la modificación de varios estados.

Cuando cambia el estado se vuelve a ejecutar TODO el componente.

Pruebas unitarias y de integración

En las pruebas buscamos que nuestra aplicación este lo más limpia posible y depurada antes de pasarla a producción para evitar contratiempos. Primero debemos probar la ruta crítica de la aplicación es decir desde el primer paso que daría el usurario de la app.

Hay 2 tipos principales de pruebas:

* Unitarias: Están enfocadas en pequeñas funcionalidades de la Aplicación.
* De Integración: Están enfocadas en cómo reaccionan varias piezas en conjunto.

### **Características**:

### Las pruebas deben ser:

1. Fáciles de escribir
2. Fáciles de leer
3. Confiables
4. Rápidas
5. Principalmente Unitarias

A.A.A:

* Arrange (Arreglar): Paso donde establecemos el estado inicial, el sujeto a probar:
* Inicializamos variables
* Realizamos las importaciones necesarias
* Preparamos el ambiente a probar
* Act. (Actuar): Aplicamos acciones o estímulos al sujeto de pruebas:
  + Llamamos métodos
  + Simulamos Clicks
  + Realizamos acciones sobre el paso anterior
* Assert (Afirmar): Observamos el comportamiento resultante. Verificamos si los resultados son los esperados.

**Configuración Testing en VITE**:

Para trabajar con Jest primero debemos instalarla: (https://jestjs.io/docs/getting-started)

yarn add --dev jest

npm install --save-dev jest

En el package.json debemos agregar el script de Jest;

"test": "jest"

Para ejecutarlo en la terminal debo colocar:

yarn Test

Si quiero que se corra contantemente y no tener que estar colocando yarn test todo el tiempo el script de ser:

"test": "jest --watchAll"

Si es el inicio y no he creado ninguna prueba me va a lanzar un error, pero es normal ya que no hemos iniciado.

Para inicial con los test debemos generar nuestro directorio donde trabajaremos con los tests. Para no hacer el SRC muy pesado podemos crear la carpeta Test fuera del SRC. Dentro de la carpeta de Test debería tener un espejo de mis componentes para poder probarlos, solo que en vez de ser .jsx será .test.js y contendrá los test.

Para que Jest reconozca el test debo llamar a la función:

*test*( ' Esta pueba no debe de fallar', ()*=>*{

  if( 0 === 0 ) {

  }

})

Donde el primer parámetro es el nombre de test y el segundo un Call Back donde realizamos la prueba que deseamos. Si no lanzamos un error Jest lo toma como que la prueba paso correctamente.

*test*( ' Esta prueba no debe de fallar', ()*=>*{

  if( 0 === 0 ) {

    throw new *Error*('este es mi error')

  }

})

Para poder ayudarnos con sintaxis de Jest y sus funciones podemos instalar con la terminal el intelisence de Jest: (--dev es igual a escribir –D, para indicarle que solo lo instale en la versión de desarrollo)

yarn add -D @types/jest

## Expect – toBe

Lo usamos para indicarle al test lo que esperamos (expect), exactamente lo que esperamos obtener (toBe)

*test*( ' Esta es mi primer prueba', ()*=>*{

*// 1. Inicializaación*

*const* message1= 'hola mundo';

*// 2. Estimulo*

*const* message2= message1.*trim*();

*// 3. Observar el comportamiento... esperado*

*// expect( message1 ).toBe( message2);*

*expect*( message1 ).toBe (message2);

})

Puedo agrupar las pruebas (describe), para indicar que pertenecen a un componente o que son de un determinado tipo de prueba en especial o que fueron creadas para un propósito en particular, etc:

*describe*('Este es mi primer grupo de pruebas', () *=>* {

*test*( ' Esta es mi primer prueba', ()*=>*{

*// 1. Inicializaación*

*const* message1= 'hola mundo';

*// 2. Estimulo*

*const* message2= message1.*trim*();

*// 3. Observar el comportamiento... esperado*

*expect*( message1 ).*toBe*( message2);

  })

})

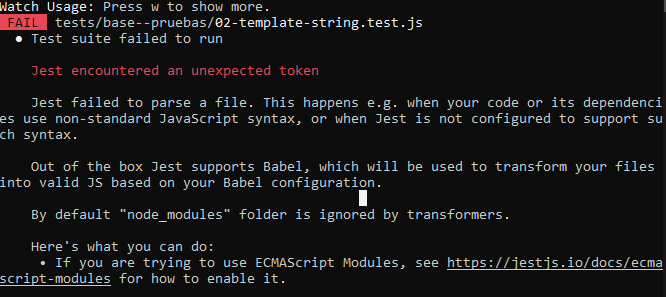
Para poder hacer alguna prueba de exportar si o si el archivo o función que deseo probar.

export *function* *getSaludo*(*nombre*) {

    return 'Hola ' + nombre;

}

Si al intentar correr un test me sale el siguiente error:



Debo instar las configuraciones de babel:

yarn add --dev babel-jest @babel/core @babel/preset-env

Luego debemos crear el archivo babel.config.js donde demos colocar el siguiente código:

*module*.*exports* = {

  presets: [['@babel/preset-env', {

    targets: {node: 'current'}

  }]],

};

Y volver a correr los test, si aun así persiste un error debo modificar el nombre de babel.config por babel.config.cjs.

## toEqual:

Si quiero comparar que el contenido de un objeto obtenido sea igual a otro objeto, debo usar “toEqual” o "toStrictEqual", ya que al comprar dos objetos distintos estoy apuntando a diferentes espacios en memoria, aunque los objetos sean exactamente iguales.

import { getUser } from "../../src/base--pruebas/05-funciones"

*describe*('Pruebas en 05-Funciones', () *=>* {

*test*('getUser debe retornar un objeto', () *=>* {

*const* testUser = {

      uid: 'ABC123',

      username: 'El\_Papi1502'

    }

*const* user = *getUser*();

*expect*( testUser ).*toEqual*( user );

  })

})

## Expectativa tipo de dato:

Si deseo evaluar el tipo de dato recibido ya sea una variable o por desestructuración de un arreglo puedo hacerlo de 2 maneras:

1. Mezclando Js con Jest.

*test*('Debe de retornar un string y un numero ', () *=>* {

*const* [letters, numbers] = *retornaArreglo*()

*expect*( typeof letters ).*toBe*( 'string');

*expect*( typeof numbers ).*toBe*( 'number' );

    })

2. Con Jest puro:

*test*('Debe de retornar un string y un numero ', () *=>* {

*const* [letters, numbers] = *retornaArreglo*()

*//            espera cualquier string/numero*

*expect*( letters ).*toEqual*( expect.*any*(*String*));

*expect*( numbers ).*toEqual*( expect.*any*(*Number*));

    })

Si deseo que el dato a evaluar sea un “undefined”, “false”, “null”, etc:

Puedo evaluarlo esperando un falso con Js o con Jest:

*test*('getHeroeById debe retornar undefined si no existe el heroe', () *=>* {

*const* id = 20;

*const* hero = *getHeroeById*( id );

    console.*log*(hero)

*expect*( hero ).*toBeFalsy*()

*expect*( hero ).*toBe*( undefined )

  })

## Promesas Jest:

Jest trabaja de forma síncrona, por lo que al encontrarse con una promesa en un test la ejecuta el código y sin esperar a una resolución, sigue hasta finalizar y luego lanza un error cuando la promesa termina de ejecutarse.

Para poder realizar test con promesas Jest nos da un parámetro (‘done’), que podemos utilizar y así hacer que funcione correctamente:

*test*('getHeroeByIdAsync debbe retornar un Heroe', (*done*) *=>* {

*const* id = 1;

*getHeroeByIdAsync*( id )

    .*then*( *hero* *=>* {

*expect*( hero ).*toEqual*( {

        id: 1,

        name: 'Batman',

        owner: 'DC'

      })

*done*()

    })

  })

Otra opción es utilizar el resolves y rejects de Jest:

*test*('getHeroeByIdAsync debe retornar un Heroe', () *=>* {

*const* id = 1;

    return *expect*( *getHeroeByIdAsync*( id )).resolves.*toEqual*( {

      id: 1,

      name: 'Batman',

      owner: 'DC'

    })

  })

*test*('getHeroeByIdAsync debe retornar un Error si el Héroe no existe', () *=>* {

*const* id = 100;

    return *expect*(*getHeroeByIdAsync*( id )).rejects.*toMatch*('No se pudo encontrar el héroe');

  })

También lo podemos usar con Async Await.

## Async Await

Para trabajar con Async Await en Jest es igual que en promesas.

*test*('getImagen debe retornar un URL de la imagen', async() *=>* {

*const* url = await *getImagen*();

*expect*( typeof url ).*toBe*('string');

  })

## Testing en componentes de React:

## React Testing Librery

React Testing Library es muy Bueno y útil para manejar el DOM virtual. Vamos a poder hacer simulaciones de clic en el dom, sin necesidad de ir nosotros a hacer clic en el DOM, entre otras cosas. Es una librería que está enfocada en lo que sucede en la pantalla, es decir en lo que sucede después de las interacciones. A diferencia de Jest que está orientada a hacer las aserciones y también a hacer ciertos moks de funciones que necesito evaluar, por ejemplo, verificar que algo llegue al Back End.

Si trabajamos con ‘Create React App’ ya viene con React-Test-renderer, pero se recomienda de igual modo, para probar el DOM, trabajar con React Testing Librery.

Para instalarlo debemos colocar en la consola estando en la raíz de nuestro proyecto:

yarn add --dev @testing-library/react

Si por algún motivo no se puede instalar probar eliminando el Node Modules e instalarlo.

Render: es una función que renderiza nuestro componente en memoria, la debemos importar de:

import { render } from "@testing-library/react"

Para poder hacer uso de Render debemos indicarle a Babel como lo debe renderizar, en el archivo ‘Jest.config.cjs’ debemos colocar dentro del module.exports:

*module*.*exports* = {

  testEnvironment: 'jest-environment-jsdom',

}

También debo instarlo en la terminal:

yarn add -D jest-environment-jsdom

En caso de que nos indique ‘Jest encountered an unexpected toked’ entonces debo modificar l archivo ‘babel.config.cjs’ y pasarlo de:

presets: [['@babel/preset-env', {targets: {node: 'current'}}]]

a:

presets: [

    [ '@babel/preset-env', { targets: { esmodules: true }}],

    [ '@babel/preset-react', { runtime: 'automatic'}],

  ],

Por ultimo debemos instalar ‘preset-react’:

yarn add –D @babel/preset-react

Si surge algún error bajamos el test y volvemos a correr.

El Render entre otras cosas actualiza el objeto screen de react Testing Librery.

Retorna un objeto que expone ciertas propiedades y métodos que podríamos utilizar, entre las propiedades se encuentra el *container* el cual se parece mucho a un nodo de HTML (Document Objectt Model).

### Verificación de un componente

Etiquetas y contenido:

Snapshot: La primera vez que se realiza/ejecuta el snapshot se toma una ‘fotografía’ del componente sobre el cual se ejecuta y lo guarda en una carpeta que crea \_\_\_snapshot\_\_ (nos guarda el código html que se ‘renderiza’ en memoria).

*const* { container } =  *render*( <*FirstApp* title={title}/> )

*expect* ( container ).*toMatchSnapshot*();

Esto sirve para que posteriormente compare el código que se debería renderizar, con lo que re renderiza en memoria según las modificaciones que vayamos realizando y así ver si las modificaciones afectan o rompen lo que debería renderizar. Si el renderizado debe cambiar, el snapshot se puede actualizar presionando en la consola de los test u.

*const* title= 'I am title';

*expect*( *getByText*( title )).*toBeTruthy*(); *// verifico que exista el texto 'I am title'*

Para poder verificar que un elemento se renderize con la etiqueta que quiero y con tenga el texto que deseo puedo utilizar el siguiente código (no es la forma recomendada, pero nos sirve para entenderlo. Más adelante veremos lo que es óptimo y recomendable).

*test*('debe mostrar el titulo en un H1', () *=>* {

*const* title= 'I am title';

*const* { container, getByText } =  *render*( <*FirstApp* title={title}/> )

*const* h1 = container.*querySelector*('h1');

*expect*( h1.innerHTML).*toBe*( title );

  })

El código anterior comprueba que le código sea extrictamente igual, por lo que si en la creación del componente (archivo componente.jsx), tenemos un espacio de más nos va a lanzar un error, por lo que, si solo nos interesa que se evalué el contenido podemos usar, (en reemplazo de toBe()):

*expect*( h1.innerHTML).*toContain*( title );

**getByTestId y otras Props:**

Nuestras pruebas deben ser flexibles, es decir que deben evaluar lo que nosotros queremos y a la vez deben ser flexibles.

Para poder buscar por los ‘data atributes’ un elemento en especial y verificar su contenido o lo que yo desee; para esto debo colocar en mi elemento hmtl en el archivo de mi componente.jsx:

<h1 data-testid='test-title'>{ title }</h1>

*/\* data atribute ==> data-testid*

*nombre o identificador del atributo ==> test-title\*/*

para poder utilizarlo debo extraerlo primero del render:

*const* { getByTestId } =  *render*( <*FirstApp* title={title}/> )

*expect*( *getByTestId*( 'test-title').innerHTML).*toBe*( title)

El getByText: no regresa solo un elemento que contenga el texto especificado, en caso de querer o tener varios elementos con el mismo texto buscado nos va a lanza un error por lo cual para tal caso debemos utilizar getAllByText, el cual nos devuelve un arreglo con todos los elementos que posean el texto indicado.

*const* {getAllByText} =  *render*(

      <*FirstApp* title={title} subtitle={subTitle} />

    )

*expect*( *getAllByText*(subTitle).length).*toBe*(3);

  })

### Screen y más props:

### es un método que debemos importar de React Testing Librery

import { render, screen } from "@testing-library/react"

Nos sirve como remplazo a los Snapshot. Ya que nos devuelve el html creado en memoria por el render.

Por lo que podemos sintetizar el código y trabajar de manera más limpia.

*render*( <*FirstApp* title={title}/> )

*expect*( screen.*getByText*( title )).*toBeTruthy*();

*render*( <*FirstApp* title={title}/> )

*expect*( screen.*getByRole*( 'heading', { leavel: 1 } ).innerHTML).*toContain*(title);

Screen.debug(), nos permite ver por consola el html creado por el render (siempre y cuando lo llamemos después del render si lo llamamos previo al render nos mostrará un <body> vacío.

screen.*debug*()

getByRole() nos permite encontrar elementos hmtl según su etiqueta por ejemplo los títulos son ‘heading’. Se le pasa como parámetros el nombre o rol de la etiqueta seguido de su nivel entre llaves.

*expect*( screen.*getByRole*( 'heading', { level: 1 } ).innerHTML).*toContain*(title);

Eventos:

Para poder trabajar con los eventos React Testing Librery nos ofrece el fireEvent, el cual debemos importar para poder usarlo.

import { fireEvent, render, screen } from "@testing-library/react"

### Click: para poder detectar el evento click usamos fireEvent. El evento click debe recibir como parámetro el elemento al cual se desea escuchar.

fireEvent.*click*(screen.*getAllByText*('+1'))

Luego de definir mi elemento click puedo, establecer el expect ya que se va a ejecutar lego de realizar el evento de forma ficticia.

fireEvent.*click*(screen.*getByText*('+1'));

*expect*(screen.*getByText*('1')).*toBeTruthy*();

v

video 69 min 6