

FRONTEND:

Componentes en ReactJS

En esta unidad, nos sumergiremos en el corazón de ReactJS, explorando los aspectos fundamentales que dan vida a las aplicaciones: veremos la creación y composición de componentes, así como la diferencia entre componentes funcionales y de clase, comprendiendo cuándo y cómo utilizar cada uno. También aprenderemos a gestionar eventos como clics de ratón, cambios de estado y otros eventos del usuario, permitiendo la creación de experiencias interactivas y dinámicas. Y, por último, veremos cuáles son las fases clave del ciclo de vida, comprendiendo cuándo ocurren eventos como la inicialización, la actualización y la destrucción de un componente.

Tipos de componentes

Uno de los conceptos que tenemos que asimilar lo antes posible cuando estamos trabajando con aplicaciones complejas en React es que todo se basa en la creación de componentes para generar la interfaz visual de nuestras aplicaciones.

A partir de este concepto vamos a poder crear nuestros "propios elementos HTML" para poder resolver situaciones que el estándar de HTML no es capaz.

Habitualmente, encontraremos un componente raíz dentro de nuestras páginas y el resto de componentes anidados a partir de él, creando lo que podríamos denominar un Árbol de componentes. Es muy importante definir correctamente las relaciones entre estos componentes para poder generar flujos de intercambio de datos eficientes.

Una de las características que más se trabajan en las aplicaciones de React es que los componentes que generemos deberían funcionar totalmente aislados del contexto en el que se encuentran. De esta manera siempre seremos capaces de extraerlos para utilizarlos en diferentes ámbitos.

Existen dos tipos de componentes: **funcionales** y de **clase**.

1. Crear y estructurar un proyecto

Antes de comenzar a definir los tipos de componentes y las características que acompañan a cada uno de ellos, vamos a ver cómo podemos generar aplicaciones completas de React para evitar tener que partir de proyectos vacíos.

Para poder crear una aplicación podemos lanzar el siguiente comando:

```
npx create-react-app proycomponentes
```

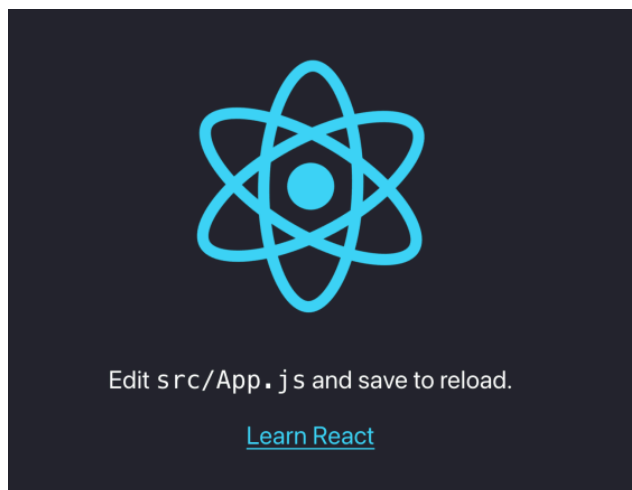
Mediante este comando invocamos el script `create-react-app` para generar una aplicación base de React. Aparte, instalamos las dependencias y dejamos el proyecto listo para arrancarlo.

📌 La librería `create-react-app` ha quedado discontinuada, pero nos sirve para poder generar aplicaciones de desarrollo mientras aprendemos.. Más adelante, usaremos el framework de react NextJS para iniciar los proyectos tal y como recomiendan en la documentación oficial para producción.

Podemos arrancarlo a partir del comando:

```
cd proycomponentes  
npm start
```

Si todo va correctamente tendremos que ver cómo se abre el navegador con la siguiente pantalla:



Estructura de un proyecto React

Vamos a analizar los ficheros más importantes dentro del proyecto que acabamos de generar:

public/index.html

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
<meta charset="utf-8" />
```

```

<link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<meta name="theme-color" content="#000000" />
<meta
name="description"
content="Web site created using create-react-app"
/>
<link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
<link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
<title>React App</title>
</head>
<body>
<noscript>You need to enable JavaScript to run this app.</noscript>
<div id="root"></div>
</body>
</html>

```

- Dentro de este fichero la parte más importante es la que define el `div` con id `root`. Este elemento es el que utilizaremos como punto de partida para poder generar nuestra aplicación de React.
- Nuestra aplicación se renderizará en un punto u otro de la página dependiendo de dónde coloquemos dicho `div`.

src/index.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

reportWebVitals();

```

- Es el segundo fichero más importante dentro de nuestra aplicación.
- La parte principal de este es la ejecución del método `render`, el cual toma el componente llamado `App` y lo renderiza dentro del `div` que hemos visto en el fichero anterior.

📌 A partir de esta estructura vamos a empezar a analizar y comprender el papel que juegan los componentes dentro de nuestras aplicaciones de React.

2. Componentes funcionales y de clase

2.1 Componentes funcionales (usados actualmente)

Este tipo de componentes se generan a través de una función javascript que retorna el código JSX que se va a renderizar siempre que utilicemos dicho componente.

Esta función recibe un único parámetro (`props`) con todos los valores de entrada que vamos a poder pasar al componente. De esta manera un mismo componente puede renderizar una vista diferente siempre y cuando modifiquemos los parámetros de entrada.

Podemos definir un componente a partir de la siguiente estructura:

src/components/Saludo.jsx

```
function Saludo(props) {  
  return <h3>Hola Qué tal</h3>;  
}  
export default Saludo;
```


- Como se observa en el ejemplo, se trata de una función sencilla que retorna un elemento JSX.
- Para poder utilizar este componente dentro de otros ficheros debemos exportarlo.

Cuando necesitemos utilizar este componente, simplemente tenemos que importarlo dentro del fichero donde vayamos a colocarlo y utilizar dicha importación para colocarlo dentro de una jerarquía de componentes superior:

src/App.js

```
import './App.css';  
import Saludo from './components/Saludo';  
function App() {  
  return (  
    <div className="App">  
      <Saludo />  
    </div>  
  );  
}  
export default App;
```

- Si importamos el nuevo componente como `Saludo` tendremos que colocar la etiqueta con el mismo nombre para poder renderizar el nuevo componente.
- Internamente cada vez que aparezca la etiqueta `<Saludo />` lo que estamos haciendo es ejecutar la función que hemos creado y que está retornando cierto JSX que se situará en el espacio donde hemos colocado el componente.

 Podemos comprobar en el inspector del navegador que no queda ni rastro del componente generado. Por contra tenemos el código HTML que devuelve.

Además, en comparación con otro tipo de frameworks para la generación de interfaces como Angular, el HTML que genera es bastante limpio.

2.2 Componentes de clase (en desuso)

De igual manera, podemos usar la sintaxis de clases definida dentro de Javascript a partir de su versión ES6 para poder definir componentes en React.

src/components/Despedida.jsx

```
import React from 'react';
class Despedida extends React.Component {
  render() {
    return <h2>Adios</h2>;
  }
}
export default Despedida;
```

- En este caso es importante la implementación del método `render` ya que será el encargado de devolver el código JSX que vamos a renderizar en cada uso del componente.
- Hacemos que la clase que estamos generando herede de `React.Component` para obtener todas las características propias de un componente de React.

📌 La tendencia es cada vez más grande hacia el uso de los componentes funcionales, pero no debemos perder de vista los componentes de clase ya que puede ser que nos tengamos que enfrentar a ellos en algún proyecto o desarrollo más antiguo..

3. Objetos

3.1 Objeto props

Cuando React detecta un componente personalizado se encarga de inyectar en él un objeto llamado **props** mediante el cual nos pasa todos los atributos asignados a nuestro componente.

Son los datos de entrada al componente mediante los cuales seremos capaces de establecer cómo se muestra el JSX que retornará cada componente.

Si el componente es funcional lo recibimos a través de los parámetros:

src/components/Saludo.jsx

```
function Saludo(props) {
  return <h3>Hola Qué tal, {props.nombre}</h3>;
}
export default Saludo;
```

src/App.js

```
<Saludo nombre="Rocío" />
<Saludo nombre="Iván" />
```

De esta manera tan sencilla seremos capaces de reutilizar el componente recibiendo diferentes parámetros de entrada.

Además, podemos usar el concepto de *object destructuring* de Javascript para especificar de una manera más concreta qué props son las que vamos a recibir dentro del componente:

src/components/Saludo.jsx

```
function Saludo({ nombre }) {
  return <h3>Hola qué tal, {nombre}</h3>;
}
export default Saludo;
```

Mediante el uso de este formato el código se vuelve mucho más legible. En caso de trabajar con componentes de clase recibimos las **props** como una propiedad de dicha clase. Podemos acceder a los datos que nos ofrece a través de `this.props`.

src/components/Despedida.jsx

```
import React from 'react';
class Despedida extends React.Component {
  render() {
    return <h2>Adios {this.props.nombre}</h2>;
  }
}
export default Despedida;
```

src/App.js

```
<Despedida nombre="Álvaro" />
```

El objeto props determina qué nos está enviando el componente padre. Se trata de una comunicación siempre hacia abajo en la jerarquía definida dentro del árbol de componentes.

En los ejemplos anteriores, el componente App actuaba como componente padre y se encargaba de renderizar el resto de componentes y de pasarles los datos necesarios para su correcto funcionamiento.

El objeto props es de solo lectura.

Ningún componente debe modificar los valores del objeto props. Esta es la definición de funciones puras. Es decir, aquel tipo de funciones que para los mismos datos de entrada siempre reciben los mismos datos de salida.

Si necesitamos guardar algún tipo de dato y que quede reflejado en el componente veremos más adelante que podemos hacerlo a través del `state`.

Podemos recuperar el código contenido entre las etiquetas de apertura y cierre de un componente a través de la propiedad `props.children`:

src/components/Texto.jsx

```
const Texto = ({ children }) => {  
  return <p>Este es el mensaje: {children}</p>  
}  
export default Texto;
```

src/App.js

```
<Texto>Mensaje importante</Texto>
```

📌 En este último ejemplo hemos usado la nomenclatura de arrow function para definir el componente `Texto`. Tendría el mismo efecto que en los componentes anteriores, simplemente es un cambio de sintaxis.

3.2 Objeto state

Se trata del espacio donde vamos a poder almacenar los datos locales del componente, es decir, todos aquellos valores necesarios dentro de un componente y mediante los cuales vamos a conformar el JSX que se va a renderizar en su uso.

Debe ser un objeto sencillo para evitarnos así errores inesperados.

Este objeto, como veremos a continuación, sólo podemos usarlo dentro de un componente de clase. Para lograr el mismo concepto dentro de un componente funcional disponemos de una relativamente nueva herramienta llamada `hook`.

Para poder utilizar el objeto `state` debemos iniciarlo dentro del constructor de clase. Además puede ser el momento seleccionado para establecer los valores iniciales de cada uno de los datos que vayamos a almacenar dentro de dicho `state`.

Un ejemplo sencillo de uso podría ser el siguiente:

src/components/ProfileCard.jsx

```
import React from 'react';  
class ProfileCard extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      titulo: 'Título de la card',  
      texto: 'Texto de la card'  
    };  
  }  
}
```

```

} render() {
return <div className="profile-card">
<h1>{this.state.titulo}</h1>
<p>{this.state.texto}</p>
</div>;
}
}
export default ProfileCard;

```

En este ejemplo, podemos ver un par de cosas:

- Si utilizamos el constructor dentro de un componente de clase estamos obligados a llamar al método `super` y pasarle como parámetro el objeto `props`.
- Dentro del constructor podemos inicializar el objeto `state` definiendo las propiedades que necesitemos con los valores que vayamos a utilizar.

Los componentes reaccionan a los posibles cambios que se realicen en los objetos `props` y/o `state`.

Siempre que necesitemos modificar alguna de las propiedades del objeto `state`, debemos hacerlo a través del método `setState` y nunca hacerlo directamente. De esta manera nos aseguramos que el componente sea consciente de los cambios y vuelva a renderizarse. Si modificamos una de las propiedades de dicho objeto no se sobrescribe el resto.

Vamos a ver este comportamiento con un ejemplo. El siguiente componente es un cronómetro que realiza una simple cuenta atrás.

src/components/Cronometro.jsx

```

import React from 'react';
class Cronometro extends React.Component {
constructor(props) {
super(props);
this.state = {
contador: this.props.inicio
};
let interval = setInterval(() => {
this.setState({ contador: this.state.contador - 1 })
if (this.state.contador === 0) {
clearInterval(interval);
}
}, 1000);
}
render() {
return <h2>{this.state.contador}</h2>
}
}
export default Cronometro;

```


src/App.js

```
<Cronometro inicio="10" />
```

- Dentro de este componente estamos utilizando el método `setInterval` para ejecutar cierta función cada 100 milisegundos.
- Dentro del intervalo modificamos a través del método `setState` la propiedad `contador` definida dentro del estado del componente. Previamente habíamos inicializado dicha propiedad a través del valor recibido en las props.
- Esta ejecución del método `setState` en este componente fuerza de nuevo la ejecución del método `render`, por lo que, a partir del nuevo valor para el contador, se modificará la instancia virtual del DOM y posteriormente, en el proceso de reconciliación se modificará de la misma manera el DOM que estamos visualizando en el navegador.

🔗 La acción de renderizado y reconciliación son transparentes para el desarrollador. Ya analizaremos algunos casos en los que sí que es importante que las tengamos en cuenta.

Las actualizaciones del estado de un componente pueden ser acciones asíncronas. Para poder salvar este problema podemos modificar el estado a través de la siguiente sintaxis.

```
this.setState((state, props) => {  
  return { contador: state.contador - 1 }  
})
```

Pasamos por parámetro una función que recibirá el valor actual de los objetos `state` y `props` en caso de estar encuadrados dentro de alguna función asíncrona.

4. useState hook

Desde la versión 16.8 de React podemos usar una nueva herramienta, los hooks. A través de estos hooks podemos conseguir realizar acciones parecidas a las que hasta ahora veníamos haciendo desde los componentes de clase pero en los componentes funcionales.

Mediante el hook `useState` podemos manejar el estado dentro de los componentes funcionales.

Lo vemos a través de este ejemplo:

src/components/Animales.jsx

```
import { useState } from "react";  
const Animales = () => {  
  const stateArr = useState({  
    animales: [  
      { nombre: 'Bobby', tipo: 'perro', edad: 12 },  
      { nombre: 'Flipper', tipo: 'pajaro', edad: 5 },  
      { nombre: 'Gary', tipo: 'gato', edad: 16 }  
    ]  
  })  
}
```

```

],
mensaje: 'Este es el mensaje'
})
return <div>
<h1>{stateArr[0].animales[0].nombre}</h1>
</div>
}
export default Animales;

```

- Debemos importar `useState` dentro de nuestro componente.
- Posteriormente, llamamos al hook `useState` y dentro definimos el objeto que queremos incluir dentro del estado de nuestro componente. Podemos guardar cualquier tipo de objeto.
- En este ejemplo, estamos almacenando en el estado un array bajo la clave `animales` y una cadena de caracteres bajo la clave `mensaje`.
- Para recuperar los valores del estado disponemos del objeto `stateArr`, el cual es de tipo array.
 - En la primera posición del array tenemos el objeto almacenado dentro del estado del componente.
 - En la segunda posición del array disponemos de un método para poder modificar dicho estado.
- Como podemos observar, la sintaxis para recuperar alguno de los valores del array es bastante incómoda. Es por ello que vamos a ver cómo podemos mejorarlo, contando que `stateArr` es un array.

src/components/Animales.jsx

```

import { useState } from "react";
const Animales = () => {
  const [animalsState, setAnimalsState] = useState({
    animales: [
      { nombre: 'Bobby', tipo: 'perro', edad: 12 },
      { nombre: 'Flipper', tipo: 'pajaro', edad: 5 },
      { nombre: 'Gary', tipo: 'gato', edad: 16 }
    ],
    mensaje: 'Este es el mensaje'
  })
  return <div>
  <h1>{animalsState.animales[0].nombre}</h1>
  </div>
  }
  export default Animales;

```

En este ejemplo, desgranamos el objeto que devuelve el hook en cada uno de los elementos que lo componen (el objeto con el estado y el método para modificarlo). De todas maneras, la sintaxis sigue siendo un poco incómoda. Aprovechando que podemos llamar al método `useState` tantas veces como queramos, podemos reorganizarlo de la siguiente manera:

src/components/Animales.jsx

```
import { useState } from "react";
const Animales = () => {
  const [animalsState, setAnimalsState] = useState([
    { nombre: 'Bobby', tipo: 'perro', edad: 12 },
    { nombre: 'Flipper', tipo: 'pajaro', edad: 5 },
    { nombre: 'Gary', tipo: 'gato', edad: 16 }
  ])
  const [mensaje, setMensaje] = useState('Este es el mensaje');
  return <div>
    <h1>{animalsState[0].nombre}</h1>
    <h2>{mensaje}</h2>
  </div>
}
export default Animales;
```

Separamos los elementos del estado en llamadas independientes del hook `useState`. De esta manera, cada elemento tiene su propio objeto representándolo y su propio método para poder modificar su valor.

De haberlos mantenido unidos, en el momento de actualizar uno de los dos, se hubiera machacado el valor del otro. Separando funcionalidades nos olvidamos de ese problema.

El uso de los métodos incluidos con cada llamada del hook `useState` fuerzan un cambio en el estado correspondiente y como pasaba con `setState` para los componentes de clase también fuerzan el renderizado del componente.

Eventos

Con lo aprendido en las clases anteriores ya somos capaces de generar interfaces de usuario simples con React. Por el momento son interfaces estáticas mediante las cuales podemos mostrar ciertos datos.

Dentro de nuestro aprendizaje uno de los hitos importantes es el momento en el que, dentro de unas clases, veamos cómo podemos generar nuestro código JSX incluyendo condicionales y bucles, ya que nos dará la posibilidad de dejar de repetir muchos bloques de código.

Otro de los puntos importantes es el que vamos a afrontar en esta clase, el manejo de eventos. Es la forma más básica que tenemos para interactuar con el usuario y en React se ha solucionado de una manera muy eficiente siguiendo las librerías nativas de Javascript.

1. Manejo de eventos

Para manejar cualquiera de los eventos Javascript que tenemos disponibles debemos enlazarlos con una función que estará definida dentro del componente donde estemos interesados en recoger dicho evento.

📌 Antes de probar los distintos bloques de código que te vamos a plantear en esta clase, sería recomendable generar un nuevo proyecto de React para así trabajar de manera más cómoda: `npx create-react-app eventos`

src/components/BotonSimple.jsx

```
const BotonSimple = () => {  
  const handleClick = (event) => {  
    console.log('Botón pulsado');  
  }  
  return (  
    <div>  
      <button onClick={handleClick}>Pulsa el botón</button>  
    </div>  
  );  
}
```

- Cuando asignamos la función al evento sobre el botón hay que tener cuidado de no incorporar los paréntesis para evitar así su ejecución
- La librería de React, sobre el evento nativo Javascript, crea una clase superior para así poder controlar las posibles incompatibilidades cuando nos cambiamos de entorno.
- En el siguiente enlace podemos encontrar todos los eventos disponibles.

<https://legacy.reactjs.org/docs/events.html#supported-events>

Si necesitamos realizar la misma acción desde un componente de clase, podemos hacerlo de la misma manera:

src/components/BotonSimpleClase.jsx

```
import { Component } from "react";  
class BotonSimpleClase extends Component {  
  handleClick = (event) => {  
    console.log('Botón pulsado');  
  }  
  render() {  
    return (  
      <div>  
        <button onClick={this.handleClick}>Pulsa el botón</button>  
      </div>  
    );  
  }  
}
```

```
}  
export default BotonSimpleClase;
```

El parámetro que recibimos dentro del manejador del evento que estamos gestionando es el evento sintético que comentábamos antes con todas las propiedades necesarias para gestionar la acción correspondiente. Esto nos abstrae de las dependencias de cada uno de los navegadores.

En los casos anteriores, el único parámetro que reciben las funciones definidas para manejar los eventos únicamente reciben el propio evento. Si queremos pasarle algún parámetro personalizado podemos hacerlo de esta manera.

src/components/EligeBoton.jsx


```
import { useState } from "react";  
const EligeBoton = () => {  
  const [mensaje, setMensaje] = useState("");  
  const handleClick = (num) => {  
    setMensaje(`Has elegido el botón ${num}`);  
  }  
  return <div>  
    <h2>Elige un botón</h2>  
    <p>{mensaje}</p>  
    <button onClick={(event) => handleClick(1)}>Botón 1</button>  
    <button onClick={(event) => handleClick(2)}>Botón 2</button>  
    <button onClick={(event) => handleClick(3)}>Botón 3</button>  
  </div>  
}
```

Podemos observar cómo en el elemento JSX, entre las llaves ya no ponemos el nombre de la función si no la definición misma. De esta manera, dentro de dicha función manejadora podemos ejecutar todas aquellas sentencias que necesitemos.

Además hemos aprovechado este ejemplo para usar el hook `useState` y modificar un mensaje donde concretamos qué botón es el que se ha seleccionado. En cada pulsación del botón deberíamos ver el cambio en el mensaje.

1.1 Componente Suma

Vamos a plantear un componente que integre todo lo visto anteriormente de una manera sencilla. El componente a generar debe recibir a través de las props dos números enteros y cuando pulsemos un botón debe mostrar el resultado de la suma de dichos números enteros.

 Un poco más adelante tienes la solución del planteamiento que hemos hecho. Te animo a que intentes desarrollar el componente antes de mirar más abajo.

src/components/Suma.jsx

```
import { useState } from "react";
const Suma = ({ numA, numB }) => {
  const [resultado, setResultado] = useState(0);
  const handleClick = () => {
    setResultado(numA + numB);
  }
  return <div>
    <button onClick={handleClick}>Calcula el resultado</button>
    <p>El resultado de la suma de {numA} y {numB} es {resultado}</p>
  </div>
}
export default Suma;
```

Podemos usar el componente de la siguiente manera:

src/App.js

```
<Suma numA={3} numB={4} />
<Suma numA={51} numB={93} />
```

Pasamos los parámetros numéricos entre llaves para asegurarnos que son números enteros en vez de cadenas de caracteres.

1.2 Otros componentes

Como ya hemos visto en los casos anteriores disponemos de muchos eventos para poder capturar la interacción del usuario dentro de nuestros componentes.

No todos los objetos que coloquemos dentro de nuestra interfaz gráfica tienen capacidad para poder recuperar eventos, pero sí hay multitud de ellos que podemos usar de la misma manera que hemos hecho con los botones anteriores para realizar las acciones que necesitemos en cada caso.

Vamos a realizar un ejemplo en el que capturaremos diferentes eventos de elementos, sobre todo relacionados con campos de texto y formularios:

src/components/Eventos.jsx

```
import { useState } from "react";
const Eventos = () => {
  const [text, setText] = useState("");
  const [rango, setRango] = useState(0);
  const [mensaje, setMensaje] = useState("");
  const [coords, setCoords] = useState({ x: 0, y: 0 })
  const onChangeInput = (event) => {
    setText(event.target.value);
  }
}
```

```

const onChangeRange = (event) => {
  setRango(event.target.value);
}
const handleOnMouseEnter = (event) => {
  setMensaje('Ha entrado el ratón')
}
const handleOnMouseMove = (event) => {
  setCoords({
    x: event.clientX,
    y: event.clientY
  })
}
return <div>
  <div className="campo-texto">
    <p>El valor del campo de texto es: {text}</p>
    <input type="text" onInput={onChangeInput} />
  </div>
  <div className="rango">
    <p>El valor del rango es: {rango}</p>
    <input type="range" min="1" max="100" step="1" onChange={onChangeRange} />
  </div>
  <div style={{
    width: '200px',
    height: '200px',
    backgroundColor: 'green'
  }}
  onMouseEnter={handleOnMouseEnter}
  onMouseMove={handleOnMouseMove}
  >
    <p>{mensaje}</p>
    <p>Las coords son X: {coords.x} Y: {coords.y}</p>
  </div>
</div>
}
export default Eventos;

```

- El primer bloque consta de un párrafo y un campo de texto. Estamos capturando el evento `onInput`, el cual se lanza cuando escribimos algo dentro del campo de texto.
- En ese caso lanzamos la función `onChangeInput`, dentro de la cual modificamos el valor de la propiedad `text`. El valor para esta propiedad lo tomamos a partir del `event`.
- En el segundo bloque estamos haciendo lo mismo pero con un campo de tipo `range`. En este caso capturamos el evento `onChange` y recuperamos el valor también a través del propio evento.
- Por último, el bloque final es un cuadrado sobre el cual estamos capturando los eventos `onMouseEnter` y `onMouseMove` para saber cuándo entra el ratón dentro del cuadrado y cuándo se mueve por dentro.

2. Comunicación hijo-padre

A través de las props hemos conseguido pasar información desde el componente padre hacia el componente hijo. Son los atributos que definen cómo van a funcionar dichos componentes y además nos permiten diferenciarlos de otros usos del mismo componente.

Existe la posibilidad de que necesitemos pasar cierta información desde el componente hijo y que suba hacia arriba en la jerarquía hacia el componente padre. Mediante la propiedad props podemos delegar la ejecución de cualquier función definida dentro del componente padre para que se ejecute con cualquier acción dentro del componente hijo.

De la misma manera que pasamos números, cadenas, arrays... a través de las props, podemos incluir la definición de una función para ejecutarla dentro del hijo.

Vamos a hacer una modificación en el componente Sumar para que informe del resultado al padre en el momento de calcularlo:

src/components/Suma.jsx

```
import { useState } from "react";
const Suma = ({ numA, numB, sumaResuelta }) => {
  const [resultado, setResultado] = useState(0);
  const handleClick = () => {
    const resultado = numA + numB;
    setResultado(resultado);
    sumaResuelta(resultado);
  }
  return <div>
    <button onClick={handleClick}>Calcula el resultado</button>
    <p>El resultado de la suma de {numA} y {numB} es {resultado}</p>
  </div>
}
export default Suma;
```

- Como podemos observar, recibimos a través de las props un nuevo parámetro llamado sumaResuelta. Desde el padre, posteriormente, pasaremos dentro de ese atributo una función.
- En el cálculo de la suma, ejecutamos la función recibida dentro de las props. Además en esa ejecución pasamos por parámetro el resultado.

Cuando utilizamos el componente dentro del componente superior, aparte de las propiedades numA y numB tenemos que pasarle también la definición de la función sumaResuelta.

src/App.js


```
import './App.css';
import Suma from './components/Suma';
function App() {
  const handleSumaResuelta = (resultado) => {
    console.log(` Se ha resuelto la suma con resultado: ${resultado} `);
  }
  return (
    <div className="App">
      <Suma numA={3} numB={4} sumaResuelta={handleSumaResuelta} />
      <Suma numA={51} numB={93} sumaResuelta={handleSumaResuelta} />
    </div>
  );
}
export default App;
```

Definimos la función `handleSumaResuelta` dentro del componente padre y pasamos su definición al componente hijo a través de la propiedad `sumaResuelta`. Resumiendo, es como si la función estuviese definida en un fichero y se estuviese ejecutando en otro diferente.

Podemos realizar la ejecución de estas funciones recibidas desde el padre directamente en la definición del código JSX que vamos a renderizar. Lo vemos a través del siguiente ejemplo. Un componente con dos botones, el primero de ellos lleva a cabo una cuenta hacia delante de un valor, el segundo avisa al componente padre del resultado de la cuenta:

src/components/Contador.jsx

```
import { useState } from "react";
const Contador = ({ informaResultado }) => {
  const [numero, setNumero] = useState(0);
  const handleClick = () => {
    setNumero(numero + 1);
  }
  return <div>
    <p>{numero}</p>
    <button onClick={handleClick}>Sube contador</button>
    <button onClick={(event) => informaResultado(numero)}>Informa Resultado</button>
  </div>
}
export default Contador;
```

En el evento click del segundo botón lanzamos directamente la función recibida del padre.

Dentro del componente padre pasamos la función a través de los atributos del hijo:

src/App.js

```
import './App.css';
```

```
import Contador from './components/Contador';
function App() {
  const handleInformaResultado = (resultado) => {
    console.log(` El contador tiene un valor de ${resultado} `);
  }
  return (
    <div className="App">
      <Contador informaResultado={handleInformaResultado} />
    </div>
  );
}
export default App;
```

3. Two-way Binding

Mediante el uso del evento `onChange` sobre un campo de texto podemos recuperar, como ya hemos visto anteriormente, el valor del mismo y asignárselo a una variable. Si, además, colocamos como valor del campo de texto dicha variable, conseguimos el efecto Two-way binding.

Lo que conseguimos con esto es que cuando modifiquemos el campo de texto se modificará la variable y viceversa.

Lo vemos a partir de un pequeño ejemplo:

src/components/FormularioSimple.jsx

```
import { useState } from 'react';
const FormularioSimple = () => {
  const [valor, setValor] = useState("");
  const handleChange = (event) => {
    setValor(event.target.value);
  }
  const handleClick = () => {
    setValor("Valor después del click");
  }
  return <div>
    <p>{valor}</p>
    <input type="text" value={valor} onChange={handleChange} />
    <button onClick={handleClick}>Cambia valor</button>
  </div>;
}
export default FormularioSimple;
```

A partir de este componente, como hemos enlazado la propiedad `valor` con el campo de texto, cada vez que se modifica, observamos cómo cambia en todos los espacios donde la hemos colocado, tanto si tocamos el botón como si cambiamos el campo de texto.

4. Creación de formulario

En el siguiente ejemplo vamos a crear un formulario con 3 campos donde recoger los datos de nombre, apellidos y edad. Dichos valores se deben mostrar en párrafos encima de los campos de texto.

Cuando se pulse el botón del formulario, debe hacer llegar los datos del mismo al componente padre.

📌 Como ya hemos propuesto en otras ocasiones, antes de ver la solución del ejercicio, plantéate cómo poder resolverlo y si no encuentras la solución, vuelve aquí para ver posibles errores o para concretar tu planteamiento

src/components/Formulario.jsx

```
import { useState } from "react";
const Formulario = ({ enviaData }) => {
  const [data, setData] = useState({
    nombre: "",
    apellidos: "",
    edad: ""
  });
  const handleChange = (event, field) => {
    setData({
      ...data,
      [field]: event.target.value
    });
  }
  const handleClick = () => {
    enviaData(data);
  }
  return <div>
    <div>
      <p>{data.nombre}</p>
      <p>{data.apellidos}</p>
      <p>{data.edad}</p>
    </div>
    <div>
      <label>Nombre</label>
      <input type="text" onChange={event => handleChange(event, 'nombre')} />
    </div>
    <div>
      <label>Apellidos</label>
      <input type="text" onChange={event => handleChange(event, 'apellidos')} />
    </div>
    <div>
      <label>Edad</label>
      <input type="text" onChange={event => handleChange(event, 'edad')} />
    </div>
  </div>
```

```
<button onClick={handleClick}>Enviar</button>
</div>;
}
export default Formulario;
```

Se crea una única función para poder recuperar los valores de los campos de texto y así no tener que repetir mucho código. El componente padre podría recuperar los valores como sigue:

src/App.js

```
import './App.css';
import Formulario from './components/Formulario';
function App() {
  const handleEnviaData = (data) => {
    console.log(data);
  }
  return (
    <div className="App">
      <Formulario enviaData={handleEnviaData} />
    </div>
  );
}
export default App;
```

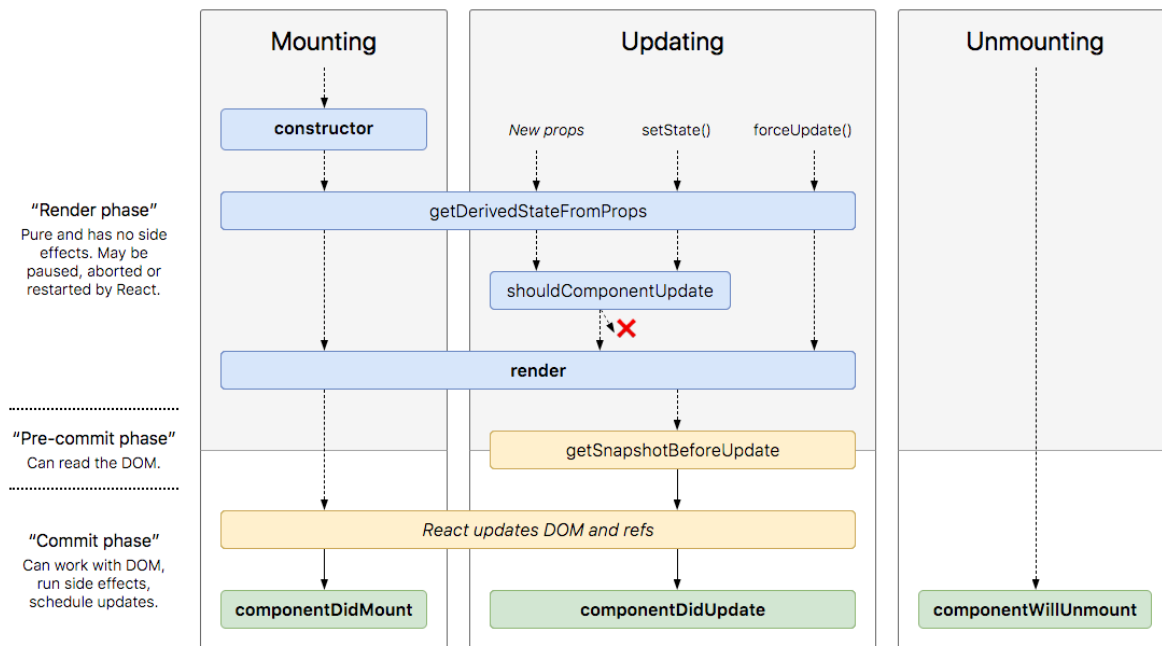
[#03 Curso de React Hooks \[Formularios onChange & onSubmit \]](#)

Ciclo de vida

Después de analizar los aspectos básicos de los componentes y las posibilidades que tenemos de interacción por parte de los usuarios, en esta clase vamos a cerrar con uno de los puntos importantes que tienen los componentes de clase, [el ciclo de vida](#) y vamos a llevar a cabo un ejemplo completo para ver cómo podemos integrar varios componentes dentro de una aplicación y que se comuniquen entre sí.

1. Etapas del ciclo de vida

Cada componente dispone de numerosos métodos dentro de su ciclo de vida que podemos sobrescribir para ejecutar código en momentos particulares del proceso. En esta imagen podemos observar un detalle de cada uno de ellos y el orden que ocupan dentro de la ejecución del componente.



Vamos a definir las diferentes etapas y a especificar cuándo y cómo podemos usar cada uno de los métodos implicados.

📌 Cabe recordar que estos métodos están únicamente disponibles para componentes de clase. Si vamos a desarrollar componentes funcionales debemos trabajar con los nuevos Hooks.

1.1 Mounting

Esta etapa discurre cuanto el componente se está creando y está siendo insertado en el DOM.

1 - constructor

- Este método se ejecuta antes de montar el componente.
- Si implementamos el constructor es para generar algún tipo de estado dentro del mismo y luego tenerlo accesible dentro de la ejecución del componente.
- Lo primero que debemos hacer es enlazar las props mediante la ejecución del método `super` ya que si no, perdemos el acceso a dichas propiedades.
- No debemos llamar a `setState` dentro del constructor. El componente no está todavía generado y no tiene sentido forzar su renderizado.
- Si el constructor debe dar valor al estado del componente puede hacerlo inicializando directamente el objeto `this.state`.

2 - static `getDerivedStateFromProps(props, state)`

- Se lanza justo antes de invocar el renderizado. Se ejecuta tanto en la fase de montaje como en la fase de actualización que veremos más adelante.

- Debería devolver un objeto para actualizar el estado o null si no queremos actualizar nada.
- No dispone de acceso a `this`, la instancia del componente.
- Lo podemos utilizar si queremos sincronizar el estado del componente en caso de que se haya producido algún cambio de última hora en las props.
- Para provocar cambios más grandes a partir de la descarga de datos o parecidos tenemos otro tipo de métodos que son de más ayuda.
- Hay que tener cuidado porque se lanza en cada renderizado, independientemente de la causa del mismo.

3 - render()

- Es el único método requerido dentro de la definición de una clase.
- Se encarga de generar el html necesario a partir del JSX y el uso de state y props.
- El método render debería ser puro, es decir, no debería hacer modificaciones del estado del componente.
- No interactúa directamente con el navegador.

4 - componentDidMount()

- Se lanza justo después de que el componente se haya insertado en la jerarquía de componentes de nuestra aplicación.
- Es el sitio donde podemos lanzar las peticiones para recuperar datos externos.
- Se puede ejecutar `setState` inmediatamente cuando carga el método y eso implica que, aunque el render se lanza una vez extra, el usuario no visualiza ningún estado intermedio.
- Debemos llevar este patrón con cuidado.

Vemos en un ejemplo la implementación de los diferentes métodos y dentro de la consola se puede apreciar el orden de ejecución.

src/components/CicloVida.jsx

```
import { Component } from "react";
class CicloVida extends Component {
  // inicializamos las props
  constructor(props) {
    super(props);
    this.state = { mensaje: 'Mensaje inicial' }
    console.log('[CicloVida] constructor');
  }
  // Dentro de este método debemos devolver el nuevo estado
  static getDerivedStateFromProps(props, state) {
    console.log('[CicloVida] getDerivedStateFromProps', props, state);
    return state;
  }
  // Lanzamos tareas de conexión a servicios externos
  componentDidMount() {
    console.log('[CicloVida] componentDidMount');
```

```

}
render() {
  console.log('[CicloVida] render');
  return (
    <div className="CicloVida">
      Test
    </div>);
}
}
export default CicloVida;

```

1.2 Updating

La actualización de un componente está provocada por un cambio en las props o en el state. Estos cambios provocan que se lancen otra serie de métodos dentro del propio ciclo de vida. Su orden y las acciones que llevan a cabo son las siguientes:

1 - `static getDerivedStateFromProps (props, state)`

- Tiene el mismo efecto que en el ciclo anterior

2 - `shouldComponentUpdate(nextProps, nextState)`

- Mediante este método le indicamos a React si debemos renderizar el componente cuando se modifiquen las props o el state.
- Por defecto el resultado que retorna es true.
- No se lanza en el primer render ni cuando forzamos la actualización del componente a través del método `forceUpdate`.
- Es un método que debemos usar para controlar el renderizado en caso de que necesitemos mejorar el rendimiento del componente. Podríamos llegar a evitar renderizados inútiles.

3 - `render`

- Lo mismo que en el ciclo anterior

4 - `getSnapshotBeforeUpdate(prevProps, prevState)`

- Se ejecuta antes de que el renderizado que se ha realizado en el método anterior se compare con el DOM y se actualice.
- Nos permite recuperar cualquier tipo de información del DOM para posteriormente pasarla al método `componentDidUpdate`.
- Podemos acceder a los elementos del DOM a través de refs (`React.createRef` o `useRef`).
- Debemos retornar un valor que será pasado como tercer parámetro al método `componentDidUpdate`.

5 - `componentDidUpdate(prevProps, prevState, snapshot)`

- Se lanza justo después de la actualización

- Podemos realizar modificaciones en el DOM o llamadas a servicios externos siempre y cuando basemos estas decisiones en cambios sucedidos sobre las props.
- Si no hacemos esto, podemos entrar en bucles infinitos al estar lanzando peticiones indiscriminadamente en cada actualización

Vemos el orden de ejecución en un ejemplo:

src/components/CicloVida.jsx

```
import { Component } from "react";
class CicloVida extends Component {
  // inicializamos las props
  constructor(props) {
    super(props);
    this.state = { mensaje: 'Mensaje inicial' }
    console.log('[CicloVida] constructor');
  }
  // Dentro de este método debemos devolver el nuevo estado
  static getDerivedStateFromProps(props, state) {
    console.log('[CicloVida] getDerivedStateFromProps', props, state);
    return state;
  }
  shouldComponentUpdate(nextProps, nextState) {
    console.log('[CicloVida] shouldComponentUpdate');
    // Devolvemos true o false si seguimos renderizando o no
    return true
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    console.log('[CicloVida] getSnapshotBeforeUpdate');
    // dentro de este método podemos guardar cierta información que posteriormente le
    // llegará al método componentDidUpdate como tercer parámetro
    return { contador: '321' };
  }
  componentDidUpdate(prevProps, prevState, snapshot) {
    console.log(snapshot);
    console.log('[CicloVida] componentDidUpdate');
  }

  handleClick = () => {
    this.setState({ mensaje: new Date().toString() });
  }
  render() {
    console.log('[CicloVida] render');
    return (
      <div className="CicloVida">
        <p>{this.state.mensaje}</p>
        <button onClick={this.handleClick}>Pulsa</button>
      </div>);
  }
}
```



```
}  
}  
export default CicloVida;
```

2. Proyecto de lista de tareas simple

Vamos a desarrollar un proyecto completo utilizando todos los conceptos que hemos aprendido anteriormente. Como siempre, te animamos a que intentes completar por ti mismo el proyecto propuesto y si tienes algún problema recurras a la solución que vamos a explicar detalladamente más adelante.

El concepto de la lista de tareas es un proyecto en el que vamos a permitir al usuario crear tareas y vamos a intentar que se visualicen en una lista. Para ello, vamos a dividir el proyecto en los siguientes componentes:

Formulario

Dentro de este componente tenemos todos los campos para la generación de la tarea. En este caso, vamos a permitir un texto y una prioridad, a elegir entre (alta, media y baja). Cuando se pulse el botón para crear la tarea, los datos generados tendrán que viajar hacia el componente padre.

App

Será el componente superior de la jerarquía y donde colocaremos los otros dos componentes. Maneja el estado de la aplicación y en este caso será el encargado de recibir las tareas del Formulario e introducirlas dentro de un array. Dicho array tendremos que pasarlo a través de las props para el siguiente componente.

Lista

Este componente recibe el array del componente superior (App) y se encarga de mostrarlas en una interfaz. Como todavía no hemos hablado de bucles ni condicionales, esto se puede resolver desarrollando un método que se encargue de, para cada tarea, meter dentro de un array el código JSX que la represente. Si posteriormente visualizamos el array, deberíamos poder renderizar todas las tareas.

🔗 A partir de aquí empezamos con la resolución de los diferentes componentes, así que te animamos a que lo intentes antes de ver los siguientes spoilers 😊

2.1 Creación del proyecto

Antes de ponernos a programar componentes procedemos a la creación del proyecto a partir del siguiente comando:

```
npx create-react-app todosimple
```

2.1 Componente Formulario

Vamos a empezar programando la funcionalidad del formulario. A través de los eventos de los campos de texto trataremos de recuperar la información, guardarla dentro del estado del componente y posteriormente enviarla hacia el padre a través de un método definido en el mismo.

src/components/Formulario.jsx

```
import { useState } from "react";
const Formulario = ({ tareaEnviada }) => {
  const [data, setData] = useState({
    texto: '',
    prioridad: 'baja'
  });
  const handleChange = (event, field) => {
    setData({
      ...data,
      [field]: event.target.value
    });
  }
  const handleSubmit = (event) => {
    event.preventDefault();
    tareaEnviada(data);
  }
  return <div className="formulario">
    <form onSubmit={handleSubmit}>
      <div>
        <label className="form-label">Texto</label>
        <input type="text" className="form-control" onChange={(event) => handleCha
          nge(event, 'texto')} />
      </div>
      <div>
        <label className="form-label">Prioridad</label>
        <select className="form-control" onChange={(event) => handleChange(event,
          'prioridad')}>
          <option value="baja">Baja</option>
          <option value="media">Media</option>
          <option value="alta">Alta</option>
        </select>
      </div>
      <button>Enviar</button>
    </form>
  </div>
}

export default Formulario;
```

- En este primer componente, lo primero que nos debería llamar la atención es el uso de la etiqueta `form` y a partir de ahí nos aprovechamos del evento `onSubmit`.
- A partir de ahí, cada uno de los campos de formulario envían la información al estado a partir del evento `onChange`.

2.2 Componente App

Este componente va a actuar como componente contenedor. Dentro del mismo vamos a almacenar todas las tareas que se vayan generando y así podremos pasárselas a la lista una vez la tengamos creada.

Lo primero que vamos a hacer es representar el Formulario y recibir los datos del mismo:

src/App.js

```
import { useState } from 'react';
import './App.css';
import Formulario from './components/Formulario';
function App() {
  const [tareas, setTareas] = useState([]);
  const handleTareaEnviada = (tarea) => {
    setTareas([
      ...tareas,
      tarea
    ]);
  }
  return (
    <div className="App">
      <p>{tareas.length}</p>
      <Formulario tareaEnviada={handleTareaEnviada} />
    </div>
  );
}
export default App;
```

- Las tareas llegan a través del método `handleTareaEnviada` y modificamos el estado del componente para incorporarlas dentro de un array.
- Hemos colocado dentro del JSX un párrafo para ver cómo se incrementan las tareas según las vamos insertando.

El último paso es la generación del componente Lista mediante el cual vamos a mostrar todas estas tareas:

src/components/Lista.jsx

```
const Lista = ({ arrTareas }) => {
  const pintarTareas = () => {
```

```

const lis = arrTareas.map(tarea => <li>{tarea.texto} - {tarea.prioridad}</li>);

return lis;
}
return <ul>
{pintarTareas()}
</ul>;
}
export default Lista;

```

- El método `pintarTareas` se encarga de generar un array donde colocamos el código JSX perteneciente a cada una de las tareas.
- Posteriormente ejecutamos este método dentro del JSX que retorna el componente.
- En las próximas clases analizaremos técnicas más elegantes para poder llevar a cabo este tipo de acciones.

Nos quedaría representar este componente dentro de **App.js**

src/App.js

```

import { useState } from 'react';
import './App.css';
import Formulario from './components/Formulario';
import Lista from './components/Lista';
function App() {
  const [tareas, setTareas] = useState([]);
  const handleTareaEnviada = (tarea) => {
    setTareas([
      ...tareas,
      tarea
    ]);
  }
  return (
    <div className="App">
      <p>{tareas.length}</p>
      <Formulario tareaEnviada={handleTareaEnviada} />
      <Lista arrTareas={tareas} />
    </div>
  );
}
export default App;

```

No es el proyecto más elegante que podíamos desarrollar pero si es un buen primer contacto para empezar a entender cómo se relacionan los componentes en sí y un punto de partida para todos los conceptos que tenemos por delante.

[#04 Curso de React Hooks \[React Hook Form \]](#)

[#06 Curso de React Hooks \[Props en Componentes \]](#)

EJERCICIO: formulario Login básico

Paso a paso: Crear un sistema de login de usuario en React

Este ejercicio te guiará paso a paso para crear un sencillo sistema de login de usuario en React. Aprenderás a utilizar conceptos como state, manejo de eventos y renderizado condicional.

1. Configuración del proyecto:

- Crea un nuevo proyecto de React utilizando `create-react-app`:

```
npx create-react-app login-app  
cd login-app
```

- (OPCIONAL) Instala un gestor de estilos opcional como Bootstrap o Material-UI para mejorar la apariencia del formulario. Lo veremos en la siguiente unidad.

2. Crear componentes:

- Crea dos componentes: `Login` y `Mensaje`.
- El componente `Login` contendrá el formulario de login con campos para usuario y contraseña, y un botón de inicio de sesión.
- El componente `Mensaje` mostrará un mensaje de bienvenida o de error dependiendo del resultado del login.

3. Implementar el estado:

- En el componente `Login`, utiliza el hook `useState` para almacenar el estado del usuario (username) y la contraseña.
- Inicializa el estado con valores vacíos ('').

4. Manejar eventos:

- Define funciones para manejar el cambio en los campos de usuario y contraseña.
- Estas funciones actualizarán el estado correspondiente utilizando `setState`.
- Define una función para manejar el evento de clic del botón de inicio de sesión.
- Esta función comparará el usuario y contraseña ingresados con valores predefinidos (o obtenidos de una base de datos real).
- Si las credenciales son correctas, actualiza el estado de `mensaje` con un mensaje de bienvenida (`Bienvenido, [username]!`).

- Si las credenciales son incorrectas, actualiza el estado de `mensaje` con un mensaje de error (Usuario o contraseña incorrectos).

5. Renderizado condicional:

- En el componente `Login`, muestra el componente `Mensaje` solo si el estado de `mensaje` tiene un valor.
- Renderiza el mensaje de bienvenida o de error dependiendo del contenido del estado de `mensaje`.

6. Ejemplo de código Login.jsx:

```
// Login.js
import React, { useState } from 'react';

const Login = () => {
  const [usuario, setUsuario] = useState('');
  //...Sigue añadiendo campos

  const handleUsuarioChange = (e) => setUsuario(e.target.value);
  //...Sigue añadiendo campos

  const handleSubmit = (e) => {
    e.preventDefault();
    const usernameCorrecto = 'admin';
    const passwordCorrecto = '123456';

    if (usuario === usernameCorrecto && contraseña === passwordCorrecto) {
      //...Sigue añadiendo campos
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="usuario">Usuario:</label>
      <input
        //...Sigue añadiendo campos
      </form>
    );
};

export default Login;

// Mensaje.js
const Mensaje = ({ mensaje }) => {
  return <p>{mensaje}</p>;
};

export default Mensaje;
```

7. Ejecutar la aplicación:

- Ejecuta la aplicación con `npm start` o `yarn start`.
- Ingresa un usuario y contraseña correctos y verifica que se muestra el mensaje de bienvenida.

- Prueba con credenciales incorrectas y verifica que se muestra el mensaje de error.

8. Mejoras opcionales:

- Puedes crear otros componentes que agreguen realismo al sitio.
- Añade estilos App.css para mejorar la imagen web.

Recuerda: Este es un ejemplo básico. Puedes adaptarlo y ampliarlo según tus necesidades y creatividad.

CUESTIONARIO

1 ¿Cómo se crea un nuevo proyecto de React usando Create React App?

- a) Ejecutando el comando "npx create-react-app mi-proyecto".
- b) Descargando un archivo ZIP desde el sitio web oficial de React.
- c) Usando el comando "npm create-react-app mi-proyecto".
- d) Instalando manualmente todas las dependencias necesarias.

Respuesta correcta: a) Ejecutando el comando "npx create-react-app mi-proyecto".

2 ¿Cuál es la principal diferencia entre un componente funcional y un componente de clase en React?

- a) Los componentes funcionales tienen un rendimiento mejor que los de clase.
- b) Los componentes de clase pueden tener estado (state), mientras que los funcionales no.
- c) Los componentes de clase se definen con la palabra clave "class", mientras que los funcionales no.
- d) Los componentes funcionales se utilizan solo para propósitos de visualización, mientras que los de clase son más versátiles.

Respuesta correcta: b) Los componentes de clase pueden tener estado (state), mientras que los funcionales no.

3 ¿Cuál es la forma correcta de definir un componente de clase en React?

- a) `function MyComponent() {}`
- b) `const MyComponent = () => {}`
- c) `class MyComponent extends Component {}`
- d) `const MyComponent = class {}`

Respuesta correcta: c) `class MyComponent extends Component {}`

4 ¿Qué es una prop en React?

- a) Una función que se ejecuta cuando un evento ocurre en un componente.
- b) Una variable que se pasa de un componente a otro como atributo.
- c) Una forma de representar el estado interno de un componente.
- d) Un método que se utiliza para actualizar el estado de un componente.

Respuesta correcta: b) Una variable que se pasa de un componente a otro como atributo.

5 ¿Cuál es la forma correcta de definir y utilizar el estado en un componente funcional en React?

- a) Utilizando la palabra clave "this.state" dentro del componente.
- b) Declarando una variable global para el estado y actualizándola directamente.

- c) Utilizando el hook `useState()` proporcionado por React.
- d) Pasando el estado como una prop al componente.

Respuesta correcta: c) Utilizando el hook `useState()` proporcionado por React.

6 ¿Qué hook de React se utiliza para manejar eventos en un componente funcional?

- a) `useEffect()`
- b) `useState()`
- c) `useMemo()`
- d) `useCallback()`

Respuesta correcta: d) `useCallback()`

7 ¿Cuál de las siguientes opciones describe mejor el concepto de "comunicación hijo-padre" en React?

- a) La capacidad de un componente padre para enviar datos a un componente hijo mediante props.
- b) La capacidad de un componente hijo para enviar datos a un componente padre mediante props.
- c) La capacidad de un componente hijo para acceder y modificar directamente el estado de un componente padre.
- d) La capacidad de un componente padre para acceder y modificar directamente el estado de un componente hijo.

Respuesta correcta: b) La capacidad de un componente hijo para enviar datos a un componente padre mediante props.

8 ¿Qué ciclo de vida de React se ejecuta solo una vez cuando un componente se monta en el DOM?

- a) `render()`
- b) `componentDidMount()`
- c) `componentDidUpdate()`
- d) `componentWillUnmount()`

Respuesta correcta: b) `componentDidMount()`

9 ¿Qué método se utiliza para actualizar el estado de un componente en React?

- a) `setState()`
- b) `updateState()`
- c) `changeState()`
- d) `modifyState()`

Respuesta correcta: a) `setState()`

10 ¿Qué es el "two-way binding" en React?

- a) Un enfoque que permite la comunicación unidireccional entre componentes.
- b) Una técnica que vincula el estado de un componente con la entrada del usuario, actualizando automáticamente ambos.
- c) Un patrón de diseño que conecta múltiples componentes para compartir datos entre ellos.
- d) Un método para prevenir la re-renderización innecesaria de componentes en React.

Respuesta correcta: b) Una técnica que vincula el estado de un componente con la entrada del usuario, actualizando automáticamente ambos.

PARA SABER MÁS

Componentes:

[CLASE DEFINITIVA de REACT: TODO LO QUE DEBES SABER DE COMPONENTES EN 2023!](#)

Eventos:

[8 Manejo de eventos en React JS - Curso ReactJS - OpenBootcampp](#)

Ciclo de vida:

[Curso React: 13. Ciclo de Vida de los Componentes - jonmircha](#)