**ENTREVISTA**

**JAVASCRIPT**

**Que el DOM =>** Document Objet Model => Es la estructura del html diagramada como arbol.

A partir de este DOM se crea la estructura de la pagina.

A través del DOM, se puede acceder y modificar el contenido, estructura y estilo de los documentos mencionados, que es para lo que se diseñó principalmente.

**Temas de accesibilidad =>** La accesibilidad hace que tus sitios web puedan ser utilizados por el mayor número de personas posible. Tradicionalmente, se ideó para las personas con discapacidad, pero la creación de sitios accesibles también beneficia a otros grupos, como los que utilizan los dispositivos móviles o los que tienen conexiones de red lentas.

**Estilos =>** SASS, CSS, BOOTSTRAP.

**Nomenclatura bem =>** (Bloque, elementO, modificador) una metodología de nomenclatura para definir las clases en los nodos HTML del documento => ejemplo

<div class="block">

<div class="block\_\_element">Elem 1</div>

<div class="block\_\_element">Elem 2</div>

<div class="block\_\_element block\_\_element--modifier">Elem 3</div>

</div>

**Nomenclatura camelCase =>** Es otra forma de escribir las clases dentro de CSS. Utilizandose por ejemplo asi => class = 'containerPrincipal'

**Pixel Perfect =>**

**Pre-procesadores de CSS** => es un programa que te permite generar CSS a partir de la syntax (en-US) única del preprocesador. Ejemplo SASS, LESS. Permite añadir algunas características que no existen en CSS puro, como variable, mixins, selectores anidados, entre otros.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***JAVASCRIPT**

**Diferencia entre una variable y constante** =>

**LET**

**let puede modificarse pero no volver a declararse.**

let a = 20;

let a = 10

// a ya fue declarada

let a = 20

a = 10

// se puede hacer

Sin embargo, si la misma variable se define en diferentes ámbitos, no habrá error

**let tiene un ámbito de bloque => Un bloque es un trozo de código delimitado por {}.**

let saludar = "dice Hola";

let tiempos = 4;

if (tiempos > 3) {

let hola = "dice Hola tambien";

console.log(hola);// "dice Hola tambien"

}

console.log(hola) // hola is not defined

**Hoisting de let**

Las variables declaradas con let y const también reciben hoisting, pero no son inicializadas con un valor por defecto. Acceder a una variable declarada con let o const antes de que sea declarada resulta en un ReferenceError:

console.log(foo); // Uncaught ReferenceError: Cannot access 'foo' before initialization

let foo = 'bar'; // lo mismo para variables declaradas con const

**VAR**

**Las variables declaradas con var se pueden volver a declarar y actualizar**

var mensaje = " mensaje";

var mensaje = "mensaje declarado de nuevo y actualizado";

console.log(mensaje) => // "mensaje declarado de nuevo y actualizado"

**y esto tambien**

var saludar = "hey, hola";

saludar = "dice Hola tambien";

console.log(saludar) => // "dice Hola tambien"

**Si la variable es declarada fuera de una función, la variable será una variable global. Si es declarada dentro de la funcion solo tiene alcance dentro de esa funcion**

var mensaje = "soy una variable con alcance global";

function newFunction() {

var mensajefuncion = "soy una variable con alcance de función";

}

console.log(mensajefuncion ); // error: mensajefuncion is not defined

**Hoisting de variables con var**

Cuando el intérprete hace hoisting de una variable declarada con var, inicializa su valor a undefined. La primera línea de código a continuación muestra undefined:

console.log(foo); // undefined

var foo = 'bar';

console.log(foo); // "bar"

Recuerda que la primera console.log(foo) muestra undefined porque a la variable foo se le hace hoisting y se le asigna un valor por defecto (no porque la variable nunca sea declarada).

El uso de una variable no declarada mostrará un ReferenceError en lugar de un undefined:

console.log(foo); // Uncaught ReferenceError: foo is not defined

**CONST**

**Cada declaración con const debe inicializarse en el momento de la declaración.**

const nombre;

nombre = 'jona' // Error

**Las declaraciones const tienen un ámbito de bloque**

**Const no se puede actualizar o volver a declarar**

const mensaje = " hola estas en estrada web group ";

mensaje = "hola de nuevo estas en estrada web group "; // error: Assignment to constant variable.

Esto también marcara error:

const mensaje = " hola estas en estrada web group ";

const mensaje = "hola de nuevo estas en estrada web group "; // error: Identifier 'greeting' has already been declared

**// Este comportamiento es algo diferente cuando se trata de objetos** declarados con const. Mientras que un objeto const no se puede actualizar, las propiedades de este objeto sí se pueden actualizar. Como resultado, si declaramos un objeto const como este:

Ejemplo =>

**const array** = [1,2,3] ---- array.push(4) ---- console.log(array) => [1,2,3,4]

**const saludar = {**

mensaje: "dice Hola",

tiempos: 4

**}**

**saludar.mensaje = "dice Hola tambien"; /// se puede hacer**

**saludar = {**

palabras: "Hola",

numero: "cinco**"**

**} // error: Assignment to constant variable.**

**Hoisting de const**

Las variables declaradas con let y const también reciben hoisting, pero no son inicializadas con un valor por defecto. Acceder a una variable declarada con let o const antes de que sea declarada resulta en un ReferenceError:

console.log(foo); // Uncaught ReferenceError: Cannot access 'foo' before initialization

const foo = 'bar'; // lo mismo para variables declaradas con const

**HOISTING**

Es el alzamiento o levantamiento de la declaracion de la variable al ambito de aplicacion.

Tanto VAR, LET, CONST y las funciones tienen hoistin pasa que los mismo van a ir variando

Ejemplo con funciones =>

function nombreDelGato(nombre) {

console.log("El nombre de mi gato es " + nombre);

}

nombreDelGato("Maurizzio");

/\*

El resultado del código es: "El nombre de mi gato es Maurizzio"

\*/

Otro ejemplo =>

nombreDelGato("Dumas");

function nombreDelGato(nombre) {

console.log("El nombre de mi gato es " + nombre);

}

/\*

El resultado del código es: "El nombre de mi gato es Dumas"

\*/

**SCOOPE**

Ambito de alcance que tienen las variables.

Existen dos tipos de scope:

Global: en él se encuentran las variables declaradas fuera de una sentencia bloque o una función;

Local: en él se encuentran las variables declaradas dentro de una sentencia bloque o una función. Una sentencia bloque es una porción de código delimitada por {}, como puede ser un bucle if, for, while, etc;

Utilizando las palabras clave **let y const** podremos declarar variables cuyo alcance será la sentencia bloque en la cual se encuentren declaradas.

Por otra parte, el alcance de las variables declaradas con **var** será al nivel de función. Por lo tanto, aquellas que no sean declaradas dentro de una función, serán variables globales.

var a = true; // global

let b = 2; // global

const c = 'Esta es una constante'; // global

if (a === true) {

const d = 10; // local de bloque

let e = 20; // local de bloque

var f = 30; // global => porque no esta dentro de una funcion

}

function probarScope () {

const g = new Date(); // local de bloque-función

let h = [1, 2, 3]; // local de bloque-función

var i = { valor: 'prueba' }; // local de función

}

**CLOSURE**

Una clausura permite acceder al ámbito de una función exterior desde una función interior.

Es cuando una funcion interna (mostrarNombre) puede acceder a valores de la funcion externa (function iniciar)

function iniciar() {

var nombre = "Mozilla"; // La variable nombre es una variable local creada por iniciar.

function mostrarNombre() { // La función mostrarNombre es una función interna, una clausura.

alert(nombre); // Usa una variable declarada en la función externa.

}

mostrarNombre();

}

iniciar();

La función iniciar() crea una variable local llamada nombre y una función interna llamada mostrarNombre(). Por ser una función interna, esta última solo está disponible dentro del cuerpo de iniciar(). Notemos a su vez que mostrarNombre() no tiene ninguna variable propia; pero, dado que las funciones internas tienen acceso a las variables de las funciones externas, mostrarNombre() puede acceder a la variable nombre declarada en la función iniciar().

**CALLBACK**

Un callback (llamada de vuelta) es una función que recibe como argumento otra función y la ejecuta. Los callbacks de por sí son síncronos.

Ejemplo => setTimeout(() => console.log("He ejecutado la función"), 2000);

Las puedo utilizar con los hooks de REACT

**PROMESAS**

El objeto Promise representa la eventual finalización (o falla) de una operación asincrónica y su valor resultante.

Recibe dos argumentos que son CALLBACK => resolve, reajeat

Una promesa puede presentar los siguientes estados:

fulfilled - La acción relacionada a la promesa se llevó a cabo con éxito

rejected - La acción relacionada a la promesa falló

pending - Aún no se ha determinado si la promesa fue fulfilled o rejected.

El método .then() toma hasta dos argumentos; el primer argumento es una función de devolución de llamada (callback) para el caso resuelto de la promesa, y el segundo argumento es una función de devolución de llamada para el caso rechazado.

**El then, catch y el finally() que se usa posterior es para saber como reacionar una vez que la promesa se termine.**

**Para hacerlo se utiliza esto =>**

1) promesa.then

El then() se ejecuta cuando la promesa se ejecuto correctamente

2) promesa.catch()

El catch() se ejecuta cuando la promesa se ejecuto en forma erronea

3) promesa.finally()

El finally() se ejecuta de todas formas ya sea por haber sido ejecutado por el then() o por el catch()

const promesa = new Promise((resolve, reajeat) => {

const productos = data ====> esta data viene desde afuera

if(productos) {

resolve(productos)

} else {

rejeat('no se puedo encontrar el produtos')

}

})

Aca la informacion del resolve (data) es transferida al then que la resive en resp

promesa.then((resp) => {

setProductos(resp) =====>>> setProductos es un estados

})

promesa.catch((err) => console.log(err))

Esto es un ejemplo usando una funcion

**FETCH**

Es una API que se utiliza para recuperar recursos.

Sirve para hacer peticiones http y viene ya incorporada en los navegadores por ende no hace falta instalar nada.

El método fetch() toma un argumento obligatorio, la ruta de acceso al recurso que desea recuperar => url.

**Devuelve una Promise (en-US) que resuelve en Response a esa petición, sea o no correcta.**

Una vez que Response es recuperada, hay varios métodos disponibles para definir cuál es el contenido del cuerpo y como se debe manejar.

EJEMPLO =>

fetch('http://example.com/movies.json')

.then(response => response.json())

.then(data => console.log(data));

.catch((err)=> console.log(err))

1) En la imagen uno como devuelve un array de esta forma puede utilizarse

2) En la imagen dos como la data es un objeto hay que desestructurarlo

**ASYNC - AWAIT**

Sirve para trabajar con procesos asyncronicos y es una forma mas limpia de trabajar con promesas.

Esta funcion que se hace con async retorna una promesa. El await no ayuda a trabar con este proceso asyncrono

Se puede usar aync sin await y al reves? =>

**El async puede estar independiente. Pero cuando ponemos el await no puede estar solo SI o SI necesita estar con el async.**

Al ponerle el await espera a que se resuelva esa peticion y luego sigue con la siguiente linea de codigo

Para manjear los ERRORES en aync-await es necesario trabajar con try - catch

Cual es mejor de utilizar Promesas o Async - Await? => Desde mi punto de vista el codigo escrito con async-await es mucho mas limpio que el codigo escrito con promesas.

**CLASES VS PROTOTIPOS**

**LOCAL-STORAGE y SESSION-STAROGE**

**Local-Storage y Session-Storage =>** la única diferencia es que la información almacenada en localStorage no posee tiempo de expiración, por el contrario la información almacenada en sessionStorage es eliminada al finalizar la sesion de la página.

**Cookies =>** Tambien sirve para alamacenar datos. Pero guarda menos informacion.

**REACT JS**

**Todos los Hooks =>** [**https://es.reactjs.org/docs/hooks-reference.html#uselayouteffect**](https://es.reactjs.org/docs/hooks-reference.html#uselayouteffect)

**Que es React.js?**

Es una libreria que se enfoja en las vistas y dentro de React podemos agregar otras libreria.

Sirve para hacer todo tipo de aplicaciones web, SPA o aplicaciones moviles.

Es desarrollado por Facebook de Software libre.

No es una biblioteca ya que no nos da un marco de trabajo => Ejemplo AngularJs.

**Comunicacion entre componentes - Props =>** Se traspasa de padres a hijos. Pueden pasarse de hijos a padres pero ya por estados o funciones.

Que es? Es una libreria NO un framework

Diferencia entre libreria y framework?

Virtual DOM?

Nuevas actualizaciones?

Algoritmo que utiliza?

PROPS VS STATE? Cual utilizar?

**Que es un Componente?**

Estructura de codigo pequena que permite dividir en pequenos pedacitos nuestra web y que hace que los mismos los hace reutilizables.

**Que es JSX?**

Forma de escritura de React.

**Hook => Que son?**

Son funciones que te permiente enganchar el estado de React y el ciclo de vida desde componentes de funcione.

**UseState => Que son?**

Son funciones que sirven para manipular el dom, un estado determinado

Siempre hay que inicializarlo de la forma en que es el estado inicial => Es decir si el estado inicial es un array, al momento de setearlo hay que ponerle un array [].

Ejemplo =>

const estadoInicial = [{id:1, nombre: 'Jona'}]

const [initial, setInitial] = useState(estadoInicial)\*

const ejemplo = () => {

setInitial([

...initial,

{id:2, nombre: 'Pepe'}

])

}

Ahi seria el ejemplo en donde setInitial tiene el array al igual que el estado inicial\*.

Hay una excepcion, salvo, por ejemplo, que el estado inicial sea un ARRAY en cuyo caso el Set no recibiria un ARRAY, sino un Set(value).

**UseEffect => Que es?**

Es un hook de react.

Efecto secundario => Algo cambio, dispara esta funcion. El componente se renderizo por primera vez, dispara esta funcion.

React cuando descubre un cambio lo vuelve a renderizar. Lo que permite, UseEffectu nos permite ejecutar lo que esta dentro del useEffect una unica vez.

Tiene como parametro un array de dependencia. Si esta vacio se ejecuta una vez al momento de construccion del componente. Si esta completo se ejecuta cuando lo que esta dentro de ese array de dependencia cambia.

**UseRef =>** Sirve para mantener una referencia y esa referencia va a permitir evitar un renderizacion

**UseLayoutEffect =>** Se dispara una vez producido TODAS las renderizacion del componete. Se utiliza para leer el diseno del dom (ejmplo el ancho y el alto)

**Memo =>** No es un hooks. Se trabaja con react y es muy similar a su hooks useMemo.

Es recomendado cuando los componentes son muy grandes.

Memo se importa de react y va a guardar determinados valores y solo se va a volver a ejecutar cuando su property cambia. Sino, no se ejecuta

import React from 'react';

// import { memo } from 'react';

export const Small = React.memo(({ value }) => {

console.log(' Me volví a dibujar :( ');

return (

<small>{ value }</small>

)

})

**useMemo =>** Este ya es un hooks. Memoriza un valor. Recibe un callback (funcion) y y el segundo parametro es un array de dependencia que lleva el valor de cuando ese valor cambie

**useCallback => Sirve para memorizar funciones** y regresa es una funcion memorizada. Primer parametro un callback y segundo una dependencia. Hay un ejemplo claro en los curso de Fernando Herrera

**Reducer** => Es una funcion comun y corriente.

Debe de ser una funcion pura => No debe de tener efectos secundarios (resolver internamente todo y no llamar a una funcion externa).

No debe de realizar tareas asincronas. No debe de llamar localStorage o sessionStorage.

Debe de retornar un nuevo estado.

Usualmente recibe dos argumentos (valor inicial y la accion a ejecutar).

La accion es como vamos a cambiar ese estado inicial.

Siempre debe de retonar un estado

**useReducer =>** Es una alternativa al useState. Nos ayuda cuando el estado inicial es mas complejo.

const [state, dispatch] = useReducer(reducer, initialArg, init)

**VITE**

Se puede crear de dos formas el entorno de desarrolo =>

**1)** npm create vite@latest

2) a) yarn (o npm) create vite (te pide datos a completar y nombre del proyecto.

b) Dentro del proyecto yarn (o npm install) (asi se instalan las dependencias)).

c) npm run dev - Copio el url y listo.

**INSTALACION Y CONFIGURACION DE JEST + REACT TESTING LIBRARY**

En proyectos de React + Vite - Si trabajariamos con create-react-app no haria falta

Instalaciones:

npm add --dev jest babel-jest @babel/preset-env @babel/preset-react

npm add --dev @testing-library/react @types/jest jest-environment-jsdom

Opcional: Si usamos Fetch API en el proyecto:

npm add --dev whatwg-fetch

Actualizar los scripts del package.json

"scripts: {

...

"test": "jest --watchAll"

Crear la configuración de babel babel.config.js

module.exports = {

presets: [

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

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

],

};

*Opcional, pero eventualmente necesario, crear Jest config y setup:*

1) jest.config.js

module.exports = {

testEnvironment: 'jest-environment-jsdom',

setupFiles: ['./jest.setup.js']

}

2) jest.setup.js

// En caso de necesitar la implementación del FetchAPI

import 'whatwg-fetch'; // <-- yarn add whatwg-fetch

**Para correrlo => npm test dentro del aplicativo, apreto w y despues p y ahi escribo donde quiero hacer las pruebas => Ejemplo, escribo => GifItem**

Hay que crear un carpeta tests dentro del directorio del aplicatavio (mismo nivel del src). Generalmente lo que se hace es crear esa carpeta test y ahi adentro se hace la misma esctructura del proyecto. Por ejemplo => dentro de test se crea una carpeta components y dentro de ahi los arhivos de test con el mismo nombre => ejemplo GitItem.test.jsx

**. Si quiero avaluar con prop-types** => Primero tengo que agregarlo con npm add prop-types

. Importamos en el componentes a evaluar =>

import PropTypes from 'prop-types'

export const GifItem = ({ title, url, id }) => {

return (

<div className="card">

<img src={ url } alt={ title } />

<p>{ title }</p>

</div>

)

}

. GifItem.prototype = {

title: PropTypes.string.isRequired,

url: PropTypes.string.isRequired

}

**. Para probarlo => Vamos a la carpeta tests**

Creamos la carpeta components y ai adentro GifItem.test.jsx

Dentro de ese archivo =>

import {render} from '@testing-library/react';

import {GifItem} from '../../src/components/GifItem'

describe('Pruebas en <GifItem/>', ()=> {

const title = 'OnePunche'

const url = 'http://'

test('debe de hacer match con el snapshot', () => {

const {container} = render(<GifItem title = {title} url = {url}></GifItem>)

expect(container).toMatchSnapshot();

})

})