

DOCUMENTACIÓN REACT JS

2 ° D A W I E S A L - M U D E Y N E

Alicia López Vázquez
Pablo Márquez Gómez
J. Antonio Vergara Sánchez
Carmen Sánchez Martín
Adrián Martínez Segura
Tibu Mayo González

Módulo 1: Introducción a ReactJS.....	4
1.1. ¿Qué es ReactJS?.....	4
1.1.1. Principios fundamentales.....	4
1.1.2 Virtual DOM.....	5
Módulo 2: Componentes y Props.....	6
2.1. Componentes en React.....	6
2.1.1. Creación de componentes funcionales y de clase.....	7
2.1.2. Propiedades (Props) en React.....	7
Módulo 3: Manejo de eventos.....	17
3.1. Eventos en React.....	17
3.1.1. Eventos de usuario.....	17
3.1.2. Enlace de eventos y métodos.....	17
3.2. Formularios en React.....	18
Módulo 4: Conceptos avanzados de componentes.....	30
4.1. Composición de componentes.....	30
4.1.1. Reutilización de componentes.....	30
4.2.Contexto en React.....	36
Módulo 5: Conceptos avanzados de componentes.....	39
5.1. Uso de React Router.....	39
5.1.1. Configuración en React.....	39

Módulo 1: Introducción a ReactJS

1.1. ¿Qué es ReactJS?

React.js, o simplemente React, es una biblioteca de JavaScript creada por Facebook para construir interfaces de usuario en aplicaciones web. Se basa en componentes reutilizables que forman partes de la interfaz, facilitando el desarrollo al evitar la repetición de código. React funciona como una aplicación de una sola página, cargando el contenido directamente desde los componentes para una renderización más rápida. La sintaxis principal es JSX, una extensión de JavaScript que integra la lógica del código con la interfaz de usuario. Aunque JSX no es obligatorio, su uso común mejora la eficiencia y legibilidad del código en aplicaciones React. En resumen, React ofrece modularidad, reutilización de componentes y renderización eficiente para interfaces de usuario web.

1.1.1. Principios fundamentales

Componentes Reutilizables:

- React descompone la interfaz de usuario en pequeños bloques llamados "componentes". Cada componente representa una parte específica de la interfaz y puede ser reutilizado en diferentes partes de la aplicación.

Unidireccionalidad de Datos:

- En React, los datos fluyen en una sola dirección. Esto significa que cualquier cambio en los datos de la aplicación se realiza de manera predecible y sigue un flujo unidireccional, facilitando la comprensión y el mantenimiento del código.

JSX (JavaScript XML):

- JSX 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 directamente dentro de los archivos JavaScript.

Estos principios proporcionan una base sólida para el desarrollo con React, simplificando la creación y mantenimiento de aplicaciones web interactivas y eficientes.

1.1.2 Virtual DOM

El Virtual DOM en React es una técnica que optimiza la actualización de la interfaz de usuario. En lugar de actualizar directamente el DOM (Document Object Model) cada vez que hay un cambio en los datos, React utiliza una representación virtual del DOM, llamada Virtual DOM.

Resumen de cómo funciona:

Cambio de Datos: Cuando los datos cambian, React crea una nueva representación virtual del DOM.

Comparación: Compara la nueva representación virtual con la anterior para identificar los cambios.

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

El Virtual DOM minimiza las manipulaciones directas del DOM, lo que resulta en una renderización más eficiente y una mejor experiencia del usuario.

Módulo 2: Componentes y Props

2.1. Componentes en React

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 logo from './logo.svg';
import './App.css';
import Welcome from './componets/saludo.jsx';
import BotonEventos from './componets/botonEventos.jsx';

function App() {
  return (
    <div>
      <Welcome name='Adrián' />
      <BotonEventos onClick={alerta} text="Mi boton" />
    </div>
  );
}

function alerta() {
  return alert("Has pulsado el botón");
}

export default App;
```

En App.js encontramos el documento principal el cual es el que se muestra en última instancia, para ello se necesitará importar los componentes que se utilizarán, y a modo de etiqueta se llamarán a dichos componentes.

Para poder trabajar con ellos es necesario exportarlo para que los archivos puedan ser importados el documento principal. Cabe destacar que los archivos .jsx son código html y javascript fusionados, 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 también tendremos los props que son una especie de variable que sirve para utilizar información de otras variables heredadas del padre en los hijos igualmente para ahorrar trabajo reutilizando código que está escrito y centralizar la modificación para posibles cambios venideros y un mantenimientos más sencillo del código a largo plazo.

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>
  );
};
```

-Componentes de Clase: Se definen mediante una clase de JavaScript que extiende `React.Component`. Permiten el uso de estado y métodos del ciclo de vida. Sin embargo, con la introducción de Hooks, el uso de componentes de clase ha disminuido en favor de los funcionales ya que es necesario definir su estructura

```
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
```

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

export default Welcome;
```

2.2 Estado y Ciclo de Vida

2.2.1 Uso del estado

En React existen componentes sin estado, que son aquellos que solo muestran las propiedades que recibe como parámetros, sin realizar ninguna actualización sobre ellos. Los componentes con estado se distinguen de los anteriores, debido a que estos tienen un estado asociado al componente, el cual se manipula a medida que el usuario interactúa con la aplicación. Un ejemplo típico de componentes con estado son los formularios.

El estado se inicia en el constructor y se puede modificar a través del método `setState()`. Este método es necesario porque el estado es inmutable, es decir, no se puede modificar directamente. Cuando se llama a `setState()`, React crea una copia del estado, aplica las modificaciones a esa copia y luego reemplaza el estado antiguo con el nuevo. Este proceso es asíncrono: puede no completarse inmediatamente.

Veamos un ejemplo de cómo se usa el estado en un componente basado en clase de React:

```
import React from 'react';

class Estado extends React.Component {
  // En el constructor creamos el objeto del estado donde damos valores a sus variables

  constructor(props) {
    super(props);

    // Inicializamos el estado con un contador en 0

    this.state = {
      contador: 0,
    }

    // Está prohibido modificar directamente el estado.

    // Usamos setInterval para incrementar el contador cada segundo.

    setInterval(() => {

      // Usamos setState para actualizar el estado.

      // Esto desencadena un nuevo renderizado del componente.
    }, 1000);
  }
}
```



```

    this.setState({
      contador: this.state.contador + 1
    });
  }, 1000);
}

// El método render se encarga de describir qué se va a dibujar en el DOM.

render() {
  return (
    <div>
      <h2>Estado de un Componente</h2>
      /* Mostramos el valor actual del contador */
      <p>{this.state.contador}</p>
    </div>
  );
}
}

export default Estado;

```

2.2.2 Ciclo de vida de los componentes

El concepto de "ciclo de vida" hace referencia a las tres fases por las que puede pasar un componente de clase, a saber, montado, actualización y desmontado del componente. Estas fases a su vez se dividen en distintos métodos que puede tener el componente.

1. **Montado**: la primera fase ocurre cuando un componente se crea y se monta en la interfaz de usuario. En esta primera fase podemos encontrar cuatro métodos, a saber:
 - a) **constructor(props)**: se ejecuta al crear la instancia del componente. Dentro deberíamos llamar a `super(props)` antes de cualquier otra instrucción, de lo contrario, `this.props` no estará definido, y puede llevar a errores. Luego, podemos inicializar el estado del componente, asignando un objeto a `this.state` donde especificamos los valores iniciales de algunas de las propiedades del componente. Finalmente, si tenemos métodos en

el componente, tenemos que enlazarlos en el constructor con bind para usarlos.

- b) **render()**: es el único método obligatorio en los componentes de clase. Cuando se llama, debe examinar a `this.props` y `this.state` y especificar aquello que queremos que aparezca en la UI, por ejemplo, un JSX, entre otros.
 - c) **componentDidMount()**: avisa de que el componente ya se ha montado en el DOM. Entre otras funciones, permite realizar peticiones HTTP o subscribir el componente a un `setInterval()` que nos devuelva datos de periódicos.
2. **Actualización**: la segunda fase ocurre cuando algún dato del componente (propiedad, estado...) se modifica y requiere que la UI se vuelva a generar para representar esos cambios. En esta segunda fase nos encontramos dos métodos:
- a. **render()**: redibuja el componente cuando detecta cambios en su estado o propiedades.
 - b. **componentDidUpdate()**: se ejecuta inmediatamente después de que la actualización del estado o las propiedades tengan lugar. Ideal para peticiones externas.
3. **Desmontado**: un solo método que se ejecuta antes de que un componente se elimine de la UI.
- a. **componentWillUnmount()**: este método se ejecuta antes de que el componente se elimine del DOM, y es útil para llevar a cabo tareas de limpieza.

Para entender mejor el ciclo de vida de un componente, vamos a generar un componente y usar los métodos de todas las etapas, para que podamos ver el uso de cada uno. En este caso vamos a crear un componente de clase llamado *Hora* que va a generar en el DOM la hora actual y dos botones, uno de ellos que va a iniciar un temporizador para que la hora se vaya actualizando cada segundo, y un botón para detenerla.

```
import React from 'react';

// Definimos una clase que extiende de React.Component

class Hora extends React.Component {
```

```
// Constructor - Se inicializan las props y el estado

constructor(props) {

    console.log(0, "El componente se inicializa, aun no
aparece en el DOM");

    // Llamamos al constructor del componente padre
    (React.Component)

    super(props);

    // Inicializamos el estado del componente

    this.state = {

        // 'hora' almacena la hora actual

        hora: new Date().toLocaleTimeString(),

        // 'temporizador' se inicializa como null

        temporizador: null

    }

    // Vinculamos los métodos 'iniciar' y 'detener' al
    contexto de 'this'

    this.iniciar = this.iniciar.bind(this);

    this.detener = this.detener.bind(this);

}

// Este método se ejecuta después de que el componente se
haya renderizado en el DOM

componentDidMount() {

    console.log(1, "El componente ya se encuentra en el
DOM");

}
```

```
// Este método se ejecuta cada vez que el estado o las
props del componente cambian (en realidad, justo antes)

componentDidUpdate(prevProps, prevState) {

    console.log(2, "El estado o las props del componente
han cambiado");

    console.log('prevProps: ', prevProps);

    console.log('prevState: ', prevState);

}

// Este método se ejecuta justo antes de que el componente
se desmonte y sea destruido

componentWillUnmount() {

    console.log(3, "El componente ha sido eliminado del
DOM");

}

// Este método inicia un temporizador que actualiza la
hora en el estado cada segundo

tictac() {

    this.temporizador = setInterval(() => {

        this.setState({

            hora: new Date().toLocaleTimeString()

        });

    }, 1000);

}

// Este método inicia el temporizador llamando al método
'tictac'
```

```
    iniciar() {  
        this.tictac();  
    }  
  
    // Este método detiene el temporizador  
  
    detener() {  
        clearInterval(this.temporizador);  
    }  
  
    // Este método renderiza el componente en el DOM  
  
    render() {  
        console.log(4, "El componente se dibuja en la IU o se  
redibuja");  
  
        return (  
            <>  
                <h2>{this.state.hora}</h2>  
                <button  
onClick={this.iniciar}>Iniciar</button>  
                <button  
onClick={this.detener}>Detener</button>  
            </>  
        );  
    }  
}  
  
// Exportamos la clase para que pueda ser utilizada en otros  
archivos
```

```
export default Hora;
```

Para los componentes funcionales se crean los **Hooks**, que permiten "engancharse" al estado y al ciclo de vida en componentes basados en funciones. Los más básicos son:

- **useState**: permite manipular el estado de un componente. En este sentido, se comportaría como el objeto *state* y la función *this.setState* de los componentes de clase.

Para poder usar este hook, primero tenemos que importarlo desde la librería de React:

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

Después de importar *useState* desde React, podemos usarlo en nuestro componente:

```
const [valor, setValor] = useState(0);
```

Esta línea de código usa una característica de JavaScript llamada desestructuración de arreglos. *useState(0)* inicializa un nuevo estado con el valor inicial 0 y devuelve un par de valores: el estado actual y una función para actualizarlo.

El código completo sería:

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

function Ejemplo() {

  const [valor, setValor] = useState(0);

  return (

    <div>

      <span>El valor del componente es {valor}</span>

      <button onClick={() => setValor(valor + 1)}>Aumentar valor</button>

    </div>

  )
}
```

```

    );
  }

export default Ejemplo;

/*Importamos React y la función useState desde la biblioteca React.
useState se utiliza para gestionar el estado del componente funcional.*/
import React, {useState} from "react";

const CambioColorEnMouse = () => {
  // Estado para controlar el color del componente
  // Esto sería una variable que se llamara en el style
  const [color, setColor] = useState('blue');

  // Manejador para el evento onMouseOver
  const handleMouseOver = () => {
    setColor('red');
  };

  // Manejador para el evento onMouseOut
  const handleMouseOut = () => {
    setColor('blue');
  };

  return (
    <p
      onMouseOver={handleMouseOver}
      onMouseOut={handleMouseOut}
      style={{ backgroundColor: color, padding: '20px', color: 'white' }}
    >
      Soy el componente, Pasa el ratón sobre mí
    </p>
  );
};

export default CambioColorEnMouse;

```

Este componente devuelve un *div* que contiene un texto y un botón. El texto refleja el valor actual del estado *valor*. Al hacer clic, se incrementa el valor del estado *valor* en 1 con la función *setValor*. Esta función se asigna al evento *onClick* del botón, que se ejecutará cada vez que el usuario haga clic en el botón.

Es importante saber que no debemos tratar a este hook como un objeto, tal y como ocurre con *this.state* en los componentes de clase, de forma que si necesitamos más de un valor cada uno debe ser almacenado en una variable diferente.

- **useEffect:** permite hacer uso del ciclo de vida en un componente funcional, y equivale a:
 - `componentDidMount()`

- componentDidUpdate()
- componentWillUnmount()

useEffect va a recibir como parámetro una función que se ejecutará cada vez que nuestro componente funcional se renderice, ya sea por cambios de estado o propiedades.

Para poder usar este hook, primero tenemos que importarlo desde la librería de React:

```
import React, { useEffect } from "react";
```

Para añadir un efecto que se ejecutará cada vez que nuestro componente se renderice, se debe pasar como parámetro una función al hook useEffect. En este caso, vamos a pasarle una función que por consola va a imprimir “Me he renderizado”:

```
import React, { useEffect } from "react";

function Efecto() {

  useEffect(function () {

    console.log("Me he renderizado!!!");

  });

  return <span>Este es un ejemplo del hook
useEffect.</span>;

}

export default Efecto;
```

Extensión Chrome:

<https://chromewebstore.google.com/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>

Módulo 3: Manejo de eventos

3.1. Eventos en React

3.1.1. Eventos de usuario

En React, los eventos son acciones que ocurren durante la interacción del usuario con una aplicación web construida con esta biblioteca. Estos eventos pueden ser cosas como hacer clic en un botón, escribir en un campo de entrada, o mover el ratón sobre un elemento.

En lugar de usar el manejo de eventos HTML tradicional, React utiliza su propio sistema de eventos sintéticos que normaliza las diferencias entre navegadores.

1. Definición del Evento: Primero, se define el tipo de evento que se desea manejar, como “onClick”, “onChange”, ...
2. Creación del Manejador de Eventos: Luego, se crea una función que se ejecutará cuando ocurra el evento. Este es el manejador de eventos.

El manejador de eventos en elementos de React, es muy similar a manejar eventos con elementos del DOM. Hay algunas diferencias de sintaxis:

- Los eventos de React se nombran usando camelCase, en vez de minúsculas.
 - Con JSX pasas una función como el manejador del evento, en vez de un string.
3. Vinculación del Manejador de Eventos al Elemento: Se vincula el manejador de eventos al elemento de la interfaz de usuario que debería responder al evento. Esto se hace mediante la asignación del manejador de eventos a la propiedad correspondiente en el elemento JSX.
 4. Ejecución del Manejador de Eventos: Cuando el usuario realiza la acción asociada al evento (como hacer clic en un botón), React ejecuta automáticamente el manejador de eventos especificado.
 5. Acceso a la información del Evento: La función del manejador de eventos recibe un objeto de evento como argumento. Este objeto contiene información sobre el evento, como la posición del ratón, el valor de un campo de entrada, etc...

3.1.2. Enlace de eventos y métodos

En React, los enlaces de eventos se utilizan para manejar interacciones del usuario, como clics de botones. Se declaran usando una sintaxis similar a HTML, pero con

camelCase. Los manejadores de eventos son funciones que responden a eventos, y se pueden definir en el componente y luego referenciar en los enlaces de eventos. También es posible pasar argumentos a los manejadores de eventos.

Además, los métodos en React son funciones definidas en los componentes para manejar lógica específica. Estos métodos pueden ser llamados desde el método render o desde los manejadores de eventos. Al utilizar métodos, se puede modularizar y organizar el código de manera más efectiva.

En resumen, 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.

```
import React from "react";

class MiComponente extends React.Component {
  /*A esta clase MiComponente le vamos a dar un funcionamiento
  | (onClick) que sera imprimir en consola lo que le diga*/
  funcionamientoDelOnClick = () => {
    console.log("Clic en el botón");
  };

  /*Lo que devuelve este componenete en este caso en un boton, pero puede
  | devolver la etiqueta que queramos*/
  render() {
    return (
      <button onClick={this.funcionamientoDelOnClick}>Haz clic aquí</button>
    );
  }
}

export default MiComponente;
```

3.2. Formularios en React

Podemos distinguir entre formularios controlados y no controlados en React. En cuanto a los controlados, se caracterizan porque los valores de los inputs son manejados por el state del componente. En este sentido, la propiedad state del componente es la *fuentes de verdad* que consideramos para saber qué valores tiene el formulario y con ello la única manera de actualizar este state es usando el hook useState. Con todo lo anterior, un campo de un formulario cuyos valores son controlados por react de esta forma, con los estados, es denominado *componente controlado*.

3.2.1. Control de componentes de formulario controlado

Un formulario controlado se refiere a un formulario cuyos elementos de entrada (campos de texto, selectores y casillas de verificación) están vinculados al estado del componente de React. Esto significa que el valor de cada elemento de entrada

del formulario es controlado por el estado de React, y cualquier cambio en estos elementos de entrada se refleja en el estado y viceversa.

La forma en la que se utiliza el estado en React es por medio de la función llamada **useState** que es parte de la API de **Hooks**. La intención de esta función es almacenar en un solo lugar el estado que el componente renderizará. Esto permite que los componentes reaccionen al cambio de estado y se vuelvan a renderizar para reflejar dicho cambio.

```
1  const [formData, setFormData] = useState({
2    nombre: '',
3    apellidos: '',
4    correo: '',
5    contraseña: '',
6  });
7
8  const [errors, setErrors] = useState({
9    nombre: '',
10   apellidos: '',
11   correo: '',
12   contraseña: '',
13 });
```

Aquí se utiliza el hook **'useState'** de React para inicializar dos estados en un componente funcional. Estos estados, **'formData'** y **'errors'**, se usan para manejar los datos del formulario y los mensajes de error asociados.

- **Inicialización de formData:**
 - **formData:** es un estado que almacena los valores de los campos del formulario.
 - **setFormData:** es la función que se utiliza para actualizar el estado **'formData'**. Se le proporciona un nuevo objeto que representa el nuevo estado, y React se encarga de manejar las actualizaciones.
- **Inicialización de errors:**
 - **errors:** es un estado que almacena los mensajes de error asociados a cada campo del formulario.
 - **setErrors:** es la función que se utiliza para actualizar el estado **'errors'**. Al igual que con **'setFormData'**, se le proporciona un nuevo objeto que representa el nuevo estado de errores.

3.2.2. Validación de formularios controlados

La validación de formularios controlados en React se refiere al proceso de asegurarse de que los datos ingresados por el usuario cumplen ciertos criterios o

reglas antes de ser enviados o procesados. En un formulario controlado, donde los elementos de entrada están vinculados al estado de React, puedes implementar la validación para garantizar la integridad y la corrección de los datos ingresados.

```
1  const handleChange = (e) => {
2    const { name, value } = e.target;
3    //Actualiza el estado del formulario con el nuevo valor del campo
4    setFormData((prevData) => ({
5      ...prevData,
6      [name]: value,
7    }));
8    //Limpia errores al cambiar el contenido del campo
9    setErrors((prevErrors) => ({
10     ...prevErrors,
11     [name]: '',
12   }));
13  };
```

La función **'handleChange'** es un controlador de eventos que se utiliza para manejar los cambios en los campos de un formulario.

- **Desestructuración de 'e.target':**
 - **'e':** es el evento que se produce cuando hay un cambio en el campo del formulario.
 - **'name':** es el atributo 'name' del elemento del formulario, que generalmente se usa para identificar el campo.
 - **'value':** es el nuevo valor del campo, es decir, el valor que ha sido ingresado por el usuario.
 - **'e.target':** es el elemento que desencadenó el evento, en este caso, el elemento del formulario que está siendo modificado.
- **Actualización del estado 'formData':**
 - **setFormData:** es la función proporcionada por el hook **'useState'** que se utiliza para actualizar el estado **'formData'**.
 - **'(prevData) => ({ ...prevData, [name]: value })':** es una función que toma el estado anterior (prevData), crea un nuevo objeto con todos los valores anteriores(...prevData) y actualiza el valor del campo correspondiente con el nuevo valor ([name]: value).
- **Limpieza de errores asociados al campo:**
 - **'setErrors':** es la función proporcionada por el hook **'useState'** que se usa para actualizar el estado **'errors'**.
 - **'(prevErrors) => ({ ...prevErrors, [name]: ' ' })':** es una función que actualiza el estado **'errors'**. Se establece el mensaje de error asociado al campo ([name]: ' ') como una cadena vacía, lo que indica que no hay errores en ese campo.

```

const handleSubmit = (e) => {
  e.preventDefault();

  //Objeto para almacenar los nuevos errores
  const newErrors = {};

  //Valida que todos los campos estén rellenos
  for (const key in formData) {
    if (!formData[key]) {
      newErrors[key] = 'Este campo es obligatorio';
    }
  }

  //Valida la contraseña
  if (formData.contraseña.length < 8) {
    newErrors.contraseña = 'La contraseña debe tener al menos 8 caracteres';
  } else if (
    ![A-Z]/.test(formData.contraseña) ||
    ![a-z]/.test(formData.contraseña) ||
    !/\d/.test(formData.contraseña)
  ) {
    newErrors.contraseña =
      'La contraseña debe contener al menos una mayúscula, una minúscula y un número';
  }

  //Actualiza el estado de los errores
  setErrors(newErrors);

  //Si no hay errores, puedes continuar con el envío del formulario
  if (Object.keys(newErrors).length === 0) {
    console.log('Datos del formulario:', formData);
  }
}

```

La función **'handleSubmit'** se usa para manejar el envío del formulario.

- **'e.preventDefault()'**: Evita que el formulario se envíe de manera tradicional, que suele recargar la página.
- **const newErrors = {}**: Es un objeto para almacenar los mensajes de error asociados a los campos del formulario.
- **Bucle for**:
 - Usa un bucle **'for...in'** para iterar sobre las claves (nombres de los campos) del objeto **'formData'**.
 - Si algún campo está vacío (!formData[key]), se agrega un mensaje de error al objeto **'newErrors'** indicando que ese campo es obligatorio.
- **Validación de la contraseña**:
 - Verifica si la longitud de la contraseña es menor que 8 caracteres. Si es así, se agrega un mensaje de error al objeto **'newErrors'**.
 - Luego, realiza una validación más compleja usando expresiones regulares para asegurarse de que la contraseña contenga al menos una mayúscula, una minúscula y un número. Si no cumple con estos requisitos, se agrega un mensaje de error al objeto **'newErrors'**.
- **setErrors(newErrors)**: Se usa para actualizar el estado **'errors'** con el nuevo objeto **'newErrors'** que contiene los mensajes de error.

- **Verificación final de errores (Último if):**

- Comprueba si el objeto **'newErrors'** está vacío. Si no hay mensajes de error, significa que el formulario es válido.
- En este caso, se muestra en la consola los datos del formulario usando **'console.log'**. En una aplicación real, esta sería la parte donde se enviarían los datos a un servidor o se realizarían otras acciones relacionadas con el envío del formulario.

```
return (
  <div className='contenedorFormulario' style={{ display: 'flex', justifyContent: 'center', alignItems: 'center' }}>
    <form onSubmit={handleSubmit}>
      /* Input para el nombre */
      <label style={{ fontSize: '20px' }}>
        Nombre:
        <input
          type="text"
          name="nombre"
          value={formData.nombre}
          onChange={handleChange}
          style={{ margin: '15px', height: '20px' }} />
        {errors.nombre && <span style={{ color: 'red' }}>{errors.nombre}</span>}
      </label>
      <br />

      /* Input para los apellidos */
      <label style={{ fontSize: '20px' }}>
        Apellidos:
        <input
          type="text"
          name="apellidos"
          value={formData.apellidos}
          onChange={handleChange}
          style={{ margin: '15px', height: '20px' }} />
        {errors.apellidos && <span style={{ color: 'red' }}>{errors.apellidos}</span>}
      </label>
      <br />

      /* Input para el correo electrónico */
      <label style={{ fontSize: '20px' }}>
        Correo electrónico:
        <input
          type="email"
          name="correo"
          value={formData.correo}
          onChange={handleChange}
          style={{ margin: '15px', height: '20px' }} />
        {errors.correo && <span style={{ color: 'red' }}>{errors.correo}</span>}
      </label>
      <br />

      /* Input para la contraseña */
      <label style={{ fontSize: '20px' }}>
        Contraseña:
        <input
          type="password"
          name="contraseña"
          value={formData.contraseña}
          onChange={handleChange}
          style={{ margin: '15px', height: '20px' }} />
        {errors.contraseña && <span style={{ color: 'red' }}>{errors.contraseña}</span>}
      </label>
      <br />

      /* Botón de envío del formulario */
      <button type="submit" style={{ height: '30px', width: '75px' }}>Enviar</button>
    </form>
  </div>
);
};

//Exporta el componente Formulario para su uso en otras partes de la aplicación
export default Formulario;
```



3.2.1. Control de componentes de formulario no controlado

En los formularios controlados que actúan como componentes funcionales, usamos `useState` para crear un estado que gestione los campos de los formularios, y cada vez que el usuario escribe en un campo, se llama a la función de actualización para ese campo.

En los formularios no controlados, los datos de los inputs se manejan en el DOM, en lugar del estado en React. De este modo, el DOM se vuelve la fuente de verdad de nuestros datos. La librería **React Hook Form** permite que los inputs sean gestionados por el DOM, lo que puede resultar en un mejor rendimiento y menos renderizados.

Con React Hook Form no se necesita crear un estado que gestione los campos de los formularios. En su lugar, utiliza la función **register** para registrar cada campo. Con ello, los valores de los campos se almacenan internamente en React Hook Form, no en el estado del componente. Para acceder a los valores de los campos, se pueden usar las funciones proporcionadas por la librería, como **handleSubmit** y **watch**.

Vamos a seguir un **ejemplo** de uso de esta librería para formularios. En primer lugar, para utilizar React Hook Form, es necesario **instalar** la librería en nuestro proyecto, usando el comando:

```
npm install react-hook-form
```

Luego, en nuestro componente, vamos a **importar** la función **useForm** del paquete `react-hook-form`:

```
import { useForm } from "react-hook-form"
```

Añadimos la línea **const { register, handleSubmit } = useForm()** que va a usar la función `useForm()` de la librería para obtener dos funciones: **register** y **handleSubmit**:

- **register**: se asigna a cada input, para hacer seguimiento de los cambios en los campos.
- **handleSubmit**: se utiliza para manejar el evento de envío del formulario, de forma que recibirá los datos del formulario si la validación del mismo es exitosa.

Para gestionar el envío del formulario usamos el evento **onSubmit** de la etiqueta `<form>`. En nuestro caso, vamos a decirle que ejecute la función **handleSubmit** de la librería. Si recibe los datos del formulario y la validación es exitosa (**aún no hecha**),

entonces vamos a pedirle que llame a su vez a una función que imprima los datos por consola. Es importante añadir una serie de campos en el formulario (nombre, edad...) y además, usar **register** para registrar esos campos, que vamos a llamar en cada input pasándole como argumento el nombre del campo que queremos gestionar. De este modo, cada input queda registrado y es reconocible dentro del formulario.


```

import { useForm } from "react-hook-form";

function Formulario() {
  const { register, handleSubmit } = useForm();
  const onSubmit = (data) => {
    console.log(data);
  }

  return (
    <div>
      <h2>Formulario</h2>
      <form onSubmit={handleSubmit(onSubmit)}>
        <div>
          <label>Nombre</label>
          <input type="text" {...register('nombre')} />
        </div>
        <div>
          <label>Dirección</label>
          <input type="text" {...register('direccion')} />
        </div>
        <div>
          <label>Edad</label>
          <input type="text" {...register('edad')} />
        </div>
        <div>
          <label>País</label>
          <select {...register('pais')}>
            <option value="es">España</option>
            <option value="it">Italia</option>
            <option value="fr">Francia</option>
          </select>
        </div>
        <input type="submit" value="Enviar" />
      </form>
    </div>
  );
}

export default Formulario;

```

3.2.2. Validación de formularios no controlados

La validación de formularios garantiza que los datos ingresados son correctos y cumplen ciertos criterios antes de enviarlos. Sin una validación, un formulario incompleto podría enviarse al servidor, causando problemas en la base de datos y errores.

Con el uso de React Hook Form, el proceso de validación se vuelve sencillo. Antes de nada, dentro de la llamada a **register** en cada campo, podemos meter un segundo parámetro, un **objeto** donde podemos añadir ciertas **propiedades**, como **required**, que puede ser true o false o **maxLength**, para controlar que sea obligatorio o su longitud.

Si un campo no cumple con su validación, no será posible enviar el formulario. Sin embargo, es importante **indicar** al usuario que existe un error de validación. Para ello, podemos acceder a otra herramienta de useForm, llamada **formState**, y en concreto, a **errors**, un objeto que contiene los errores de validación de los campos. El uso de herramientas pasa a **const (register, formState: { errors }, handleSubmit) = useForm()**.

Bajo cada input, podemos añadir una expresión de tipo **{errors.nombre?.type === 'required' && <p>El campo nombre es requerido</p>}** que verifica si existe un error required para el campo nombre. Si existe, muestra el mensaje “El campo nombre es requerido”.

Vamos a validar en nuestro código campos que sean obligatorios y que tengan una longitud máxima. Además, vamos a añadir otros campos que tienen validaciones más complejas, como sería el caso del email, que tenemos que usar el validador **pattern**.

```

function Formulario() {
  const { register, formState: { errors }, handleSubmit } = useForm();
  const onSubmit = (data) => {
    console.log(data);
  }
  return (
    <div>
      <h2>Formulario</h2>
      <form onSubmit={handleSubmit(onSubmit)}>
        <div>
          <label>Nombre</label>
          <input type="text" {...register('nombre', {
            required: true,
            maxLength: 10
          })}/>
          {errors.nombre?.type === 'required' && <p>El campo nombre es
            requerido</p>}
          {errors.nombre?.type === 'maxLength' && <p>El campo nombre
            debe tener menos de 10 caracteres</p>}
        </div>
        <div>
          <label>Dirección</label>
          <input type="text" {...register('direccion', {
            required: true,
          })}/>
          {errors.direccion?.type === 'required' && <p>El campo direccion
            es requerido</p>}
        </div>
        <div>
          <label>Email</label>
          <input type="text" {...register('email', {
            pattern: /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/ ,
          })}/>
          {errors.email?.type === 'pattern' && <p>El formato del
            email es incorrecto</p>}
        </div>
        <div>
          <label>Edad</label>
          <input type="text" {...register('edad')}/>
        </div>
        <div>
          <label>País</label>
          <select {...register('pais')}>
            <option value="es">España</option>
            <option value="it">Italia</option>
            <option value="fr">Francia</option>
          </select>
        </div>
        <input type="submit" value="Enviar" />
      </form>
    </div>
  );
}
export default Formulario;

```

Con esta forma de añadir las validaciones no podemos recoger todas las posibles condiciones que pueden cumplir nuestros campos, sobre todo en formularios más **personalizados**. Por ejemplo, si queremos que la edad esté comprendida entre un rango de números.

Para solucionarlo, podemos crear un fichero js donde añadir una función propia de validación, para luego poder usarla en nuestros campos de formularios, importándola.

Vamos a seguir con el caso de la edad, y vamos a crear una función en nuestro fichero que nos va a devolver true si la edad está comprendida entre 18 y 65, y false si no lo está.

```
const edadValidator = (value) => {  
  return value >= 18 && value <= 65;  
}  
export {edadValidator}
```

Luego, vamos a usarla en nuestro componente Formulario, para ello, vamos a importarla, y vamos a usarla dentro de register como valor a la palabra reservada **validate**:

```

import { useForm } from "react-hook-form";
import { edadValidator } from "../validator";
function Formulario() {
  const { register, formState: { errors }, handleSubmit } = useForm();
  const onSubmit = (data) => {
    console.log(data);
  }
  return (
    <div>
      <h2>Formulario</h2>
      <form onSubmit={handleSubmit(onSubmit)}>
        <div>
          <label>Nombre</label>
          <input type="text" {...register('nombre', {
            required: true,
            maxLength: 10
          })}/>
          {errors.nombre?.type === 'required' && <p>El campo nombre es
            requerido</p>}
          {errors.nombre?.type === 'maxLength' && <p>El campo nombre
            debe tener menos de 10 caracteres</p>}
        </div>
        <div>
          <label>Dirección</label>
          <input type="text" {...register('direccion', {
            required: true,
          })}/>
          {errors.direccion?.type === 'required' && <p>El campo direccion
            es requerido</p>}
        </div>
        <div>
          <label>Email</label>
          <input type="text" {...register('email', {
            pattern: /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/ ,
          })}/>
          {errors.email?.type === 'pattern' && <p>El formato del
            email es incorrecto</p>}
        </div>
        <div>
          <label>Edad</label>
          <input type="text" {...register('edad', {
            validate: edadValidator
          })}/>
          {errors.edad && <p>La edad debe estar entre
            18 y 65</p>}
        </div>
      </form>
    </div>
  );
}

```

Módulo 4: Conceptos avanzados de componentes

4.1. Composición de componentes

En el contexto de **desarrollo de software**, la **composición de componentes** 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**, la composición de componentes es un principio esencial que impulsa la construcción de interfaces de usuario (UI) reutilizables y mantenibles. La reutilización de componentes y la aplicación de patrones de composición son prácticas fundamentales en el desarrollo con React.

4.1.1. Reutilización de componentes

La reutilización de componentes es un principio fundamental en el desarrollo de software que implica aprovechar piezas de código existentes en lugar de escribir nuevo código desde cero. Algunos beneficios clave de la reutilización de componentes son:

- **Eficiencia en el Desarrollo:** Al reutilizar componentes, se reduce el tiempo y los recursos necesarios para desarrollar nuevas funcionalidades.
- **Consistencia:** El uso consistente de componentes ayuda a mantener la coherencia en toda la aplicación.
- **Mantenibilidad:** Los componentes bien diseñados son más fáciles de mantener, ya que se pueden actualizar y corregir sin afectar otras partes del sistema.

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.

En **React** se centra en la creación de componentes independientes y modulares que pueden ser fácilmente integrados en diferentes partes de la aplicación. Esto se logra mediante la creación de componentes funcionales o de clase que encapsulan ciertas funcionalidades y se pueden reutilizar en múltiples lugares.

Ejemplo de reutilización de componentes en React:

1º Creamos nuestro componente.

```
JS App.js  button.jsx X
src > components > button.jsx > ...
1  import React from 'react';
2
3  const Button = ({ onClick, label }) => (
4    <button onClick={onClick}>{label}</button>
5  );
6
7  export default Button;
```

2º Podemos utilizarlo cuantas veces queramos en nuestro proyecto.

```
JS App.js  X  button.jsx
src > JS App.js > App
1  import './App.css';
2  import React from 'react';
3  import Button from './components/button';
4
5
6  function App() {
7
8    return(
9      <div>
10     <h1>Aplicación con Botones</h1>
11     <Button onClick={() => alert('Click 1')} label="Botón 1" />
12     <Button onClick={() => alert('Click 2')} label="Botón 2" />
13   </div>
14
15   )}
16
17   export default App;
```

4.1.2 Patrones de composición

Los 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 en React:**

El patrón de 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.

Cogemos un componente creado, en este caso un botón:

```
src > components > button.jsx > ...  
1   import React from 'react';  
2  
3   const Button = ({ onClick, label }) => (  
4     <button onClick={onClick}>{label}</button>  
5   );  
6  
7   export default Button;
```

Creamos el patrón de composición de decorador, donde marcamos que muestre por consola, que el componente utilizado, se ha montado al iniciar la aplicación.

```
> components > withLogger.jsx > ...  
1   import React from 'react';  
2  
3   const withLogger = (WrappedComponent) => {  
4     return class extends React.Component {  
5       componentDidMount() {  
6         console.log(`Componente ${WrappedComponent.name} montado.`);  
7       }  
8  
9       render() {  
10        return <WrappedComponent {...this.props} />;  
11      }  
12    };  
13  };  
14  
15  export default withLogger;
```


En nuestra app.js, creamos un componente del patrón de decorador, añadiendo el componente botón. Y en consola, mostrará si el botón se ha montado.

```
> JS App.js > ...
1 import './App.css';
2 import React from 'react';
3 import Button from './components/button';
4 import withLogger from './components/withLogger';
5
6 const DecoratedButton = withLogger(Button);
7
8 function App() {
9
10
11
12   return (
13     <div>
14       <h1>Aplicación con Decorador</h1>
15       <DecoratedButton onClick={() => alert('click')} label="Click Me" />
16     </div>
17   );
18 }
19
20 export default App;
```

- **Composición de Componentes:**

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

En este caso creamos un componente que contiene el “programa” y otro que es el que pone la parte visual.

```

src > components > composicionComponentes.jsx > ...
1  import React, { useState } from 'react';
2
3  // Componente contenedor que maneja la lógica y el estado
4  const ComponenteContenedor = () => {
5    const [contador, setContador] = useState(0);
6
7    const incrementarContador = () => setContador(contador + 1);
8
9    return (
10     <div>
11       <h2>Componente Contenedor</h2>
12       <p>Contador: {contador}</p>
13       <button onClick={incrementarContador}>Incrementar</button>
14       <ComponentePresentacional contador={contador} />
15     </div>
16   );
17 };
18
19 // Componente presentacional que se centra en la representación visual
20 const ComponentePresentacional = ({ contador }) => (
21   <div>
22     <h2>Componente Presentacional</h2>
23     <p>Recibiendo el contador: {contador}</p>
24   </div>
25 );
26
27 // Uso de ambos componentes
28 const ComponentePresentacionalEjemplo = () => <ComponenteContenedor />;
29
30 export default ComponentePresentacionalEjemplo;

```

- **Render Props:**

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

```

src > components > renderProps.jsx > ...
1  import React from 'react';
2
3  // Componente padre que proporciona la función de renderizado como prop
4  const RenderPropsPadre = ({ render }) => {
5    const data = 'Datos desde RenderPropsPadre';
6    return <div>{render(data)}</div>;
7  };
8
9  // Componente hijo que utiliza la función de renderizado
10 const RenderPropsHijo = ({ data }) => <p>{data}</p>;
11
12 // Uso del componente padre e hijo
13 const RenderPropsEjemplo = () => (
14   <RenderPropsPadre render={({data}) => <RenderPropsHijo data={data} /> } />
15 );
16
17 export default RenderPropsEjemplo;
18

```

- **Hooks y Composición:**

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

```

src > components > hooksComposicion.jsx > ...
1  import React, { useState, useEffect } from 'react';
2
3  const HooksComposicionEjemplo = () => {
4    const [contador, setContador] = useState(0);
5    const [mensaje, setMensaje] = useState('');
6
7    useEffect(() => {
8      if (contador > 5) {
9        setMensaje('¡Contador mayor que 5!');
10     } else {
11       setMensaje('');
12     }
13   }, [contador]);
14
15   const incrementarContador = () => setContador(contador + 1);
16
17   return (
18     <div>
19       <p>Contador: {contador}</p>
20       <button onClick={incrementarContador}>Incrementar</button>
21       <p>{mensaje}</p>
22     </div>
23   );
24 };
25
26 export default HooksComposicionEjemplo;

```

4.2.Contexto en React

Es una característica que permite pasar datos a través del árbol de componentes sin tener que pasar explícitamente las propiedades a cada nivel. Esto es especialmente útil cuando se trabaja con componentes anidados que necesitan compartir información sin necesidad de pasarla a través de todos los niveles intermedios.

“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.”

4.2.1 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.

```
const MiContexto = React.createContext();
```

4.2.2 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.

Imagina que tienes un dato, por ejemplo, el nombre del usuario, que deseas compartir entre diferentes componentes en tu aplicación. Puedes utilizar el `Provider` para “envolver” esos componentes y hacer que ese dato esté disponible para todos ellos.

```
Unset
const UserContext = React.createContext();

function App() {
  const nombreUsuario = "John";

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

En este ejemplo, el componente `App` actúa como el proveedor del contexto (`Provider`). Está proporcionando el contexto `UserContext` con el valor `nombreUsuario`. Todos los componentes hijos de `App` pueden acceder a este valor si están envueltos por `UserContext.Provider`.

- Los componentes que desean acceder al valor proporcionado por el proveedor pueden hacerlo utilizando el `Consumer` o el hook `useContext`.

Usando `Consumer`;

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

Usando `useContext`;

```
Unset
import React, { useContext } from 'react';

function ComponenteConsumidor() {
  const nombreUsuario = useContext(UserContext);

  return <p>Hola, {nombreUsuario}</p>;
}
```

En ambos casos, el componente `ComponenteConsumidor` está consumiendo el contexto `UserContext` para acceder al valor proporcionado por el `Provider` (`nombreUsuario` en este caso) y utilizarlo en su renderizado.

Ejemplo final con provider y useContext;

```
Unset
import React, { useContext } from 'react';

const UserContext = React.createContext();

function SaludoUsuario() {
  const nombreUsuario = useContext(UserContext);

  return <p>Hola, {nombreUsuario}</p>;
}

function App() {
  const nombreUsuario = "John";

  return (
    <UserContext.Provider value={nombreUsuario}>
      <SaludoUsuario />
    </UserContext.Provider>
  );
}
```

Módulo 5: Conceptos avanzados de componentes

5.1. Uso de React Router

React Router es la librería más popular para gestionar la navegación en aplicaciones internas hechas con dicha herramienta. Para su configuración básica, se envuelven los componentes en el componente **BrowserRouter** en el punto de entrada. Luego, utiliza **Route** para asociar componentes a rutas específicas en el componente donde deseas gestionar las rutas. Para crear enlaces a estas rutas, se usa el componente **Link**. Puedes hacer rutas dinámicas pasando parámetros. La navegación programática es posible con el objeto **history**. React Router también admite redirecciones y rutas anidadas.

En resumen, React Router proporciona 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.

5.1.1. Configuración en React

Instalación:

1. Instala React Router en tu proyecto:

‘Comando para poner en la consola’

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

Configuración en tu aplicación:

2. Wrap de Componentes en BrowserRouter:

En el componente principal (por lo general `index.js` o `App.js`), envuelve tus componentes con `BrowserRouter`:

```
JavaScript
import React from 'react';
import { BrowserRouter as Router } from 'react-router-dom';
import App from './App';

ReactDOM.render(
  <Router>
    <App />
  </Router>,
  document.getElementById('root')
);
```

Uso de Rutas:

3. Configuración de Rutas:

En el componente donde deseas configurar tus rutas, utiliza componentes como `Route` y `Switch` de React Router.

```
JavaScript
import React from 'react';
import { Route, Switch } from 'react-router-dom';

import Inicio from './Inicio';
import AcercaDe from './AcercaDe';
import Contacto from './Contacto';

const App = () => {
  return (
    <div>
      <Switch>
        <Route path="/acerca-de" component={AcercaDe} />
        <Route path="/contacto" component={Contacto} />
        <Route path="/" component={Inicio} />
      </Switch>
    </div>
  );
};

export default App;
```


- `'Route'`: Define una ruta y especifica el componente que se renderizará cuando la URL coincida.
- `'Switch'`: Renderiza el primer `'Route'` que coincide con la URL.

Navegación:

4. Creación de Enlaces:

Utiliza el componente `'Link'` para crear enlaces a tus rutas:

```
JavaScript
import React from 'react';
import { Link } from 'react-router-dom';

const Menu = () => {
  return (
    <nav>
      <ul>
        <li><Link to="/">Inicio</Link></li>
        <li><Link to="/acerca-de">Acerca de</Link></li>
        <li><Link to="/contacto">Contacto</Link></li>
      </ul>
    </nav>
  );
};

export default Menu;
```

- `'Link'`: Crea enlaces a rutas definidas.

5.1.2. Rutas anidadas y parámetros

El anidamiento de rutas consiste en agrupar una serie de rutas react que empiezan con la misma URL para simplificar nuestro código. Podemos anidar rutas con React Router que se despeguen de la misma página de nuestra aplicación.

¿Cómo anidar rutas con React Router?

El anidamiento de rutas consiste en agrupar una serie de rutas react que empiezan con la misma URL para simplificar nuestro código. Podemos anidar rutas con React Router que se despeguen de la misma página de nuestra aplicación.

La forma en la que se anidan rutas es muy parecida a la forma en la que se crean rutas simples. Para ello tendremos que especificar dentro del componente **<Route>** que actuará como “Padre” otros componentes **<Route>** que actuarán como “hijos”. Estos “hijos” comparten el **prefijo** en la ruta, el cuál es la ruta “padre”.

También deberemos conocer un nuevo componente que también nos proporciona **react-router-dom**, el componente **<Outlet>**. El componente Outlet se utiliza en escenarios de enrutamiento anidado.

Se trata de una etiqueta JSX de cierre automático que representa el componente Outlet de react-router-dom. Cuando un componente de Ruta padre tiene Rutas hijo, la Ruta padre utiliza el componente Outlet para renderizar la Ruta hijo coincidente. En otras palabras, el componente Outlet sirve como marcador de posición para las rutas hijas de la ruta padre.

En resumen, esta línea de código utiliza el componente Outlet para mostrar la ruta hija adecuada en función de la ruta actual en una configuración de rutas anidadas.

Para entender cómo funciona el enrutamiento anidado, veremos un ejemplo práctico:

Tenemos un componente llamado **Menu.js**:

```
function Menu() {
  return (
    <nav>
      <ul style={{ display: "flex", listStyle: "none" }}>
        <li style={{ marginRight: "10px" }}><NavLink
to="/">Inicio</NavLink></li>
        <li style={{ marginRight: "10px" }}><NavLink
to="/registro">Registro</NavLink></li>
        <li style={{ marginRight: "10px" }}><NavLink
to="/login">Login</NavLink></li>
        <li style={{ marginRight: "10px" }}><NavLink
to="/rutas">Rutas anidadas</NavLink></li>
      </ul>
    </nav>
  );
}
```

Este componente generará un listado en el que cada punto será un componente **<NavLink>** también proporcionado por **react-router-dom**. Su funcionamiento es similar al del componente **<Link>**, es similar a utilizar la etiqueta **<a>** en HTML pero evitando el refresco de la página completa cuando el link es clicado.

En este caso contamos con 4 **<NavLink>** con los nombres Inicio, Registro, Login y Rutas anidadas. Nos centraremos en este último.

En el componente **Public.js** se llama a **<Menu>** dentro del elemento de react-router-dom; **<Router>**:

```

function Public() {
  return (
    <div>
      <h1>Rutas y Rutas Anidadas</h1>
      <Router>
        <Menu />
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/home" element={<Navigate to="/" />} />
          <Route path="/registro" element={<Registro />} />
          <Route path="/login" element={<Login />} />
          <Route path="*" element={<NotFound />} />
          <Route path="/rutas/" element={<RutasAnidadas />} >
            <Route path="frontend" element={<p>FrontEnd
              ✨</p>}/>

            <Route path="backend" element={<p>BackEnd 🤖</p>}/>
            <Route path="bd" element={<p>BD 💾</p>}/>
            <Route path="seguridad" element={<p>Cyber Security
              🔒</p>}/>

            <Route path="cloud" element={<p>Cloud ☁️</p>}/>
            <Route path="architecture" element={<p>Architecture
              🏗️</p>}/>

            <Route path="testing" element={<p>Testing 🧪</p>}/>
            <Route path="hacking" element={<p>Hacking 🧑‍🔧</p>}/>
          </Route>
        </Routes>
      </Router>
    </div>
  );
}

```

El componente menu incluye el componente **<Routes>** donde se declararán las diferentes rutas **<Route>** que nos dirigirán a diferentes sitios en función del camino “**path**” que tengan definido.

Centrándonos nuevamente en el componente que nos dirige a RutasAnidadas, podemos observar la **<Route>** “Padre” (/rutas/) con sus diferentes **<Route>** “Hijas”. Las rutas hijas comparten el prefijo /rutas/.

Por último tenemos el componente **RutasAnidadas.js**:

```

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

```

```

        <li><Link to={"BD"}>BD</Link></li>
        <li><Link to={"seguridad"}>Cyber Security</Link></li>
        <li><Link to={"cloud"}>Cloud</Link></li>
        <li><Link to={"architecture"}>Architecture</Link></li>
        <li><Link to={"testing"}>Testing</Link></li>
        <li><Link to={"hacking"}>Hacking</Link></li>
      </ul>
      <Outlet/>
    </div>
  );
}

```

Es aquí donde observamos el componente **<Outlet>** que forma parte de la biblioteca **react-router-dom**.

Se trata de una etiqueta JSX de cierre automático que representa el componente Outlet de react-router-dom. Cuando un componente de Ruta padre tiene Rutas hijo, la Ruta padre utiliza el componente Outlet para renderizar la Ruta hijo coincidente. En otras palabras, el componente Outlet sirve como marcador de posición para las rutas hijas de la ruta padre. En resumen, esta línea de código utiliza el componente Outlet para mostrar la ruta hija adecuada en función de la ruta actual en una configuración de rutas anidadas.

Haciendo un breve resumen sobre el enrutamiento anidado, así es cómo quedaría nuestra aplicación:

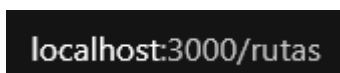
Estos serían los cuatro enlaces que genera el componente **Menu.js**:



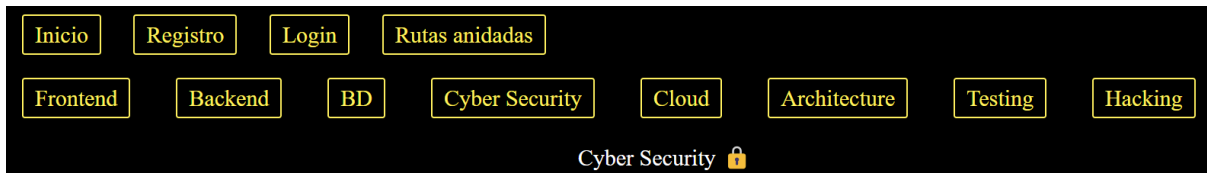
Al clicar en “**Rutas anidadas**” se nos despliegan los siguientes enlaces, generados por el componente **RutasAnidadas.js**:



En este momento, la url se vería tal que así:



Si clicamos en cualquiera de los enlaces, empezaría a funcionar el componente **<Outlet>** definido en **RutasAnidadas.js** y entraría en las rutas que hemos anidado en **Public.js**. Por lo que en caso de, por ejemplo, clicar en “**Cyber Security**”, se nos mostrará el siguiente **<p>** en pantalla:



En este momento, la url se vería tal que así:

`localhost:3000/rutas/seguridad`

¿Qué son los parámetros en las rutas?

Los parámetros en las rutas permiten capturar valores dinámicos en las URL, lo que facilita la construcción de componentes reutilizables y la manipulación de datos basada en la ruta.

Veámoslo a través de un ejemplo:

```
// 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>
);

export default App;
```

Enlaces: En la sección de ``, se crean dos enlaces utilizando la etiqueta `<Link>`. Cada enlace tiene una URL específica, que incluye un segmento variable `:username`. Estos son los parámetros de la ruta que serán capturados dinámicamente.

Ruta con Parámetro: La `<Route>` se configura con la ruta `/user/:username`. El `:username` en la ruta indica que esta parte de la ruta es un parámetro dinámico. Cuando la URL coincide con esta ruta, React Router captura el valor proporcionado en lugar de `:username` y lo pasa como parte del objeto `match.params` al componente `UserProfile`.

```
// 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>
);
```

Componente UserProfile: El componente UserProfile recibe match como una prop. match.params.username contiene el valor dinámico proporcionado en la URL. Por ejemplo, si haces clic en "Usuario John", la URL se convierte en "/user/john", y match.params.username dentro de UserProfile será igual a "john".

La clave aquí es entender que :username en la ruta es un marcador de posición para un valor dinámico, y React Router se encarga de pasar ese valor al componente correspondiente a través del objeto match.params.

Módulo 6: Manejo de Estado Global

6.1 Introducción a la gestión de estado global

El concepto de *estado* es fundamental para desarrollar aplicaciones en React. Consiste en una forma de almacenar y manipular datos que pueden cambiar con el tiempo y afectar el comportamiento y la representación de los componentes en React.

Podemos distinguir entre estado local y estado global. El estado local se refiere a los datos que pertenecen a un componente específico y que sólo pueden ser accedidos y modificados dentro de ese componente. El estado global se refiere a los datos que pueden ser accedidos y modificados por cualquier componente de la aplicación.

El estado global es especialmente útil en aplicaciones mas grande donde los datos necesitan ser compartidos entre muchos componentes a diferentes niveles de anidación. En lugar de pasar los datos a través de props de un componente padre a un componente hijo, puedes almacenar los datos en un estado global y permitir que cualquier componente acceda o modifique ese estado según sea necesario. En cambio, el estado local es útil en aplicaciones pequeñas con pocos componentes.

El manejo de estado es el proceso de seguimiento y actualización del estado de una aplicación.

Si hablamos de estado local, podemos entenderlo como el proceso de seguimiento y actualización del estado específico a un componente. Por ejemplo, el valor de un campo de entrada de texto de un formulario, o si un modal está visible o no.

El manejo del estado local se realiza de forma habitual usando el hook `useState` en React. Este hook permite añadir estados a los componentes funcionales de React y actualizar ese estado a lo largo del tiempo. Por ejemplo, para clicar un botón:

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

function Button() {

  // Inicializar el estado local

  const [isClicked, setIsClicked] = useState(false);

  // Función para manejar el clic del botón

  const handleClick = () => {

    setIsClicked(!isClicked);

  };

  return (

    <button onClick={handleClick}>

      {isClicked ? 'Botón ha sido clicado' : 'Botón no ha sido clicado'}

    </button>

  );

}
```

En este código vemos que un estado local llamado *isClicked* que rastrea si el botón ha sido clicado o no. Cuando este botón es clicado, la función *handleClick* se ejecuta, actualizando el estado de la función *isClicked* al valor opuesto de su estado actual.

El manejo del estado global es el proceso de seguimiento y actualización del estado que es compartido entre múltiples componentes. En otras palabras, el estado es accesible por todos los componentes de la jerarquía de componentes de React.

Existen varias formas de manejar el estado global en React, entre ellas, destacamos dos: Context API, característica integrada de React, y Redux, librería

6.1.1 Context API

¿Qué es?

Context es una forma de compartir datos entre componentes sin tener que pasarlos a través del árbol de componentes. Esto nos permite solucionar un problema que pasaba al compartir demasiados props entre componentes, el **Prop Drilling**, lo que hace que la aplicación se vuelva difícil de mantener y escalar.

También, con la ayuda de los Hooks, nos permite tener un control del estado global o para compartir datos entre componentes que no están conectados directamente.

Cuando usar Context API

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

Se usa principalmente cuando algunos datos tienen que ser accesibles por muchos componentes en diferentes niveles de anidamiento. Se debe utilizar con moderación porque hace que la reutilización de componentes sea más difícil.

¿Cómo usar Context API?

Crear un Context

Para empezar a usar la Context API, primero necesitamos crear un contexto. Esto se hace usando **React.createContext()**.

```
Const MyContext = React.createContext(defaultValue);
```

El argumento *defaultValue* es el valor que se utilizará si el contexto no está envuelto en un **Provider**. Es opcional y normalmente se deja vacío.

Context.Provider

Una vez creado un *Context*, podemos usar el componente **Context.Provider** para envolver partes de nuestro árbol de componentes que necesiten acceso a los valores del Contexto. El componente *Context.Provider* acepta un **prop value**, que es el valor actual del contexto.

```
<MyContext.Provider value={/*Algún valor*/}>
  {/* Children */}
</MyContext.Provider>
```

Todos los componentes hijos del *Context.Provider* tendrán acceso al valor del contexto.

Esto normalmente lo tendrás que envolver en el componente raíz de tu aplicación, o en aquella parte de tu *app* dónde tenga sentido tener acceso a la información que guardes del contexto.

Context.Consumer

Dentro de los componentes hijos del *Context.Provider*, podemos usar el componente **Context.Consumer** para acceder al valor del contexto.

```
<MyContext.Consumer>
  {value => /*renderiza algo basado en el valor del contexto*/}
</MyContext.Consumer>
```

Hook useContext

Este hook se usa para acceder al valor del contexto. Esto puede hacer que nuestro código sea más limpio y fácil de entender.

```
const value = useContext(MyContext);
```

Ejemplo práctico

Ejemplo práctico de cómo usar la *Context API* para compartir el estado global. Imagina que estamos construyendo una aplicación de comercio electrónico y queremos compartir la información del carrito de compras entre varios componentes.

Primero creamos un *Context* para el carrito de compras:

```
const CartContext = React.createContext();
```

Luego en nuestro componente principal, utilizaremos el **CartContext.Provider** para compartir el estado del carrito de compras:

```
Function App() {
  const [cart, setCart] = useState([]);

  return {
    <CartContext.Provider value={{ cart, setCart }}>
      /* otros componentes */
    </CartContext.Provider>
  );
}
```

```
}
```

Dentro de otros componentes, podemos usar *CartContext.Consumer* o *useContext* para acceder al estado del carrito de compras:

```
Function Cart() {
  const { cart } = useContext(CartContext);

  return {
    <div>
      {cart.map(item => {
        <div key={item.id}>{item.name}</div>
      })}
    </div>
  );
}
```

La Context API proporciona una forma flexible de compartir el estado global en una aplicación React. Sin embargo, también tiene sus limitaciones.

Por ejemplo, puede ser menos eficiente para grandes aplicaciones con muchos cambios de estado, y puede ser más difícil de manejar para estados complejos o lógicas de negocio. Además, el uso excesivo de la Context API puede hacer que el código sea más difícil de entender y mantener, ya que el estado se comparte implícitamente a través del árbol de componentes.

A pesar de estas limitaciones, la Context API sigue siendo una buena opción para muchos casos de uso, especialmente cuando se trata de compartir datos que no cambian con frecuencia o que no requieren lógicas complejas, como el usuario autenticado actual o el tema preferido.

6.1.2 Redux

Redux es una **biblioteca** de JavaScript que se utiliza para **manejar el estado** de una aplicación en aplicaciones de interfaz de usuario, como las creadas con React, aunque no exclusivamente. En este sentido, es posible utilizar Redux en otras plataformas como Angular.

Redux nos permite:

1. **Manejar el estado global de la aplicación.** Redux mantiene el estado de toda la aplicación en un solo lugar, lo que puede llegar a facilitar su seguimiento y gestión.

2. **Actualizar el estado de manera predecible.** Con Redux, las actualizaciones al estado se hacen de una forma que se puede prever y rastrear fácilmente en la aplicación.
3. **Separar la lógica de la interfaz de usuario y la lógica de estado.** Redux permite separar ambas lógicas, lo que puede hacer que el código sea más fácil de entender.
4. **Depurar y probar fácilmente la aplicación.** Redux hace que sea más fácil depurar y probar la aplicación, ya que puedes rastrear cada cambio de estado a la acción que lo causó.

Redux se basa en tres principios:

1. **Una sola fuente de verdad.** Todo el estado de la aplicación está contenido en un único **store**. Este store es un objeto JS que contiene todo el estado de la aplicación.
2. **El estado es de solo lectura.** La única forma de modificar el estado es emitiendo una **acción**. Una acción es un objeto JS que describe qué cambio se debe hacer en el estado.
3. **Los cambios se hacen mediante funciones puras.** Para controlar como el store es modificado por las acciones se usan **reducers** puros. Estos son funciones puras que reciben el estado actual de la aplicación y la acción a realizar y devuelven un nuevo estado. Podemos tener un solo reducer en toda nuestra aplicación o varios.

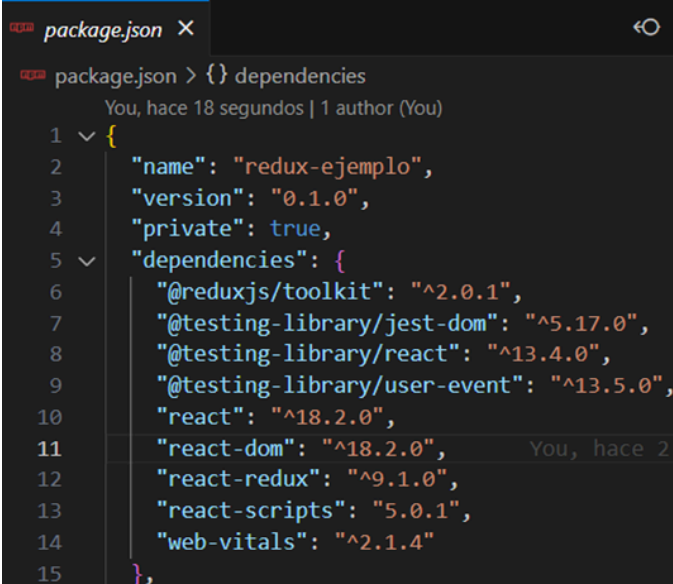
De todas las herramientas de Redux disponibles, vamos a usar **Redux Toolkit**, que proporciona un conjunto de utilidades que simplifican el desarrollo con Redux. Incluye funciones para configurar Redux y crear reductores y acciones. Además, vamos a usar **React Redux**, que es el paquete oficial que permite a la aplicación en React interactuar con Redux. Permite acceder a los Hooks de Redux, que podemos usar para acceder al estado de Redux y despachar acciones desde nuestros

componentes. Estas dos bibliotecas trabajan juntas para manejar el estado de una aplicación en React con Redux eficiente y predecible.

Para usar Redux en React, tenemos que instalar esta librería desde la terminal, con el comando:

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

Podemos comprobar que nuestras bibliotecas se han instalado correctamente desde **package.json**:



```
package.json > {} dependencies
You, hace 18 segundos | 1 author (You)
1 {
2   "name": "redux-ejemplo",
3   "version": "0.1.0",
4   "private": true,
5   "dependencies": {
6     "@reduxjs/toolkit": "^2.0.1",
7     "@testing-library/jest-dom": "^5.17.0",
8     "@testing-library/react": "^13.4.0",
9     "@testing-library/user-event": "^13.5.0",
10    "react": "^18.2.0",
11    "react-dom": "^18.2.0",
12    "react-redux": "^9.1.0",
13    "react-scripts": "5.0.1",
14    "web-vitals": "^2.1.4"
15  }
```

Ejemplo de uso

Antes de empezar, comentar que suele entenderse **slice** en Redux como una porción o segmento del estado global de la aplicación. En otras palabras, es una forma de dividir el estado de una aplicación en partes mas manejables basadas en el área de la aplicación que se trate.

Este ejemplo va a interactuar con una API externa para acceder a cierta información sobre un usuario y mostrar esa información en un componente llamado Header. La idea es que esta información se almacene en el estado global de la aplicación usando Redux Toolkit y haciendo uso de los principales conceptos de la biblioteca: store, reducers y actions.

1. Definir nuestro Slice.

Dentro de src vamos a crear una carpeta llamada redux y dentro un archivo llamado userSlice.js.

```

redux > JS userSlice.js > default
import { createSlice } from "@reduxjs/toolkit";

export const userSlice = createSlice({
  name: "user",
  initialState: {
    name: "",
    username: "",
    email: ""
  },
  reducers: {
    addUser: (state, action) => {
      const { name, username, email } = action.payload;
      state.name = name;
      state.username = username;
      state.email = email;
    },
  },
});

export const { addUser } = userSlice.actions;
export default userSlice.reducer;

```

Aquí:

- Se importa la función `createSlice` de `@reduxjs/Toolkit`.
- Se utiliza `createSlice` para definir un slice llamado `userSlice`. Este `userSlice` incluye el nombre del slice, a saber, **user**, el estado inicial, con las propiedades **name**, **username** y **email** inicializadas como vacías, y el reducer, donde se definen las acciones, en este caso, **addUser**, que toma el estado actual y una acción, extrae `name`, `username` y `email` del `payload` de la acción, y actualiza el estado con estos valores.
- Se exporta la acción `addUser` para poder ser utilizada en otros lugares de la aplicación.
- Se exporta el reductor generado automáticamente por Redux para ser usado en el store.

1. Crear nuestro store

Dentro de nuestra carpeta `redux` vamos a crear un archivo llamado **store.js** que va a contener:

```
import { configureStore } from "@reduxjs/toolkit";
import userReducer from "./userSlice";

export const store = configureStore({
  reducer: {
    user: userReducer,
  }
});
```

Aquí:

- Se importa la función `configureStore` de `@reduxjs/Toolkit`.
- Se importa `userReducer` desde `userSlice.js`. Este es el reductor que se generó automáticamente cuando creamos el slice de usuario con `createSlice` en `userSlice.js`
- Se configura el store de Redux usando `configureStore`. Aquí, se especifica el reductor `userReducer` bajo el campo `reducer`. Esto establece que el reductor `userReducer` maneja el estado del slice llamado `user`, que definimos en `userSlice.js`.

1. Integrar el store en la aplicación mediante un Provider

Un Provider es un componente proporcionado por `react-redux` que se utiliza para envolver la aplicación en el nivel superior. Proporciona el store de Redux a todos los componentes descendientes, permitiéndoles acceder al estado global y despachar acciones.

```
JS index.js > ...
You, hace 2 horas | 1 author (You)
✓ import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import {Provider} from "react-redux";
import {store} from "./redux/store";

const root = ReactDOM.createRoot(document.getElementById('root'));
✓ root.render(
✓   <React.StrictMode>
✓   | <Provider store={store}>
   | | <App />
   | </Provider>
   </React.StrictMode>
);

reportWebVitals();
```

Aquí:

- Se envuelve App con Provider, que acepta una propiedad store, store de Redux que proporciona acceso al estado global a los componentes descendientes de App.
- El componente App está dentro del Provider, lo que significa que todos los componentes dentro de App tendrán acceso al estado global almacenado en el store.

1. Crear componente que interactúa con el store

Por último, vamos a crear un componente llamado Header que va a interactuar con nuestro estado global. Para eso generamos dentro de components un archivo llamado Header.jsx:

```
components > Header.jsx > default

import {useSelector} from 'react-redux';

function Header() {
  const user = useSelector((state) => state.user)
  return (
    <header>
      <h1>Redux Toolkit Example</h1>
      <ul>
        <li>Name: {user.name}</li>
        <li>Email: {user.email}</li>
        <li>Username: {user.username}</li>
      </ul>
    </header>
  );
}

export default Header;
```

Se usa el hook `useSelector` de `react-redux` para acceder al estado del slice `user` en el store de REDUX. Luego, muestra el name, username y email del usuario en la interfaz de usuario.

1. Configurar petición a la API y mostrar datos

Por último, nos vamos al componente principal. `App.js`, de la aplicación. Al cargar la aplicación, realiza una solicitud a la API `JSONPlaceholder` para obtener datos de un usuario. Luego, despacha la acción `addUser` con los datos del usuario para actualizar el estado en Redux. Finalmente, renderiza `Header`, que muestra los datos del usuario obtenidos en la API.


```

5 App.js > ...
You, hace 1 segundo | 1 author (You)
import './App.css';
import Header from "../components/Header";
import { useEffect } from 'react';
import { useDispatch } from 'react-redux';
import { addUser } from "../redux/userSlice";
function App() {

  const dispatch = useDispatch();

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users/1")
      .then((response) => response.json())
      .then((data) => dispatch(addUser(data)))
  }, []);

  return (
    <>
    <Header/>
    </>
  );
}

You, hace 4 horas • Initialize project using Create R
export default App;

```

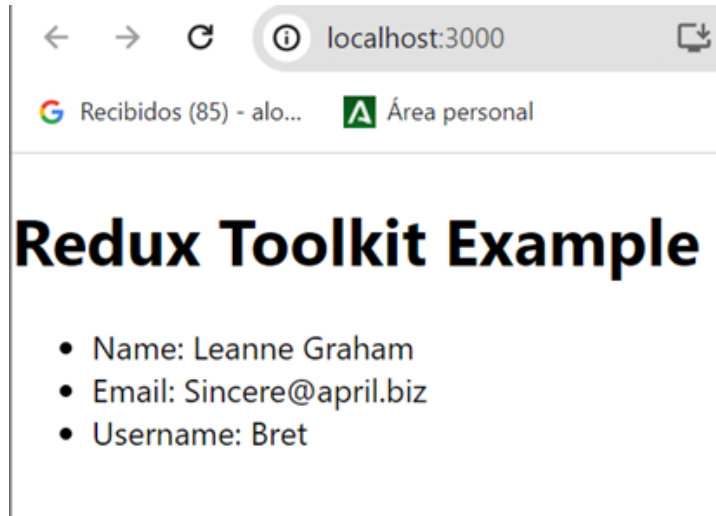
La dirección en la API nos lleva a los datos:

```

{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org"
}

```

Y Header se renderiza como:



Módulo 7: Consumo de APIs

7.1. Realización de solicitudes HTTP

¿Qué 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.

7.1.1. Uso de Fetch y Axios:

Para realizar solicitudes HTTP desde una aplicación React, se pueden utilizar dos enfoques principales: `Fetch` y `Axios`.

Uso de Fetch:

"Fetch" es una API nativa de JavaScript para realizar solicitudes HTTP. Puedes usarlo para hacer solicitudes y manejar las respuestas.

Ejemplo de uso de `Fetch`:

```
JavaScript
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Uso de Axios:

"Axios" es una biblioteca de JavaScript que simplifica el proceso de realizar solicitudes HTTP y manejar respuestas. Es una opción popular y más fácil de usar que `Fetch`.

Para usar Axios, primero debes instalarlo:

Comando instalación

```
Unset  
npm install axios
```

Ejemplo de uso de 'Axios':

```
JavaScript  
import axios from 'axios';  
  
axios.get('https://api.example.com/data')  
  .then(response => console.log(response.data))  
  .catch(error => console.error('Error:', error));
```

7.1.2. Manejo de datos asincrónicos:

¿Qué son datos asincrónicos?

En el contexto de JavaScript, las operaciones asincrónicas son aquellas que no bloquean la ejecución del código. Las solicitudes HTTP son un ejemplo común de operaciones asincrónicas.

Uso de Promesas:

Tanto `Fetch` como `Axios` devuelven promesas, lo que permite el manejo de datos asincrónicos. Las promesas son objetos que representan un valor que puede estar disponible ahora, en el futuro o nunca;

Async/Await:

Para simplificar aún más el manejo de operaciones asincrónicas, puedes usar `async/await`. Esto facilita la lectura y el manejo de promesas.

Ejemplo de uso de `async/await` con `Axios`:

```
JavaScript  
import axios from 'axios';
```

```
async function fetchData() {  
  try {  
    const response = await axios.get('https://api.example.com/data');  
    console.log(response.data);  
  } catch (error) {  
    console.error('Error:', error);  
  }  
}  
  
fetchData();
```

En este código, `async` indica que la función contiene operaciones asincrónicas, y `await` se utiliza para esperar a que la promesa se resuelva.

Estos son conceptos fundamentales al trabajar con APIs en React y realizar solicitudes HTTP, especialmente cuando se trata de manejar datos asincrónicos.

7.2. Integración de datos en componentes

La integración de datos en componentes se refiere a cómo 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 se refiere a cómo puede cambiar su estado y volver a renderizarse en respuesta a nuevos datos obtenidos de una fuente externa, como una API.

Cómo funciona:

1. **Solicitud de datos:** Se hace una solicitud a una API para obtener datos. Normalmente se hace en el método **componentDidMount** para los componentes de clase, o el hook **useState** para componentes funcionales.
2. **Establecer el estado:** Cuando la API devuelve los datos, se almacenan en el estado del componente usando el método **setState** para componentes de clase o en el método de actualización de estado devuelto por el hook **useState** para componentes funcionales.
3. **Renderizado:** React detecta que el estado del componente ha cambiado y vuelve a renderizar el componente. En el método de renderizado, se puede acceder a los

nuevos datos a través del estado del componente y usarlos para determinar qué se muestra en la interfaz de usuario.

4. **Actualizaciones:** Si en algún momento se hace otra solicitud a la API y se obtienen nuevos datos, se puede actualizar el estado del componente con esos nuevos datos. React volverá a renderizar el componente para reflejar los nuevos datos.

Hay que tener en cuenta que React solo va a renderizar el componente si los nuevos datos son diferentes a los datos antiguos. Esto significa que, si se hace una nueva solicitud API y se obtienen los mismos datos que antes, React no renderizará el componente.

7.2.2. Manipulación de respuestas JSON

La manipulación de respuestas JSON se refiere a cómo se obtienen los datos, en formato JSON devueltos por una API, y se transforman o manipulan para su uso en la aplicación.

Algunos conceptos clave:

1. **Parseo de JSON:** Cuando se recibe una respuesta de una API, los datos a menudo vienen como una cadena JSON. Para poder trabajar con esos datos, se necesita **parsearlos** en un objeto javascript. Esto se hace con la función **JSON.parse()** o, si se está usando la función **Fetch** para hacer la solicitud, se puede usar el método **.json()** en la respuesta.
2. **Acceso a los datos:** Cuando se hayan convertido los datos en un objeto javascript, se puede acceder a los datos usando la notación de puntos o corchetes. Por ejemplo, si hay un objeto con una propiedad **name**, se puede acceder a ella con **object.name** o **object['name']**.
3. **Transformación de los datos:** A veces, los datos vienen de la API sin el formato que se necesita para la aplicación. En este caso se podrían usar métodos de javascript como **.map()**, **.filter()**, **.reduce()**, etc. Para transformarlo en el formato que se necesite.
4. **Almacenamiento de los datos:** Una vez parseados y posiblemente transformados los datos, generalmente se almacenan en el estado del componente para que se puedan usar en el método de renderizado.