Pruebas unitarias y de integracion - Probando las secciones anteriores

¿Qué veremos en esta sección?

Introducción a las pruebas

AAA

- Arrange – Arreglar
- Act – Actuar
- Assert – Afirmar

Primeras pruebas

Jest

Expect

toBe

Enzyme

Comandos útiles en la terminal para pruebas

Revisar elementos renderizados en el componente

Simular eventos

Esta sección de pruebas es sumamente importante porque nos dará la base de las pruebas que estaremos haciendo durante el curso, las pruebas irán creciendo en complejidad, por lo que les recomiendo que nos aseguremos de comprender bien todos estos conceptos.

1. Introducción a las pruebas unitarias y de integracion

¿Qué son las pruebas?

No son una pérdida de tiempo. Nos sirve para mejorar la experiencia de usuario.

Hay dos tipos principales de pruebas. Unitaria y de integración. Las unitarias estan enfocadas en pequeñas funcionalidades y las de integracion estan enfocadas en cómo reaccionan varias piezas en conjunto.

Característias de las pruebas:

- Fáciles de escribir
- Fáciles de leer
- Confiables
- Rápidas
- Principalmente unitarias

Las pruebas de integracion deberían confirmar la interaccion de los pequeños componentes probados de manera unitaria.

AAA

- Arrange: Preparamos el estado inicial del elemento a probar.
- Act: Aplicamos acciones o estímulos al elemento.
- Asserts: Observar el comportamiento resultante. Comprobar que los resultados sean los esperados.

Mitos

Hacen que mi aplicación no tenga errores: Estan hechas por personas, pueden fallar.

Las pruebas no puden fallar: Podemos tener falsos-positivos o falsos-negativos.

Hacen mas lenta mi aplicación: No llegan a producción, solo corren en la maquina de desarrollo, son locales.

Es una pérdida de tiempo: En parte si lo son, pero solo si hacemos pruebas de cosas que no tienen sentido probar. Por ejemplo probar una libreria hecha por otra persona.

Hay que probar todo: Puede tomar tanto tiempo o más que lo que tardamos en crear la aplicación. Por lo tanto a final de cuentas hay que probar la ruta critica y si hay tiempo otras característias.

2. Inicio de la sección - Pruebas sobre lo aprendido anteriormente

Vamos a utilizar el recurso dado por el instructor. Son ejercicios realizados en la sección de JavaScript.

3. Mi primera prueba

Una manera de estructurar los directorios y los archivos de pruebas es generar una carpeta test y dentro los archivos DEBEN tener la extension .test.js ya que es lo que va a buscar el test suit para correr las pruebas.

Con una terminal vamos a ir al directorio donde se encuentra nuestro .test.js y vamos a ejecutar npm run test, por default va a ejecutar todos los test que encuentre a partir de la raíz. Si hay un unico test lo puedo hacer desde la raíz, si hay mas de un test pero separados en distintas carpetas puedo ingresar a una de esas carpetas y ejecutarlo. Si tengo muchos test en una misma carpeta padre debo poner el nombre. Por default va a buscar los .test.js asi que con solo el nombre me alcanza.

Tambien puedo utilzar los snippets del a suit para filtrar las pruebas a partir de la raíz donde me encuentro.

El nombre "test" puede ser renombrado dentro del JSON de package en la seccion de scripts, cambiandolo por otro nombre. Realmente estamos ejecutando ese script que viene precargado con configuraciones basicas/standard para realizar pruebas con JEST (libreria muy utilizada para pruebas en react que viene por defecto al hacer npx create-react-app my-app). El JEST lo podemos mosificar si no queremos trabajar con las configuraciones por default.

Las hay activo un comando de pruebas al guardar cambios en el test ejecutado este se vuelve a lanzar.

En general dentro de la carpeta test deberiamos tener el nombre de los directorios para simular donde estan contenidos realmente.

Una prueba basica es utilizar dentro de nuestro .test.js el metodo test del JEST ("NOMBRE_QUE_ESPERA", CALLBACK);

ejemplo:

test ('Esta es mi primera prueba - debe ser true', ()=>{});

Si queremos probar una funcion debemos exportarla para poder ejecutarla dentro del test.

3. Jest - Expect - toBe

Documentación: https://jestjs.io/ - https://jestjs.io/docs/expect

Puedo agrupar mas pruebas con el describe.

Ejemplo:

```
describe('Pruebas en el archivo NOMBRE', () => {
   test ('NOMBRE PRUEBA - QUE ESPERA', ()=>{
      //Arrange
      const mensaje = "COMPARACION1";
      //Act
      const mensaje2 = `COMPARACION2`;
      //Assert
      expect(mensaje).toBe(mensaje2);
   });

   TEST('OTRO TEST'), () =({
      //AAA
   });
});
```

4. Pruebas asincronas

Documentación: https://jestjs.io/docs/asynchronous

Para realizar un test asincrono podemos pasarle por parametros el callback done. El done le va a decir a mi JEST/Test Suit cuando debe de terminar la prueba y que espere a su ejecucion. La idea del done es ejecutarlo una vez terminada la prueba asincronica.

Si bien en el curso muestra como usar el done en una prueba que devuelve una promesa la documentación aclara que el done debe utilizarse en caso de ejecutar un callback, y si utilizamos promesas utilizar pruebas asincronas. Leer la documentacion.

5. Pruebas sobre componentes de React

https://developer.mozilla.org/es/docs/Glossary/Wrapper

6. React Testing Library

Enzyme fue discontinuado, vamos a seguir con la siguiente libreria: https://testing-library.com/docs/react-testing-library/intro/ - Viene por defecto en React 18.

Un snapshot es una fotografia (JSON) del componente renderizado. Las snapshots son muy utiles ya que nos permiten saber que hay dentro de un componente y su estructura.

Las snapshots son muy utiles ya que si, por ejemplo, hicimos cambios en el componente que no debian hacerse y que nos salte un error ya que para renderizarlos debiamos tener cierto comportamiento. Si testeamos el componente cuando lo hicimos, tiempo despues volvemos a testearlo y la snapshot falla quiere decir que a ese componente le hicimos cambios en algun punto.

Al recuperar un int de un snapshot este se devuelve como un string.

7. Simulacion de botones

https://testing-library.com/docs/guide-events/

Recuperamos el html y le pasamos la siguiente nomenclatura: fireEvent.EVENTO(HTML);