

Proyecto React

2º DAW IES Al-Mudeyne

Índice

Módulo 1: Introducción de React

Módulo 2: Componentes y Props

Módulo 3: Manejo de eventos

Módulo 4: Conceptos Avanzados de Componentes

Módulo 5: Navegación en React

Módulo 6: Manejo de Estado Global

Módulo 7: Consumo de APIs

Módulo 8: Sintaxis JSX

Módulo 1: Introducción a React

1.1. ¿Qué es React?

1.1.1. Principios fundamentales

1.1.2. Virtual DOM

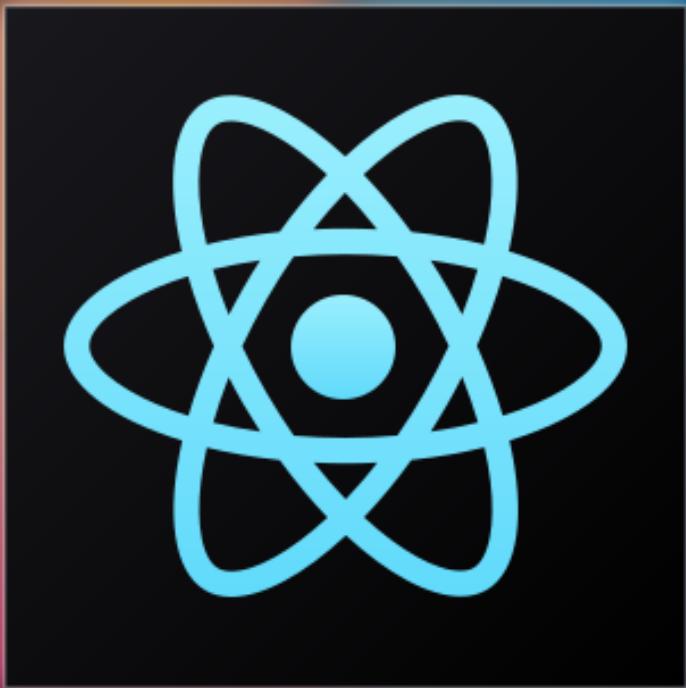
1.1. ¿Qué es React?

Biblioteca de Js creada por Facebook para construir interfaces de usuarios en interfaces web.

Se basa en componentes reutilizables, facilitando el desarrollo al evitar la repetición de código.

Funciona como una aplicación de una sola página. Carga el contenido desde los componentes para una renderización más rápida.

La sintaxis principal es JSX, una extensión de JS que integra la lógica con la interfaz. Su uso no es obligatorio pero mejora la eficiencia y legibilidad del código.



1.1.1 Principios Fundamentales

·Componentes Reutilizables

Cada componente representa una parte específica de la interfaz y puede ser reutilizado en diferentes partes de la aplicación.

·Unidireccionalidad de Datos

Los datos fluyen en una sola dirección. Los datos fluyen de los componentes padres a los componentes hijos. Universal: React se puede ejecutar tanto en el cliente como en el servidor.

·JSX (JavaScript XML)

Es una forma de escribir código que combina JavaScript y HTML de manera más legible. Hace que la creación de interfaces de usuario sea más sencilla al permitir escribir código similar al HTML

1.1.2 Virtual DOM

Al cambiar los datos en React, se genera un nuevo DOM virtual (VDOM), una representación ideal en memoria de la interfaz de usuario (IU) que se sincroniza con el DOM real. Esto, a través de bibliotecas como ReactDOM, minimiza manipulaciones directas al DOM, logrando una renderización más eficiente.

Comparación:

La nueva representación virtual se compara con la anterior para detectar cambios.

Actualización eficiente:

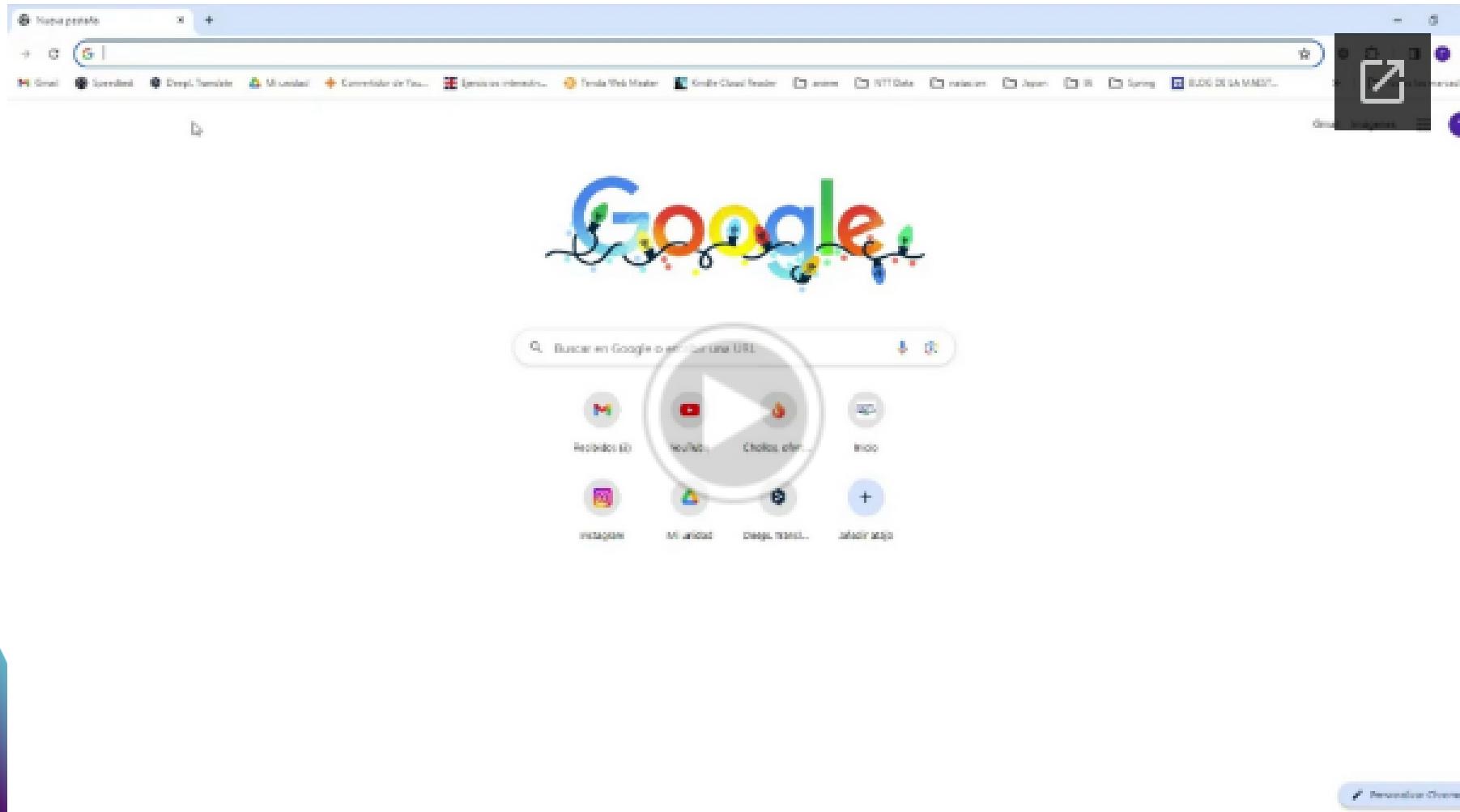
Solo se aplican al DOM real los cambios identificados, evitando actualizaciones innecesarias y mejorando el rendimiento.

1.2. Configuración del entorno de desarrollo

1.2.1. Instalación de Node.js y npm

1.2.2. Creación de una aplicación React

Video Tutorial



Módulo 2: Componentes y Props

2.1. Componentes en React

2.1.1. Creación de componentes funcionales y de clase

2.1.2. Propiedades (Props) en React

2.1. Componentes en React

a) Definición:

- En React, los componentes son bloques de construcción fundamentales que permiten crear interfaces de usuario reutilizables y modulares. Hay dos tipos principales de componentes: funcionales y de clase.
- La idea de hacer todos estos archivos es dividir el grueso del código en diferentes módulos separados, y por si es necesario en algún otro momento del proyecto que sea más fácil reutilizando

2.1.1. Creación de componentes funcionales y de clase

- Componentes Funcionales: Son funciones de JavaScript que devuelven un elemento de React. Antes de la introducción de React Hooks, estos componentes eran principalmente usados para componentes simples que no requerían estado o métodos del ciclo de vida

```
import React from 'react';

const boton = ({ texto, onClick }) => {
  return (
    <button onClick={onClick}>
      {texto}
    </button>
  );
};
```

2.1.1. Creación de componentes funcionales y de clase

- En React, los componentes son bloques de construcción fundamentales que permiten crear interfaces de usuario reutilizables y modulares. Hay dos tipos principales de componentes: funcionales y de clase

```
import React, { Component } from 'react';

class Saludo extends Component {
  render() {
    return <h1>Hola, {this.props.nombre}</h1>;
  }
}

export default Saludo;
```

2.1.2. Propiedades (Props) en React

- Las propiedades (Props) son mecanismos que permiten pasar datos de un componente padre a un componente hijo en React. Son inmutables y ayudan a mantener la unicidad y consistencia de los componentes. Las Props se pasan como argumentos a los componentes y se utilizan para personalizar y configurar el comportamiento y la apariencia de un componente.

```
function BotonEventos(props){  
    return <button onClick={props.onClick}>{props.text}</button>  
}  
export default BotonEventos
```

2.2. Estado y ciclo de vida

2.2.1. Uso del estado

2.2.2 Ciclo de vida de los componentes

2.2.1. Uso del estado

a) Distinción:

- **Componentes sin estado:** no manejan su propio estado interno, presentan información que llega como parámetro
- **Componentes con estado:** manejan su propio estado interno, permiten manipular información al interactuar con la aplicación

b) Ejemplos de componentes sin estado:

- Lista
- Encabezado de página

c) Ejemplos de componentes con estado:

- Formulario
- Contador

Formulario

Lista

```
import React from 'react';

function ItemList(props) {
  return (
    <li>
      {props.product.name} - {props.product.price}
    </li>
  );
}

export default ItemList;
```

```
import { useState } from 'react';
function SimpleForm() {
  const [name, setName] = useState('');
  return (
    <div>
      <form>
        <input
          type="text"
          placeholder="Nombre"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
        <button type="submit">Enviar</button>
      </form>
    </div>
  );
}
export default SimpleForm;
```

2.2.2. Ciclo de vida de los componentes

a) Definición:

- Fases por las que puede pasar un componente desde su creación hasta su destrucción

b) Fases:

- a) **Mounting (montado)**: primera vez que el componente va a generarse, incluyendose en la UI
- b) **Updating (actualizado)**: el componente ya generado se actualiza
- c) **Unmounting (desmontado)**: el componente se elimina de la UI

c) Hooks para manejar el ciclo de vida:

1) Definición:

- Funciones que permiten agregar características a los componentes funcionales (gestionar estado, acceso a datos...)

2) Hooks:

Mounting Updating

- **useState**: permite inicializar y manipular estado de un componente

Mounting Updating Unmounting

- **useEffect**: se ejecuta después de que el componente se renderice (enviar solicitudesHTTP, actualizar cachés)

Contador

```
import { useState, useEffect } from "react";
function Counter() {
    const [counter, setCounter] = useState(0);
    const handleClick = () => {
        setCounter(counter + 1);
    }

    useEffect(() => {
        console.log('Componente montado o actualizado: ', counter);
        return () => {
            console.log("Componente destruido");
        }
    }, [counter]);

    return (
        <div>
            <h1>{counter}</h1>
            <button onClick={handleClick}>Sumar uno</button>
        </div>
    )
}
export default Counter;
```

Módulo 3: Manejo de eventos

3.1. Eventos en React

3.1.1. Eventos de usuario

3.1.2. Enlace de eventos y métodos

3.2. Formularios en React

3.1. Eventos en React

¿Qué son?

Acciones desencadenadas por el usuario, como hacer clic en un botón o escribir en un campo de entrada. Estos eventos permiten que las aplicaciones respondan dinámicamente a las acciones del usuario

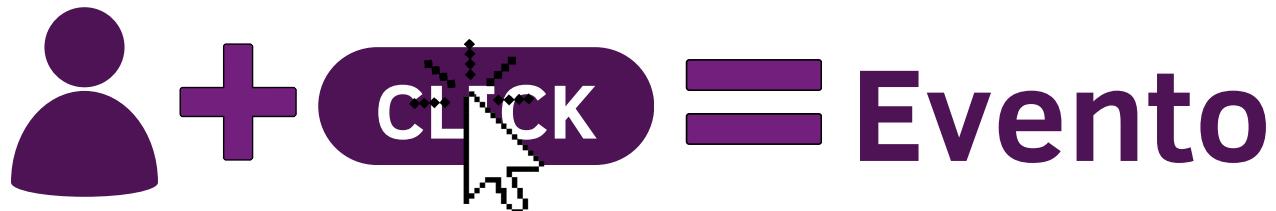
¿Qué ofrecen?

Mejorar la experiencia de usuario y la interactividad de la aplicación

¿Cómo se integran

los eventos se manejan utilizando sintaxis JSX similar a HTML, lo que facilita su integración en la lógica de los componentes.

3.1.1. Eventos de Usuario



React utiliza su propio sistema de manejo de eventos.

1. Se define el tipo de evento (onClick, onChange, ...)
2. Se crea la función que se ejecutará cuando ocurra el evento (Manejador de Eventos)
3. Se vincula el manejador al elemento de la interfaz de usuario.
4. Cuando el usuario realiza la acción, React ejecuta automáticamente el manejador especificado
5. El manejador de evento recibe un objeto de evento como argumento. Éste contiene información sobre el evento, la posición del ratón, el valor de un campo de entrada , ...

3.1.2. Enlace de eventos y métodos

Los enlaces de eventos se utilizan para manejar interacciones del usuario.

Se declaran usando una sintaxis similar a HTML, pero con **camelCase**.

Pueden definirse en el componente y luego referenciarlos en los enlaces de eventos. También es posible pasarles argumentos.

Los enlaces de eventos y métodos son herramientas clave en React para gestionar la interactividad y la lógica en los componentes de la interfaz de usuario.

3.2. Formularios en React

¿Qué son?

Los formularios en React son una parte fundamental de la construcción de aplicaciones web interactivas. Permite a los usuarios ingresar datos y realizar acciones.

React proporciona algunas herramientas para manejar su comportamiento de manera eficiente y efectiva.

Existen dos tipos de formularios, los controlados y no controlados.

3.2.1. Formularios controlados

- Los valores de los campos de entrada (inputs) son controlados por el estado de React.
- Cada vez que el usuario escribe algo en un campo de entrada, ese cambio se refleja inmediatamente en el estado de React.
- Para actualizar los valores del formulario, se debe cambiar explícitamente el estado utilizando, por ejemplo, '**useState()**'.
- Es útil cuando se necesitan hacer validaciones o realizar acciones específicas en respuesta a los cambios en el formulario.

3.2.1. Ejemplo

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

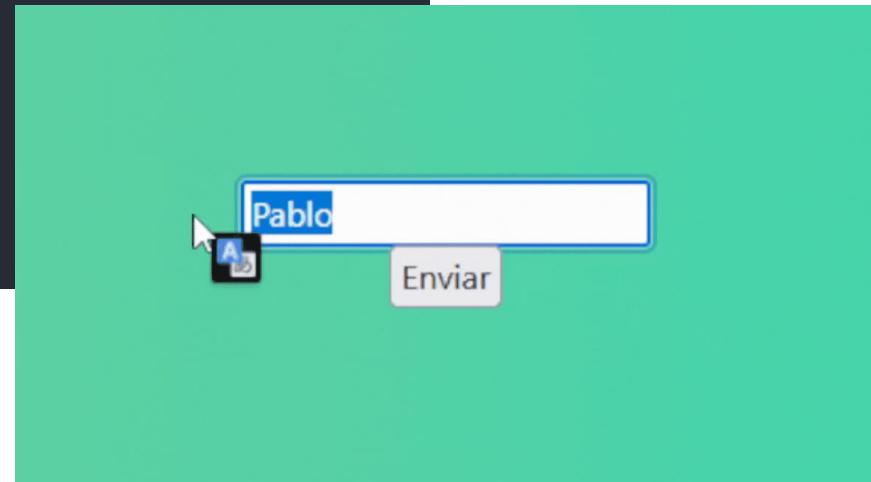
function FormControlado() {
  //se inicializa el estado 'nombre' con el valor vacío
  const [nombre, setNombre] = useState('');

  //función que se ejecuta cuando el formulario se envía
  const handleSubmit = (event) => {
    event.preventDefault(); //evita que el formulario se envíe por defecto
    alert('El nombre introducido es: ' + nombre); //muestra un mensaje con el nombre introducido
  }

  //función que se ejecuta cuando el input cambia
  const handleChange = (event) => {
    setNombre(event.target.value); //actualiza el estado 'nombre' con el valor del input
  }

  return (
    <form onSubmit={handleSubmit}>
      {/* campo de entrada controlado por el estado 'nombre' */}
      <input type="text" value={nombre} onChange={handleChange} />
      <input type="submit" value="Enviar" />
    </form>
  );
}

export default FormControlado;
```



3.2.2. Formularios no controlados

a) Definición:

- Los valores de los inputs son controlados directamente por el DOM

b) Librería:

- React Hook Form

c) Ventajas:

- Mejor rendimiento
- Menos renderizado

d) Instalación e importación:

- npm install react-hook-form
- import {useForm} from 'react-hook-form'

d) Funciones del hook useForm:

- **register**: vincular un campo de entrada con el estado interno de la librería
- **handleSubmit**: manejar el envío del formulario

e) Validación:

- **formState: {errors}**: errores de validación asociados a los campos de entrada

FORMULARIO NO CONTROLADO

```
import { useForm } from 'react-hook-form';
function SimpleForm() {
  const { register, handleSubmit, formState: { errors } } = useForm();
  const onSubmit = (data) => {
    console.log('Nombre ingresado:', data.nombre);
  };
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input
        type="text"
        placeholder="Ingresa tu nombre"
        {...register('nombre')}
      />
      {errors.nombre?.type === 'required' && <p>Campo requerido</p>}
      <button type="submit">Enviar</button>
    </form>
  );
}
export default SimpleForm;
```

Módulo 4: Conceptos Avanzados de Componentes

4.1. Composición de componentes

4.1.1. Reutilización de componentes

4.1.2. Patrones de composición

4.1. Composición de componentes

En el contexto de **desarrollo de software**, se refiere a la forma en que se pueden combinar y organizar diferentes partes de un sistema para construir aplicaciones más grandes y complejas. Esta práctica facilita la reutilización de código, mejora la mantenibilidad y promueve un diseño modular.

En **React**, es un principio esencial. Ya que la reutilización de componentes y la aplicación de patrones de composición son prácticas fundamentales.

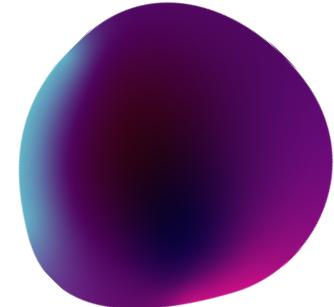
4.1.1. Reutilización de componentes

Es un principio fundamental en el desarrollo de software.

Sus beneficios, **eficiencia, consistencia y mantenibilidad**.

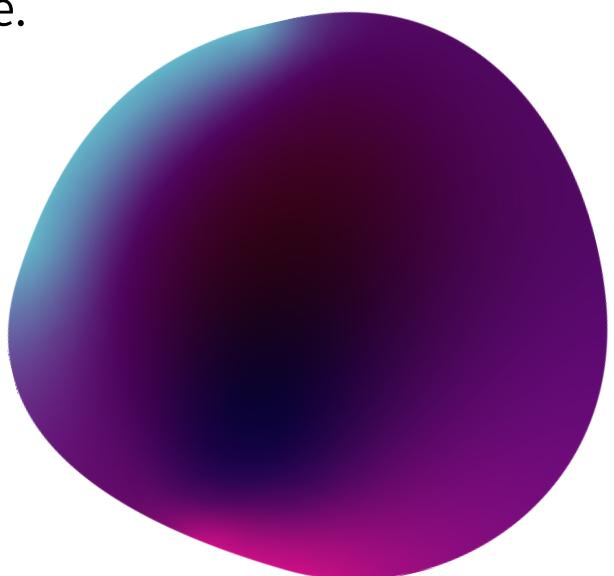
La reutilización de componentes se puede lograr a través de **bibliotecas, frameworks y patrones de diseño** que facilitan la integración de componentes existentes en nuevos proyectos.

4.1.2. Patrones de composición



Son enfoques probados y prácticos para combinar componentes de manera efectiva.

En **React**, algunos patrones de composición son fundamentales para estructurar y organizar aplicaciones de manera eficiente.



Composición de Decorador

Se puede lograr mediante el uso de **HOCs** (Higher Order Components).

Estos son componentes que toman un componente y devuelven otro componente con funcionalidades adicionales.

Esto es útil cuando se desea extender o modificar el comportamiento de un componente sin cambiar su código fuente.

Otros patrones de composición:

- **Composición de Componentes:**

Construir componentes a partir de otros componentes, combinando partes más pequeñas para formar un todo.

- **Render Props:**

Pasar una función como prop a un componente para permitirle renderizar algo dentro del componente padre.

- **Hooks y Composición:**

Utilizar React Hooks para gestionar el estado y el ciclo de vida en componentes funcionales.

Módulo 5: Navegación en React

5.1. Uso de React Router

5.1.1. Configuración básica

5.1.2. Rutas anidadas y parámetros

5.1. Uso de React Router

Partiendo desde el principio, React Router es la librería más popular para gestionar la navegación en aplicaciones internas hechas con dicha herramienta.

Su principal funcionalidad es proporcionar una forma declarativa y basada en componentes para manejar la navegación en aplicaciones React, facilitando la creación de interfaces de usuario dinámicas y amigables con el usuario. Y logrando a su vez reutilizar código de unas páginas en otras.

5.1.1. Configuración básica

Al ser React Router una librería solo nos tendremos que invocar un terminal en la raíz de nuestro proyecto y escribir el comando:

```
npm install react-router-dom
```

5.1.2. Rutas anidadas y parámetros

Definición de Rutas Anidadas:

- Agrupación de rutas React bajo la misma URL.
- Simplificación del código al organizar rutas relacionadas.
- Posibilidad de desplegar rutas desde la misma página de la aplicación.

Anidamiento de rutas

- Utilización de componentes <Route> para crear rutas simples y anidadas.
- Especificación de rutas "hijas" dentro del componente "padre".
- Componente <Outlet>: la Ruta padre utiliza el componente Outlet para renderizar la Ruta hijo coincidente.

```
function Public() {  
  return (  
    <div>  
      <Router>  
        <Menu />  
        <Routes>  
          <Route path="/rutas/" element={<RutasAnidadas />} >  
            <Route path="frontend" element={<p>FrontEnd ✨</p>}/>  
            <Route path="backend" element={<p>BackEnd 🤖</p>}/>  
          </Route>  
        </Routes>  
      </Router>  
    </div>  
  );  
}
```

```
function RutasAnidadas() {  
  return (  
    <div>  
      <ul style={{display: "flex"}}>  
        <li><Link to={"frontend"}>Frontend</Link></li>  
        <li><Link to={"backend"}>Backend</Link></li>  
      </ul>  
      <Outlet/>  
    </div>  
  );  
}
```

localhost:3000/rutas/backend

Ejemplo:

Parámetros en las rutas

¿Qué son los Parámetros en las Rutas?

- Herramienta para capturar valores dinámicos en las URL.
- Facilita la construcción de componentes reutilizables.
- Manipulación de datos basada en la ruta.

Ejemplo Práctico:

- Uso del componente principal App con <Router>.
- Creación de enlaces con diferentes valores para "username".
- Configuración de ruta con parámetro ":username" y componente UserProfile.

Funcionamiento:

- Enlaces dinámicos con valores de "username".
- React Router captura dinámicamente el valor en la ruta.
- Acceso al valor a través del objeto match.params en el componente UserProfile.

// Componente principal de la aplicación

```
const App = () => (
  <Router>
    <div>
      <ul>
        {/* Creación de enlaces con diferentes valores para "username" */}
        <li><Link to="/user/john">Usuario John</Link></li>
        <li><Link to="/user/jane">Usuario Jane</Link></li>
      </ul>

      {/* Ruta con parámetro ":username" y componente asociado UserProfile */}
      <Route path="/user/:username" component={UserProfile} />
    </div>
  </Router>
);
```

// Componente funcional UserProfile que recibe match como prop

```
const UserProfile = ({ match }) => (
  <div>
    <h2>Perfil de Usuario</h2>
    {/* Accediendo al parámetro de la ruta llamado "username" */}
    <p>Usuario: {match.params.username}</p>
  </div>
);
```

Ejemplo:

Módulo 6: Manejo de Estado Global

6.1. Introducción a la gestión de estado global

6.1.1. Context API

6.1.2. Redux

6.1. Introducción a la gestión de estado global

Tipos de estado:

- Local: específico del componente (useState)
- Global: accesible por todos los componentes (Context API y Redux)

El manejo del estado global permite:

- Compartir datos entre componentes sin necesidad de utilizar props.
- Solucionar el Prop Drilling: pasar datos a través de múltiples niveles.

6.1.1. Contexto en React

“Imagina una aplicación con múltiples componentes anidados, y necesitas compartir ciertos datos entre ellos sin pasar las propiedades a través de cada nivel. Aquí es donde el contexto se vuelve útil.”

Es una característica que permite pasar datos a través del árbol de componentes sin tener que pasar explícitamente las propiedades a cada nivel.

6.1.1. Uso del contexto

Context está diseñado para compartir datos que pueden considerarse “globales” para un árbol de componentes en React, como el usuario autenticado actual, el tema o el idioma preferido.

El contexto se define utilizando **React.createContext()** y se comparte utilizando un componente **Provider** que envuelve a los componentes que necesitan acceder a ese contexto. Para consumir el contexto, se utiliza el componente **Consumer** o el hook **useContext** en componentes funcionales

6.1.1. Proveedores y consumidores de contexto

- **Proveedores de Contexto:** Un proveedor de contexto es un componente en React que proporciona el contexto a los componentes secundarios. Este componente utiliza la prop value para pasar los datos del contexto.
- Los componentes que desean acceder al valor proporcionado por el proveedor pueden hacerlo utilizando el **Consumer** o el hook **useContext**.

Ejemplos:

- **Proveedores de Contexto:**

```
const UserContext = React.createContext();
function App() {
  const nombreUsuario = "John";

  return (
    <UserContext.Provider value={nombreUsuario}>
      {/* Otros componentes que pueden acceder al nombreUsuario */}
    </UserContext.Provider>
  );
}
```

- **Usando Consumer:**

```
class ComponenteConsumidor extends React.Component {
  render() {
    return (
      <UserContext.Consumer>
        {nombreUsuario => <p>Hola, {nombreUsuario}</p>}
      </UserContext.Consumer>
    );
  }
}
```

6.1.2 Redux

Qué es:

- Biblioteca JavaScript

Principios:

- Única fuente de verdad: **store** contiene todo el estado de la aplicación.
- Estado de solo lectura: solo se puede modificar emitiendo una **acción**.
- Cambio mediante funciones puras: se usan **reducers**, que reciben el estado actual y la acción a realizar y devuelven un nuevo estado.

Instalación de Redux Toolkit y React Redux

- `npm i @reduxjs/Toolkit react-redux`

6.1.2 Redux

¿Cómo usarlo?

1. Definir un slice (segmento de estado global) usando **createSlice** con nombre, estado inicial y reductores (acciones) para modificarlo.
2. Crear un store usando **configureStore** de React Toolkit. Se configura con un objeto reducer donde especificamos el slice al que nos referimos y el reductor que contiene las acciones de ese estado.
3. Usar el componente **Provider** para proporcionar el store a todos los componentes.
4. Crear componentes que interactúen con el Store usando **useSelector** de react-redux, que tiene acceso a los estados globales.

slice.js

```
import { createSlice } from '@reduxjs/toolkit';
const initialState = {
  name: '',
};
const userSlice = createSlice({
  name: 'user',
  initialState,
  reducers: {
    setName: (state, action) => {
      state.name = action.payload;
    },
    clearName: (state) => {
      state.name = '';
    },
  },
});
export const { setName, clearName } = userSlice.actions;
export default userSlice.reducer;
```

store.js

```
import { configureStore } from '@reduxjs/toolkit';
import userReducer from './userSlice';
const store = configureStore({
  reducer: {
    user: userReducer,
  },
});

export default store;
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import createStore from './store';

import App from './';

const store = createStore();

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

user.js

```
import React from 'react';
import { useSelector } from 'react-redux';
function UserDisplay() {
  const userName = useSelector(state => state.user.name);
  return (
    <div>
      <p>Nombre de usuario: {userName}</p>
    </div>
  );
}

export default UserDisplay;
```

Módulo 7: Consumo de APIs

7.1. Realización de solicitudes HTTP

7.1.1. Uso de Fetch y Axios

7.1.2. Manejo de datos asincrónicos

7.1. Realización de solicitudes HTTP

Que es una API :

- Una API (Interfaz de Programación de Aplicaciones) permite la comunicación entre diferentes software.
- En el contexto web, se utiliza para solicitar y enviar datos entre el frontend y el backend.

El manejo del estado global permite:

- Compartir datos entre componentes sin necesidad de utilizar props.
- Solucionar el Prop Drilling: pasar datos a través de múltiples niveles.

7.1.1. Uso de Fetch y Axios

FETCH:

- Función nativa de JavaScript para hacer solicitudes HTTP.
- Sintaxis simple y moderna basada en promesas.
- Soporta varios tipos de datos: JSON, texto, blobs, formularios, etc.
- Disponible en navegadores modernos y en Node.js.
- Ejemplo:
-

```
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

7.1.1. Uso de Fetch y Axios

AXIOS:

- Librería externa para hacer solicitudes HTTP.
- Sintaxis simplificada en comparación con fetch.
- Interceptores y configuración global para manejar solicitudes y respuestas.
- Compatible con navegadores y Node.js.
- Ejemplo:
-

```
axios.get(url)
  .then(response => console.log(response.data))
  .catch(error => console.error('Error:', error));
```

7.2. Manejo de datos asincrónicos

Que es una API :

- Hooks de React para manejar el estado y los efectos secundarios en componentes funcionales.
- **useState**: Para definir y actualizar el estado en componentes funcionales.

```
const [data, setData] = useState(null);
```

- **useEffect**: Para realizar operaciones secundarias, como solicitudes HTTP, cuando el componente se monta, se actualiza o se desmonta.

```
useEffect(() => {
  // Operaciones secundarias aquí
}, []); // [] para ejecutar solo una vez al montar
```

7.2. Integración de datos en componentes

7.2.1. Actualización de componentes con datos externos

7.2.2. Manipulación de respuestas JSON

7.2. Integración de datos en componentes

La integración de datos en componentes se refiere a como los datos obtenidos de una API se incorporan en los componentes de React para su visualización y manipulación

7.2.1. Actualización de componentes con datos externos

La actualización de componentes con datos externos en React implica solicitar datos de una API, almacenarlos en el estado del componente y volver a renderizarlo para reflejar los nuevos datos.

Si se obtienen datos diferentes en futuras solicitudes, se actualiza el estado y se vuelve a renderizar el componente para mantener la interfaz de usuario actualizada.

Sin embargo, React solo renderiza el componente si los nuevos datos son diferentes a los antiguos, evitando renderizaciones innecesarias.

7.2.2. Manipulación de respuestas JSON

La manipulación de respuestas JSON en desarrollo web implica procesar los datos devueltos por una API en formato JSON para su uso en una aplicación.

Esto incluye el parseo de la cadena JSON a un objeto javascript mediante **JSON.parse()** o **json()** al usar **Fetch**. Posteriormente, se accede a los datos mediante notacion de puntos o corchetes. En casos donde se necesita cambiar el formato de los datos, se emplean métodos como **.map()**, **.filter()** o **.reduce()**.

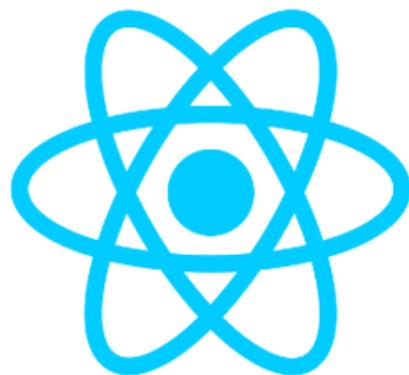
Finalmente, los datos parseados y posiblemente transformados se almacenan en el estado del componente para su uso en el proceso de renderizado.

Módulo 8: Sintaxis JSX

1. INTRODUCCIÓN
2. SINTAXIS BÁSICA
3. ELEMENTOS Y COMPONENTES
4. RENDERIZACIÓN
5. MANEJO DE EVENTOS
6. ESTILOS
7. REGLAS IMPORTANTES

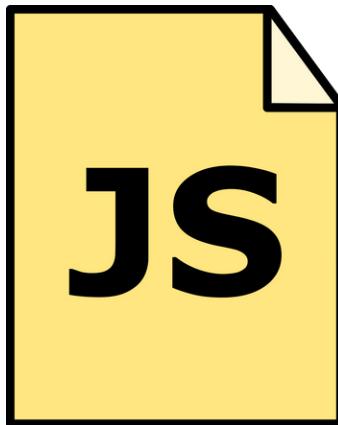
1. Introducción

JSX es una extensión sintáctica de JavaScript que permite a los desarrolladores escribir marcas similares a HTML dentro de un archivo JavaScript. Esto facilita la creación de interfaces de usuario dinámicas e interactivas para aplicaciones web.



2. Sintaxis básica

```
const element = <h1>Hello, mundo!</h1>;
```



2. Sintaxis básica

```
IMPORT REACT FROM 'REACT';

CONST MYCOMPONENT=()=>{

  CONST NOMBRE='JOHN';
  CONST EDAD=30;

  RETURN(
    <DIV>

      <H1>HOLA,{NOMBRE}!</H1>
      <P>TIENES{EDAD}AÑOS</P>
      <P>EL AÑO QUE VIENE TENDRÁS {EDAD+1}</P>
      {EDAD}>=18 && <P>ERES ADULTO.</P>

    </DIV>
  );
};

EXPORT DEFAULT MYCOMPONENT;
```

3. Elementos y componentes

Diferencia entre:

Los elementos son instancias de un componente o elementos JSX simples.

```
const simpleElement = <p>Este es un elemento JSX simple</p>;
```

Los componentes son funciones o clases de React que pueden aceptar propiedades (props) y devolver elementos JSX. Estos son componentes reutilizables ya que los puedes usar en diferentes lugares de tu aplicación.

```
const Saludo = (props) => {  
  return <p>Hola, {props.nombre}!</p>;  
};
```

4. Renderización

Cuando hablamos de renderización en JSX, nos referimos al proceso de convertir componentes de React escritos en JSX en elementos de la interfaz de usuario que se muestran en el navegador.

```
function MiComponente() {  
  return <div>Hola, soy un componente.</div>;  
}
```

```
const miElemento = <MiComponente />;
```

```
import ReactDOM from 'react-dom';
```

```
ReactDOM.render(miElemento, document.getElementById('root'));
```

4. Renderización

The screenshot shows a code editor with two tabs open: `App.js` and `index.js`.

App.js:

```
ejercicios-react > src > JS App.js > App
You, hace 21 horas | 1 author (You)
1 import logo from "./logo.svg";
2 import MiComponente from "./Components/Ejerc
3 import MiComponente2 from "./Components/Ejer
4 import MockupCompleto from "./Components/Eje
5 import ComponentePadre from "./Components/Ej
6 import Contenedor from "./Components/Ejercic
7 import "./App.css";

8
9 function App() {
10   return (
11     <div> You, hace
12     | <Contenedor />
13   </div>
14 );
15 }
16
17 export default App;
18
```

index.js:

```
ejercicios-react > src > JS index.js > ...
You, anteayer | 1 author (You)
1 import React from 'react';
2 import ReactDOM from 'react-dom/client';
3 import './index.css';
4 import App from './App';
5 import reportWebVitals from './reportWebVitals';

6
7 const root = ReactDOM.createRoot(document.getElementById('root'));
8 root.render(
9   <React.StrictMode>
10   | <App />
11   </React.StrictMode>
12 );
13
14 reportWebVitals();
15
```

4. Renderización



```
index.html M X JS index.js M JS App.js M
ejercicios-react > public > index.html > ...
You, anteayer | 1 author (You)
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8" />
5      <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
6      <meta name="viewport" content="width=device-width, initial-scale=1" />
7      <meta name="theme-color" content="#000000" />
8      <meta
9        name="description"
10       content="Web site created using create-react-app"
11     />
12     <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
13     <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
14     <title>React App</title>
15   </head>
16   <body>
17     <noscript>You need to enable JavaScript to run this app.</noscript>
18     <div id="root"></div>
19   </body>
20 </html>
```

5. Manejos de Eventos

Los eventos en JSX se manejan de manera similar a HTML, utilizando atributos de eventos como onClick y onChange.

```
<button onClick={manejarClick}>Haz clic</button>
<input onChange={manejarCambio} />
```

```
<button onclick="manejarClick()">Haz clic</button>
<input onchange="manejarCambio()" />
```

6. Estilos

CSS puede aplicarse a componentes JSX de varias formas;

- Estilos inline
- Archivos CSS independientes
- Bibliotecas CSS-in-JS.

Los estilos inline se definen directamente dentro del marcado JSX utilizando objetos JavaScript, mientras que los archivos CSS independientes o las bibliotecas CSS-en-JS permiten aplicar estilos externos y modulares a los componentes.

6. Estilos

```
IMPORT REACT FROM 'REACT';

CONST MYCOMPONENT = () => {
  CONST STYLES = {

    BACKGROUND-COLOR:'BLUE',
    COLOR:'WHITE',
    PADDING:'10PX'
  };

  RETURN(
    <DIV STYLE={STYLES}>
      <H1> HELLO, WORLD! </H1>
      <P> ESTILOS EN LINEA </P>

    </DIV>
  );
};

EXPORT DEFAULT MYCOMPONENT;
```

7. Reglas importantes JSX

- Devuelve siempre un único elemento root
- Utilizar className en lugar de class
- Utiliza llaves para las expresiones de JavaScript
- Utiliza camelCase para la mayoría de las cosas en JSX
- Cierra siempre las etiquetas
- Utiliza etiquetas de autocierre para elementos vacíos

Gracias

Tibu Mayo González

Alicia López Vázquez

Pablo Márquez Gómez

Adrián Martínez Segura

Carmen Sánchez Martín

José Antonio Vergara Sánchez