Instalaciones necesarias

Visual Studio Code

Extensiones VS Code

- **Simple React Snippets**
- **Auto Close Tag**

Google Chrome

Extensiones Chrome

- **React Developer Tools**

NodeJS

En powershell instalar créate-react-app, ejecutar

~~npm install -g create-react-app~~

Instalar Next.js

npx create-next-app@latest

Crear una nueva app, ejecutar

~~npx create-react-app nombre-de-tu-aplicación~~

npx create-next-app@latest nombre-de-tu-aplicación

Instalar react-html-parser

npm install react-html-parser --legacy-peer-deps

Instalar classNames

npm install classnames

Iniciar la app, ejecutar

npm run start

npm run dev

Git

JavaScript Moderno

El "JavaScript moderno" se refiere a las características y funcionalidades introducidas en las versiones más recientes de ECMAScript. A partir de ECMAScript 6 (ES6), lanzado en 2015, se han agregado muchas mejoras y nuevas características que hacen que el desarrollo en JavaScript sea más eficiente y potente.

ECMAScript

ECMAScript es un estándar de scripting que define las reglas, tipos, estructuras y componentes del lenguaje. ECMAScript es importante porque asegura que diferentes entornos y navegadores interpreten el código JavaScript de manera consistente.

ECMAScript 6, también conocido como ES6 o ECMAScript 2015, es una versión del estándar ECMAScript que introdujo una serie de características y mejoras significativas al lenguaje JavaScript. Algunas de sus principales novedades incluyen:

Declaraciones de variables: Introducción de let y const, que permiten declarar variables de alcance de bloque y constantes, respectivamente, mejorando la gestión de variables.

Template literals: Permiten crear cadenas de texto de manera más flexible y legible.

Características de los template literals:

- **Interpolación de variables:** Puedes insertar variables y expresiones dentro de las cadenas usando la sintaxis ${expresión}.
- **Multilinea:** Permiten crear cadenas de varias líneas sin necesidad de concatenar.
- **Uso de comillas invertidas:** Se definen usando comillas invertidas (backticks) ` en lugar de comillas simples ' o dobles ".

Funciones de flecha: Una nueva forma de definir funciones que permite una sintaxis más concisa y mantiene el contexto de this del entorno donde se definen.

Clases: ES6 introduce la sintaxis de clases, facilitando la creación de objetos y la herencia, lo que hace que el código orientado a objetos sea más limpio y fácil de entender.

Parámetros por defecto: Los parámetros por defecto permiten asignar un valor predeterminado a un parámetro de una función si no se proporciona un argumento al llamarla.

Parámetros rest: Los parámetros rest permiten representar un número variable de argumentos como un array. Se usa el operador ... antes del nombre del parámetro.

Operador spread: El operador spread (...) en JavaScript se utiliza para expandir elementos de un iterable (como un array o un objeto) en lugares donde se esperan múltiples elementos.

Destructuración: Una forma sintáctica que permite extraer valores de arrays o propiedades de objetos de manera más sencilla.

Módulos: Permite la creación de módulos que pueden exportar e importar funciones, objetos o variables, ayudando a organizar mejor el código y mejorar la reutilización.

Promesas: Facilitan el manejo de operaciones asíncronas, permitiendo encadenar operaciones y manejar errores de manera más clara.

API Fetch

La API Fetch es una interfaz moderna y más poderosa para realizar solicitudes HTTP en el navegador. Proporciona un método para hacer peticiones a recursos remotos, y utiliza promesas para manejar respuestas asíncronas, lo que facilita trabajar con operaciones de red.

Características Principales de la API Fetch

Sintaxis Sencilla: La API es más fácil de usar que XMLHttpRequest, con una sintaxis más limpia y moderna.

Promesas: Utiliza promesas, lo que permite encadenar .then() y .catch() para manejar respuestas y errores de manera más clara.

Soporte para JSON: La API facilita la conversión entre objetos JavaScript y JSON, ya que ofrece métodos para manejar respuestas en formato JSON.

Configuración Flexible: Permite configurar solicitudes (método, cabeceras, cuerpo, etc.) de manera sencilla.

ES8 async – await

async: Se usa para declarar una función asíncrona. Dentro de esta función, puedes usar await.

await: Se utiliza para esperar a que una promesa se resuelva. Esto hace que el código se ejecute de forma más secuencial, como si fuera sincrónico

¿Qué es React?

React es una biblioteca de JavaScript para construir interfaces de usuario, especialmente para aplicaciones web de una sola página. Fue desarrollada por Facebook y se utiliza para crear componentes reutilizables que manejan su propio estado. React permite que los desarrolladores creen aplicaciones rápidas y eficientes al actualizar solo las partes de la interfaz que han cambiado, en lugar de volver a renderizar toda la página. Además, utiliza un concepto llamado "**Virtual DOM**", que optimiza las actualizaciones al minimizar el acceso al DOM real del navegador. Es popular por su flexibilidad, rendimiento y gran ecosistema de herramientas y bibliotecas complementarias.

¿Qué es el virtual DOM de React?

El Virtual DOM (DOM virtual) es una representación en memoria del DOM real que utiliza React para optimizar el rendimiento de las aplicaciones. Funciona de la siguiente forma:

1. Representación en memoria: Cuando creas componentes en React, estos generan un Virtual DOM que refleja la estructura de la interfaz de usuario. Este Virtual DOM es un objeto que se encuentra en la memoria y es mucho más rápido de manipular que el DOM real.
2. Actualización eficiente: Cuando hay cambios en el estado de un componente, React primero actualiza el Virtual DOM en lugar del DOM real. Luego, compara el nuevo Virtual DOM con el anterior mediante un proceso llamado "reconciliación".
3. Diferencias: React identifica las diferencias entre los dos DOMs y determina cuáles son los cambios necesarios para que el DOM real se actualice. Esto significa que solo se realizan las modificaciones necesarias en el DOM real, lo que reduce la cantidad de operaciones de acceso al DOM.
4. Actualización del DOM real: Finalmente, React aplica esos cambios de manera eficiente al DOM real, lo que resulta en una actualización más rápida y fluida de la interfaz de usuario.

Este enfoque ayuda a mejorar el rendimiento de las aplicaciones, especialmente en aquellas con muchos cambios de estado y renderizados dinámicos.

JSX

JSX (JavaScript XML) es una sintaxis utilizada en React que permite escribir estructuras de interfaz de usuario en un formato similar a HTML dentro de JavaScript. Aunque no es obligatorio usar JSX, facilita la creación de componentes visuales de una manera más intuitiva y legible.

Ejemplo:

```
const MiComponente = () => {
  return <h1>Hola desde MiComponente!</h1>;
};
```

Componentes

En React, un componente es una unidad reutilizable de código que define parte de la interfaz de usuario (UI). Los componentes son fundamentales para construir aplicaciones React, ya que permiten dividir la UI en piezas más pequeñas y manejables.

Tipos de Componentes

Componentes de Clase:
Son definidos usando la sintaxis de clases de JavaScript. Pueden tener estado y métodos de ciclo de vida.

Ejemplo:

```
import React, { Component } from 'react';
class MiComponente extends Component {
   render() {
      return <h1>Hola desde un componente de clase</h1>;
   }
}
```

Componentes Funcionales:
Son funciones de JavaScript que devuelven JSX. A partir de React 16.8, pueden usar hooks para manejar estado y efectos secundarios.

Ejemplo:

```
import React from 'react';
const MiComponente = () => {
   return <h1>Hola desde un componente funcional</h1>;
```

};

Ciclo de vida de un componente

En React, los componentes tienen un ciclo de vida que se puede dividir en diferentes fases: montaje, actualización y desmontaje.

Componente de clase
Principales métodos del ciclo de vida de un componente de clase:

Montaje
Estos métodos se llaman cuando el componente se crea e inserta en el DOM.

constructor(props): Se utiliza para inicializar el estado y enlazar métodos.

render(): Este método es obligatorio y devuelve el JSX que representa la UI del componente.

componentDidMount(): Se invoca inmediatamente después de que un componente se monta. Es un buen lugar para realizar llamadas a API.

Actualización
Se producen cuando hay cambios en las props o el estado del componente.

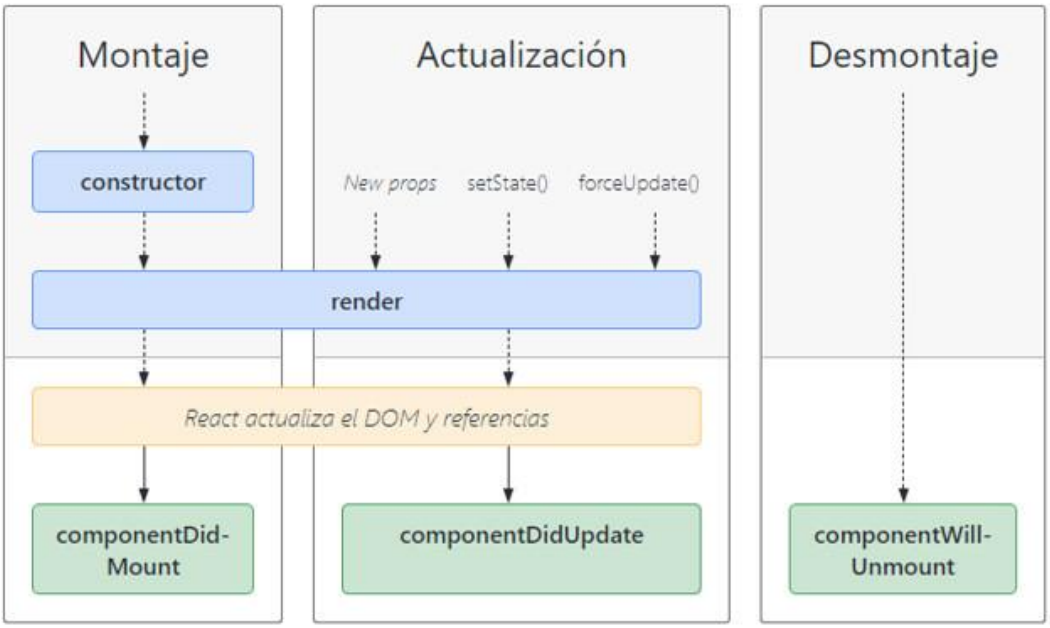
shouldComponentUpdate(nextProps, nextState): Permite optimizar el rendimiento, ya que se puede usar para evitar un renderizado innecesario. Debe devolver true o false.

render(): Se vuelve a invocar para renderizar el componente.

componentDidUpdate(prevProps, prevState, snapshot): Se invoca inmediatamente después de que se actualiza el componente. Es útil para realizar operaciones posteriores a la actualización, como las llamadas a APIs basadas en cambios de props o estado.

Desmontaje
Este método se llama cuando el componente se elimina del DOM.

componentWillUnmount(): Se utiliza para limpiar recursos, como timers o suscripciones a eventos.



Ejemplo:

```
class MiComponente extends React.Component {
  constructor(props) {
    super(props);
    this.state = { contador: 0 };
    // Enlace del método
    // this.incrementarContador = this.incrementarContador.bind(this);
  }
  componentDidMount() {
    console.log('Componente montado');
  }
  shouldComponentUpdate(nextProps, nextState) {
    return nextState.contador !== this.state.contador;
  }
  componentDidUpdate(prevProps, prevState) {
```

```
    console.log('Componente actualizado');
  }
  componentWillUnmount() {
    console.log('Componente desmontado');
  }
incrementar = () => {
   this.setState({ contador: this.state.contador + 1 });
};
// incrementarContador(){
//    this.setState(prevState => ({ contador: prevState.contador + 1, }));
//  }
  render() {
    return (
      <div>
        <p>Contador: {this.state.contador}</p>
        <button onClick={this. incrementar }>Incrementar</button>
      </div>
    );
  }
}
```

Componente funcional

En los componentes funcionales de React, el ciclo de vida se gestiona principalmente a través de hooks, especialmente el hook useEffect. A diferencia de los componentes de clase, donde el ciclo de vida se maneja con métodos como componentDidMount, componentDidUpdate y componentWillUnmount, en los componentes funcionales estos aspectos se pueden manejar dentro de una única función de efecto.

Ciclo de Vida en Componentes Funcionales

Montaje: Similar a componentDidMount en componentes de clase. Aquí es donde puedes realizar efectos secundarios cuando el componente se monta por primera vez.

Actualización: Similar a componentDidUpdate. Se ejecuta cada vez que el componente se actualiza, ya sea por cambios en las props o en el estado.

Desmontaje: Similar a componentWillUnmount. Aquí puedes limpiar cualquier efecto que hayas creado, como cancelar suscripciones o limpiar temporizadores.

Ejemplo:

```
import React, { useState, useEffect } from 'react';

const MiComponente = () => {
  const [contador, setContador] = useState(0);

  // Efecto que se ejecuta en el montaje y actualización
useEffect(() => {
    console.log('El componente se ha montado o actualizado');
 }, [contador]); // Se ejecuta cuando 'contador' cambia
useEffect(() => {
    console.log('El componente se ha montado');
    // Función de limpieza que se ejecuta en el desmontaje
    return () => {
      console.log('El componente se va a desmontar');
    };
  }, []);
   const incrementarContador = () => {
    setContador(prevContador => prevContador + 1);
  };
```

```
  return (
    <div>
      <p>Contador: {contador}</p>
      <button onClick={incrementarContador}>Incrementar</button>
    </div>
  );
};

export default MiComponente;
```

Props

Las props (abreviatura de "properties") en un componente de React son un mecanismo que permite pasar datos y configuraciones de un componente padre a un componente hijo. Las props son inmutables, lo que significa que un componente no puede cambiar las props que recibe; solo puede utilizarlas para renderizar su contenido.

Características de las props:

Datos Dinámicos: Permiten enviar información dinámica a los componentes. Esto es esencial para construir aplicaciones interactivas y reutilizables.

Comunicarse entre Componentes: Facilitan la comunicación entre componentes. Un componente padre puede pasar datos a sus hijos, lo que ayuda a mantener la jerarquía y la estructura de la aplicación.

Personalización: Permiten personalizar el comportamiento y el estilo de un componente, haciendo que sean más versátiles.

Hooks

Un hook en React es una función que permite a los desarrolladores utilizar el estado y otros aspectos del ciclo de vida de los componentes funcionales. Los hooks fueron introducidos en React 16.8 para facilitar el uso del estado y la lógica de ciclo de vida sin necesidad de crear componentes de clase.

Tipos de Hooks

Hooks Básicos:

useState: Permite agregar estado local a un componente funcional.

useEffect: Permite realizar efectos secundarios, como llamadas a APIs o suscripciones, y manejar el ciclo de vida del componente.

useContext: Permite acceder al contexto de React sin necesidad de utilizar un componente de clase.

Hooks Personalizados: Puedes crear tus propios hooks que encapsulen lógica que se puede reutilizar en varios componentes. Esto ayuda a mantener el código limpio y organizado.

Otros Hooks

useReducer: Es una alternativa a useState que permite manejar estados más complejos. Utiliza un reductor, similar a Redux, para gestionar el estado y las acciones.

useRef: Permite crear una referencia mutable que persiste a través de renders. Es útil para acceder directamente a elementos del DOM o para almacenar valores que no causan re-render.

useMemo: Permite memorizar valores calculados y evitar cálculos innecesarios en cada render. Solo recalcula el valor si las dependencias cambian.

useCallback: Memoriza funciones, lo que ayuda a prevenir que se creen nuevas instancias de funciones en cada render. Ideal para optimizar componentes que dependen de funciones como props.

useLayoutEffect: Similar a useEffect, pero se ejecuta sincrónicamente después de todas las mutaciones del DOM. Útil para leer el DOM y hacer cambios antes de que el navegador pinte.

useImperativeHandle: Permite personalizar la instancia que se expone a los componentes padres cuando se utiliza ref. Generalmente se usa con forwardRef.

useDebugValue: Permite mostrar una etiqueta de depuración personalizada para los hooks personalizados en las herramientas de React.

Ejemplo useState y useEffect:

```
import React, { useState, useEffect } from 'react';

const Contador = () => {
  const [contador, setContador] = useState(0);

  // Este efecto se ejecuta al montar el componente y cada vez que 'contador' cambia
  useEffect(() => {
    console.log(`El contador ha cambiado a: ${contador}`);

    // Opcional: limpiar efecto al desmontar
    return () => {
      console.log('Limpieza del efecto');
    };
  }, [contador]); // Dependencia: se ejecuta cuando 'contador' cambia

  return (
    <div>
      <p>Contador: {contador}</p>
      <button onClick={() => setContador(contador + 1)}>Incrementar</button>
    </div>
  );
};

export default Contador;
```

Ventajas de los Hooks

Simplicidad: Permiten escribir componentes más simples y legibles.

Reutilización de lógica: Puedes crear hooks personalizados para compartir lógica entre componentes.

Eliminación de clases: Facilitan el uso de estado y efectos sin necesidad de crear clases, lo que puede simplificar el código.

Consideraciones

Reglas de los Hooks: Los hooks deben ser llamados en el nivel superior de un componente y no dentro de bucles, condiciones o funciones anidadas. Esto garantiza que React mantenga el estado de los hooks correctamente.

Uso en componentes funcionales: Los hooks solo pueden ser utilizados en componentes funcionales o en otros hooks personalizados.

En resumen, los hooks son una herramienta poderosa en React que permiten gestionar el estado y los efectos de una manera más declarativa y concisa, facilitando la creación de aplicaciones más limpias y eficientes.

PropTypes

Las PropTypes en React son una forma de validar las propiedades (props) que se pasan a un componente. Esto ayuda a asegurar que el componente reciba el tipo correcto de datos y que se sigan las expectativas sobre lo que se puede pasar como props. Esto es especialmente útil para detectar errores en desarrollo.

Características principales de PropTypes:

Validación de Tipos: Puedes especificar qué tipo de datos espera tu componente. Por ejemplo, si una prop debe ser un número, un string, un array, etc.

Requerido o No: Puedes marcar una prop como requerida. Si no se pasa la prop requerida, se mostrará una advertencia en la consola.

Documentación: Sirve como una forma de documentar las props que espera el componente, lo que facilita la comprensión para otros desarrolladores (o para ti mismo en el futuro).

Pequeña Practica

Instalar Material-UI

https://mui.com/material-ui/getting-started/installation/

Links

https://tailwindcss.com/docs

Codigo Ejmplo

https://github.com/jlcm201/example