

# Tutorial de React

Aplicaciones distribuidas en internet

Aurora Andreu Mateo, Juan Fernando Gaviria Álvarez y  
Adrián Federico Bartel Prieto



# Contenido

1. Introducción a React. ....	3
2. Configuración.....	3
3. JSX: Sintaxis de React.....	5
4. Componentes en React.....	6
5. Estado y ciclo de vida en componentes de clase.....	8
Manejo del estado.....	9
6. Hooks en React .....	10
Uso de useState y useEffect .....	11
Reglas y mejores prácticas para usar Hooks.....	12
7. Manejando Eventos.....	12
Cómo manejar eventos en React.....	12
Binding de eventos en componentes de clase .....	13
Uso de eventos en componentes funcionales .....	13
8. Renderizando condicional y listas.....	14
Técnicas para renderizar condicionalmente componentes .....	14
Uso de map para renderizar listas de elementos .....	15
9. Formularios en React.....	17
10. Estilos y CSS en React.....	20
11. Rutas y navegación con React Router.....	22
12. Llamadas a API's y manejo de datos externos .....	23

## 1. Introducción a React.

React es una biblioteca de JavaScript muy utilizada actualmente que se centra exclusivamente en construir interfaces de usuario. En esta biblioteca se introduce el concepto de componentes, que son interfaces de usuario independientes y reutilizables. De esta forma, tanto el desarrollo como el funcionamiento de la interfaz es fácil, ya que permite una actualización automática. Es decir, se diseñan vistas simples para cada estado de la aplicación y React actualizará y renderizará los componentes adaptándose a los cambios realizados. Una de las fundamentales ventajas de React es que permite desarrollar aplicaciones tanto web como móviles. En resumen, es una biblioteca eficiente y flexible.

## 2. Configuración

Primero vamos a ver como instalar Node.js y Node Package Manager (NPM). Node.js es un entorno de ejecución para JavaScript en el servidor y NPM es el sistema de gestión de paquetes que le acompaña.

Para la instalación de Node.js hay que entrar en el sitio web de abajo e instalar la versión recomendada para el sistema operativo que se vaya a usar.

<https://nodejs.org/en>

Una vez realizado el proceso de instalación, para comprobar que se ha instalado correctamente, abrimos una terminal y comprobamos las versiones de node y de NPM:

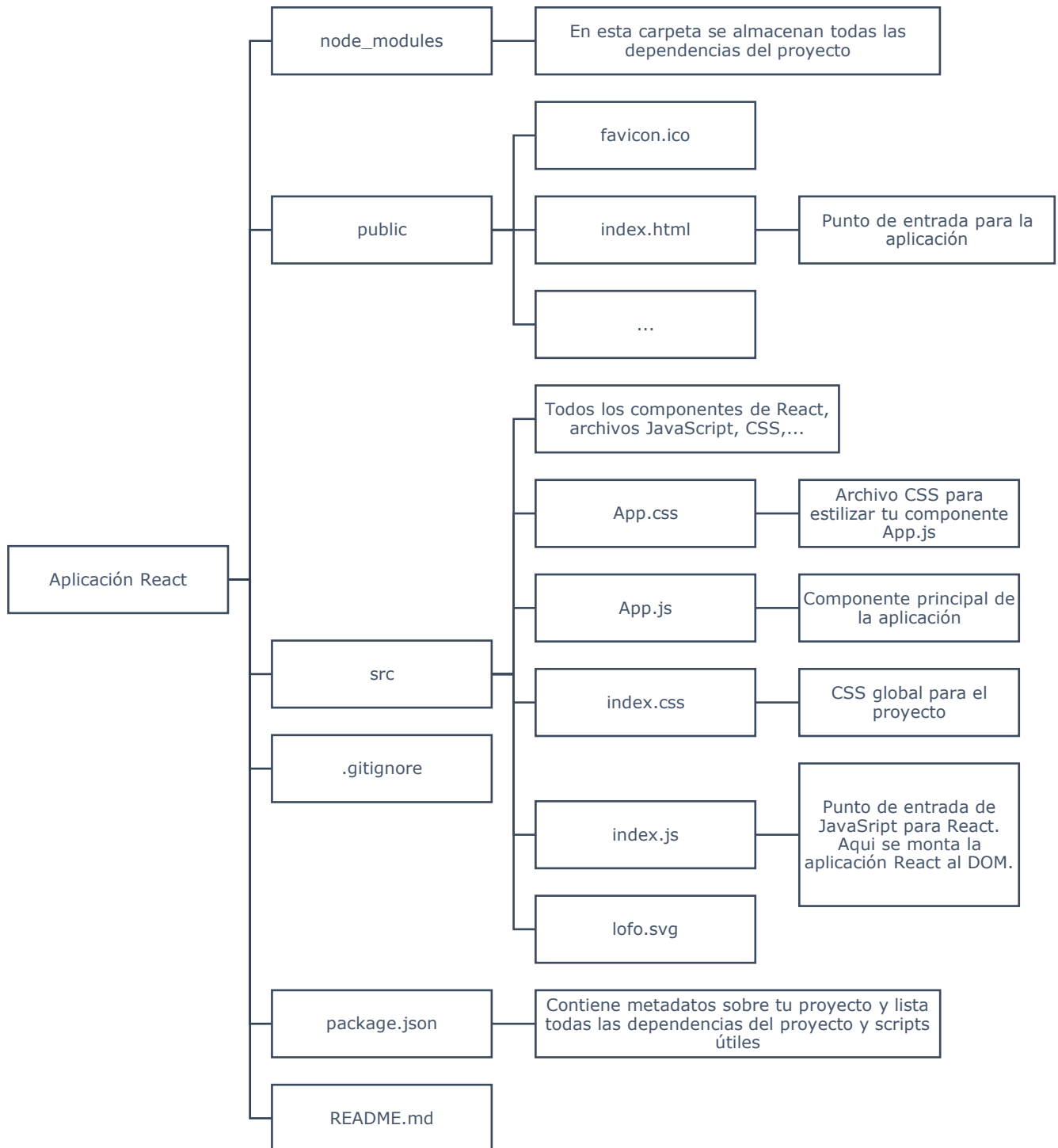
```
node -v
```

```
npm -v
```

Para crear un nuevo proyecto con React vamos a usar la herramienta Create React App, que es una herramienta para crear aplicaciones React con una configuración mínima. Para ello, usamos el siguiente comando:

```
npx create-react-app [nombre-de-la-aplicación]
```

La estructura de directorios básica que vamos a encontrar en un proyecto React es:



Para ejecutar la aplicación:

```
npm start
```

### 3. JSX: Sintaxis de React

JSX es una extensión de sintaxis para JavaScript que se usa principalmente con React para describir como debería verse la interfaz de usuario. JSX es una sintaxis que se convierte en llamadas a funciones de JavaScript, permitiendo de esta forma desarrollar estructuras en un formato visualmente mas cercano a HTML, pero con poder JavaScript.

Aunque visualmente JSX y HTML son similares, tienen diferencias clave:

- JSX se procesa en JavaScript permitiendo incorporar lógica de JavaScript a la estructura de la interfaz de usuario
- Algunos nombres de atributos difieren debido a su nomenclatura
- JSC cierra todos los elementos

En JSX, se pueden insertar expresiones JavaScript dentro de llaves para integrar dinámicamente valores y lógica en la interfaz de usuario.

```
const nombre = "Aurora"
const saludo = <h1> Hola, {nombre} </h1>
```

```
const nombre = "Aurora"
const saludo = <h1> Hola, {nombre} </h1>
```

En los componentes de React, además de JavaScript, se pueden incluir JSX. Podemos tener dos tipos: Componentes de Clase y Componentes Funcionales. En los dos próximos ejemplos, el componente Saludar toma propiedades y las utiliza dentro de JSX. Se podrían usar estos componentes de la misma manera que usarías etiquetas HTML. Los componentes de clase serían:

```
class Saludar extends React.Component {

  render() {

    return <h1>Hola, {this.props.nombre}</h1>;

  }

}
```

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

En cambio, los componentes funcionales

```
function Saludar( props ) {

  return <h1>Hola, {props.nombre}</h1>;

}
```

```
function Saludar( props ) {
  return <h1>Hola, {props.nombre}</h1>;
}
```

El código HTML para ambos componentes seria:

```
const App = () => {

  return <Saludar nombre="Aurora" />;

}
```

```
const App = () => {
  return <Saludar nombre='Aurora' />;
}
```

En este ejemplo le estamos pasando la propiedad nombre.

## 4. Componentes en React

Como hemos introducido de forma breve anteriormente, en React hay dos tipos principales de componentes: funcionales y de clase.

Las componentes funcionales son la forma mas sencilla de escribir componentes. Son funciones de JavaScript que retornan elementos de React (JSX). Pueden aceptar argumentos (props) que permiten pasar datos al componente. Este tipo de componentes puede manejar estado y efectos secundarios.

En cambio, los componentes de clase extienden de 'React.Component'. tienen un método render() que retorna elementos de React. Pueden mantener su propio estado privado y manejar ciclos de vida del componente.

Los componentes pueden utilizarse en otros componentes, por ejemplo, aquí se usa el componente Saludar dos veces pasando diferentes nombres como propiedad.

```
function App() {  
  
  return (  
  
    <div>  
  
      <Saludar nombre="Aurora" />  
  
      <Saludar nombre="Juan" />  
  
    </div>  
  
  );  
}
```

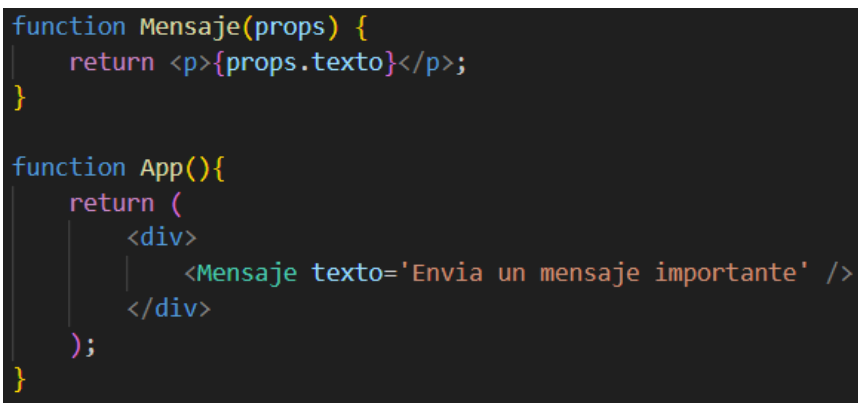


```
function App() {  
  return (  
    <div>  
      <Saludar nombre='Aurora' />  
      <Saludar nombre='Juan' />  
    </div>  
  );  
}
```



Las propiedades son la forma de pasar datos a un componente padre a un componente hijo en React. Son propiedades de lectura, lo que significa que un componente hijo no puede modificar propiedades que recibe del padre.

```
function Mensaje(props) {  
  
    return <p>{props.texto}</p>;  
  
}  
  
function App() {  
  
    return (  
  
        <div>  
  
            <Mensaje texto="Envia un mensaje importante" />  
  
        </div>  
  
    );  
  
}
```



```
function Mensaje(props) {  
    return <p>{props.texto}</p>;  
}  
  
function App(){  
    return (  
        <div>  
            <Mensaje texto='Envia un mensaje importante' />  
        </div>  
    );  
}
```

## 5. Estado y ciclo de vida en componentes de clase

El estado es una característica importante de React que permite a los componentes almacenar datos que pueden cambiar a lo largo del tiempo. El ciclo de vida de un componente es una serie de eventos que se producen cuando un componente se crea, se actualiza o se elimina.

## Manejo del estado

Para manejar el estado en un componente de clase, se utilizan los métodos `this.state` y `this.setState()`.

- `this.state` es un objeto que almacena el estado actual del componente.
- `this.setState()` actualiza el estado actual del componente con un nuevo objeto.

Por ejemplo, el siguiente código crea un componente de clase con un estado que almacena el número de clics que ha recibido:

```
class Contador extends React.Component {
  state = {
    clicks: 0
  };
  render() {
    return (
      <div>
        <p>Número de clics: {this.state.clicks}</p>
        <button onClick={this.incrementarClics}>Incrementar</button>
      </div>
    );
  }
  incrementarClics() {
    this.setState({
      clicks: this.state.clicks + 1
    });
  }
}
```

Cuando el usuario hace clic en el botón, se llama al método `incrementarClics()`, que actualiza el estado del componente con un valor de `clicks` incrementado en 1.

### 1. Métodos del ciclo de vida

Los componentes de clase tienen una serie de métodos del ciclo de vida que se llaman en diferentes momentos de su vida. Los métodos del ciclo de vida más comunes son:

- `componentWillMount()`: Se llama antes de que el componente se monte por primera vez.
- `render()`: Se llama cada vez que el componente se renderiza.
- `componentDidMount()`: Se llama después de que el componente se monte por primera vez.
- `componentWillReceiveProps()`: Se llama antes de que se actualicen las propiedades del componente.
- `shouldComponentUpdate()`: Se llama antes de que se actualice el componente.
- `componentDidUpdate()`: Se llama después de que se actualice el componente.

Por ejemplo, el siguiente código muestra cómo utilizar el método `componentDidMount()` para realizar una acción después de que el componente se monte por primera vez:

```
class Contador extends React.Component {
  state = {
    clicks: 0
  };
  componentDidMount() {
    // Realizar una acción después de que el componente se monte
  }
  render() {
    return (
      <div>
        <p>Número de clics: {this.state.clicks}</p>
        <button onClick={this.incrementarClicks}>Incrementar</button>
      </div>
    );
  }
  incrementarClicks() {
    this.setState({
      clicks: this.state.clicks + 1
    });
  }
}
```

## 6. Hooks en React

Los hooks son una nueva característica de React que permiten utilizar el estado y el ciclo de vida de React desde componentes funcionales.

Los hooks se utilizan mediante la palabra clave `use()`. Cada hook tiene su propia sintaxis y comportamiento.

Por ejemplo, el siguiente código utiliza el hook `useState()` para manejar el estado de un componente:

```
const Contador = () => {
  const [clicks, setClicks] = useState(0);
  return (
    <div>
      <p>Número de clics: {clicks}</p>
      <button onClick={() => setClicks(clicks + 1)}>Incrementar</button>
    </div>
  );
};
```

### Uso de useState y useEffect

El hook useState() permite manejar el estado de un componente funcional. El hook devuelve un par de valores: el valor actual del estado y una función para actualizar el estado.

Por ejemplo, el siguiente código utiliza el hook useState() para crear un componente que muestra el número de clics que ha recibido:

```
const Contador = () => {
  const [clicks, setClicks] = useState(0);
  return (
    <div>
      <p>Número de clics: {clicks}</p>
      <button onClick={() => setClicks(clicks + 1)}>Incrementar</button>
    </div>
  );
};
```

Cuando el usuario hace clic en el botón, se llama a la función setClicks(), que actualiza el estado del componente con un valor de clicks incrementado en 1.

El hook useEffect() permite ejecutar código cuando el componente se monta, se actualiza o se desmonta. El hook acepta una función que se ejecuta en el contexto del componente.

Por ejemplo, el siguiente código utiliza el hook useEffect() para realizar una acción después de que el componente se monte por primera vez:

```
const Contador = () => {
  useEffect(() => {
    // Realizar una acción después de que el componente se monte
  }, []);
  return (
    <div>
      <p>Número de clics: {clicks}</p>
      <button onClick={() => setClicks(clicks + 1)}>Incrementar</button>
    </div>
  );
};
```

## Reglas y mejores prácticas para usar Hooks

Hay algunas reglas y mejores prácticas que se deben seguir al usar hooks:

- Los hooks no deben utilizarse dentro de clases.
- Los hooks deben utilizarse en el contexto del componente en el que se definen.
- Los hooks deben utilizarse de forma independiente y no deben depender de otros hooks.

## 7. Manejando Eventos

Los eventos son una forma de que los usuarios interactúen con una aplicación React.

### Cómo manejar eventos en React

Para manejar eventos en React, se utilizan los atributos `onClick`, `onChange`, `onBlur`, etc. Estos atributos aceptan una función que se llama cuando se produce el evento.

Por ejemplo, el siguiente código utiliza el atributo `onClick` para manejar el evento de clic en un botón:

```
const Boton = () => {
  return (
    <button onClick={() => alert('Hola!')}>Click aquí</button>
  );
};
```

Cuando el usuario hace clic en el botón, se llama a la función `alert()`, que muestra un mensaje de alerta.

### Binding de eventos en componentes de clase

En componentes de clase, se utiliza el método `bind()` para vincular una función a un evento.

Por ejemplo, el siguiente código utiliza el método `bind()` para vincular una función a un evento de clic:

```
class Boton extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    alert('Hola!');
  }
  render() {
    return (
      <button onClick={this.handleClick}>Click aquí</button>
    );
  }
}
```

### Uso de eventos en componentes funcionales

En componentes funcionales, se puede utilizar el operador flecha para vincular una función a un evento.

```
const Boton = () => {
  return (
    <button onClick={(e) => alert(e.target.id)}>Click aquí</button>
  );
};
```

En el ejemplo anterior, la función `alert()` se vincula al evento `onClick` del botón. La función recibe un parámetro, que es el objeto `e` que contiene información sobre el evento. En este caso, se utiliza el parámetro `e` para obtener el ID del elemento que ha desencadenado el evento.

Las funciones vinculadas a eventos en componentes funcionales se ejecutan en el contexto del componente. Esto significa que tienen acceso a todas las propiedades y métodos del componente.

## 8. Renderizando condicional y listas

### Técnicas para renderizar condicionalmente componentes

En React, se pueden utilizar diferentes técnicas para renderizar condicionalmente componentes.

Una técnica común es utilizar la sentencia `if`.

Por ejemplo, el siguiente código utiliza la sentencia `if` para renderizar un componente `<P>` solo si la variable `mostrar` tiene el valor `true`:

```
const ComponenteCondicional = () => {
  const mostrar = true;
  return (
    <div>
      {mostrar && <P>Este componente se renderizará</P>}
    </div>
  );
};
```

Otra técnica común es utilizar el operador &&.

Por ejemplo, el siguiente código utiliza el operador && para renderizar un componente <P> solo si la variable mostrar tiene el valor true y la variable edad es mayor que 18:

```
const ComponenteCondicional = () => {
  const mostrar = true;
  const edad = 20;
  return (
    <div>
      {mostrar && edad > 18 && <P>Este componente se renderizará</P>}
    </div>
  );
};
```

### Uso de map para renderizar listas de elementos

El hook map() permite renderizar una lista de elementos de forma iterativa.

Por ejemplo, el siguiente código utiliza el hook map() para renderizar una lista de números:



```
const ListaNumeros = () => {  
  const numeros = [1, 2, 3, 4, 5];  
  return (  
    <div>  
      {numeros.map((numero, index) => (  
        <p key={index}>{numero}</p>  
      ))}  
    </div>  
  );  
};
```

## 9. Formularios en React

En React, el estado es una parte fundamental para manejar la información en un componente. Para un formulario, podemos utilizar el hook **useState** para crear un estado que almacene los datos del formulario. Esto nos permite rastrear y actualizar los valores de los campos del formulario.

Para capturar los cambios en los campos del formulario, podemos definir un manejador de cambio (**handleChange**) que actualice el estado del formulario cada vez que un usuario introduce información. Además, podemos definir un manejador de envío (**handleSubmit**) que se ejecuta cuando el formulario se envía, permitiéndonos realizar acciones como enviar datos a un servidor.

```

import React, { useState } from "react";

const Formulario = () => {
  // Estado para almacenar los datos del formulario
  const [formData, setFormData] = useState({
    nombre: "",
    email: "",
  });

  // Manejador de cambios en los inputs
  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({ ...formData, [name]: value });
  };

  // Manejador de envío del formulario
  const handleSubmit = (e) => {
    e.preventDefault();
    // Aquí se puede realizar acciones con los datos del formulario
    console.log("Datos del formulario:", formData);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Nombre:
        <input
          type="text"
          name="nombre"
          value={formData.nombre}
          onChange={handleChange}
        />
      </label>
      <br />
      <label>
        Email:
        <input
          type="email"
          name="email"
          value={formData.email}
          onChange={handleChange}
        />
      </label>
      <br />
      <button type="submit">Enviar</button>
    </form>
  );
};

export default Formulario;

```

Cuando un usuario ingresa información en un campo, el estado asociado a ese campo debe actualizarse. Esto se realiza mediante la función **set** proporcionada por el hook **useState**. A veces es bueno tener un estado individual para cada campo del formulario.

```

import React, { useState } from "react";

const Formulario = () => {
  const [nombre, setNombre] = useState("");
  const [email, setEmail] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log("Nombre:", nombre);
    console.log("Email:", email);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Nombre:
        <input
          type="text"
          value={nombre}
          onChange={(e) => setNombre(e.target.value)}
        />
      </label>
      <br />
      <label>
        Email:
        <input
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
        />
      </label>
      <br />
      <button type="submit">Enviar</button>
    </form>
  );
};

export default Formulario;

```

Se puede añadir validaciones básicas como que el nombre o el email no estén vacíos. También podemos agregar otro estado para manejar mensajes de error.

```

const Formulario = () => {
  const [nombre, setNombre] = useState('');
  const [email, setEmail] = useState('');
  const [error, setError] = useState('');

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

    // Validación básica
    if (!nombre || !email) {
      setError('Todos los campos son obligatorios');
      return;
    }

    // Lógica de envío si la validación es exitosa
    console.log('Nombre:', nombre);
    console.log('Email:', email);
    setError('');
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Nombre:
        <input type="text" value={nombre} onChange={(e) => setNombre(e.target.value)} />
      </label>
      <br />
      <label>
        Email:
        <input type="email" value={email} onChange={(e) => setEmail(e.target.value)} />
      </label>
      <br />
      <button type="submit">Enviar</button>
      {error && <p style={{ color: 'red' }}>{error}</p>}
    </form>
  );
};

export default Formulario;

```

## 10. Estilos y CSS en React

Podemos aplicar estilos directamente en línea utilizando el atributo **style** en un componente de React. Esto es útil para estilos simples y específicos de un solo componente.

```
const MiComponente = () => {
  return (
    <div style={{ color: 'blue', fontSize: '16px', padding: '10px' }}>
      Contenido del componente
    </div>
  );
};
```

También podemos usar clases de CSS externas para aplicar estilos a tus componentes. Importa el archivo de estilos y aplica las clases al componente.

```
import './MiComponente.css';

const MiComponente = () => {
  return <div className="mi-componente">Contenido del componente</div>;
};
```

CSS Modules es una técnica que permite el encapsulamiento de estilos en archivos CSS para componentes específicos. Esto evita posibles conflictos de nombres de clases.

```
import React from 'react';
import styles from './MiComponente.module.css';

const MiComponente = () => {
  return <div className={styles.miComponente}>Contenido del componente</div>;
};
```

**styled-components** es una biblioteca popular para escribir estilos de componentes con JavaScript dentro de los archivos de React. Ofrece un enfoque más dinámico y permite la creación de componentes reutilizables con estilos.

```
npm install styled-components
```

```
import styled from 'styled-components';

const StyledDiv = styled.div`
  color: purple;
  font-size: 20px;
  padding: 15px;
`;

const MiComponente = () => {
  return <StyledDiv>Contenido del componente</StyledDiv>;
};
```

Ambos enfoques, CSS Modules y styled-components, ofrecen soluciones efectivas para manejar estilos en aplicaciones de React. La elección entre ellos dependerá de las preferencias y necesidades específicas del proyecto.

## 11. Rutas y navegación con React Router

React Router es una librería popular que facilita la navegación y gestión de rutas en aplicaciones React. Lo primero que debemos hacer es instalar la librería:

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

Y en el componente principal implementamos la siguiente configuración:

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

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

En nuestro componente principal podemos definir las rutas utilizando componentes proporcionados por React Router, como **Route** y **Switch**.

```

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;

```

Para permitir la navegación entre páginas, puedes usar componentes proporcionados por React Router, como [Link](#).

```

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

const Header = () => {
  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 Header;

```

## 12. Llamadas a API's y manejo de datos externos



**fetch** es una función nativa de JavaScript para realizar solicitudes HTTP. Es compatible con la mayoría de los navegadores modernos y proporciona una interfaz simple y promisificada. Aquí hay un ejemplo básico de cómo usar **fetch** en React:

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

const MiComponente = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://api.example.com/data');
        const result = await response.json();
        setData(result);
      } catch (error) {
        console.error('Error al obtener datos:', error);
      }
    };

    fetchData();
  }, []); // Dependencia vacía para ejecutar solo una vez al montar el componente

  return (
    <div>
      {data ? (
        // Renderizar los datos obtenidos
        <p>{data.message}</p>
      ) : (
        // Renderizar mientras se obtienen los datos
        <p>Cargando...</p>
      )}
    </div>
  );
};

export default MiComponente;
```

**axios** es una biblioteca externa que simplifica las solicitudes HTTP y proporciona una sintaxis limpia y amigable. Necesitamos instalarla antes de usarla:

```
npm install axios
```

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

const MiComponente = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get("https://api.example.com/data");
        setData(response.data);
      } catch (error) {
        console.error("Error al obtener datos:", error);
      }
    };

    fetchData();
  }, []); // Dependencia vacía para ejecutar solo una vez al montar el componente

  return (
    <div>
      {data ? (
        // Renderizar los datos obtenidos
        <p>{data.message}</p>
      ) : (
        // Renderizar mientras se obtienen los datos
        <p>Cargando...</p>
      )}
    </div>
  );
};

export default MiComponente;
```

En React cuando realizamos solicitudes HTTP o trabajamos con operaciones asíncronas, es importante manejar el estado del componente durante diferentes fases del ciclo de vida. Aquí hay un ejemplo básico utilizando **useState** y **useEffect**:

```

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

const MiComponente = () => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get('https://api.example.com/data');
        setData(response.data);
      } catch (error) {
        setError(error);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, []); // Dependencia vacía para ejecutar solo una vez al montar el componente

  return (
    <div>
      {loading && <p>Cargando...</p>}
      {error && <p>Error: {error.message}</p>}
      {data && <p>{data.message}</p>}
    </div>
  );
};

export default MiComponente;

```

**loading** se utiliza para indicar si los datos están siendo cargados.

**error** captura cualquier error que pueda ocurrir durante la solicitud.

**data** almacena los datos una vez que se han obtenido con éxito.

## 13. Bibliografía

<https://softwarecrafters.io/react/tutorial-react-js-introduccion>

React documentation: Hooks: <https://reactjs.org/docs/hooks-intro.html>: <https://reactjs.org/docs/hooks-intro.html>

React documentation: useState: <https://reactjs.org/docs/hooks-state.html>: <https://reactjs.org/docs/hooks-state.html>

React documentation: useEffect: <https://reactjs.org/docs/hooks-effect.html>: <https://reactjs.org/docs/hooks-effect.html>

React documentation: Events: <https://reactjs.org/docs/events.html>: <https://reactjs.org/docs/events.html>

React documentation: Rendering lists: <https://reactjs.org/docs/lists-and-keys.html>: <https://reactjs.org/docs/lists-and-keys.html>

React documentation: State and Lifecycle: <https://reactjs.org/docs/state-and-lifecycle.html>

<https://es.legacy.reactjs.org/docs/forms.html>

<https://develohero.io/blog/como-manejar-formularios-en-react-desde-cero>

<https://es.legacy.reactjs.org/docs/code-splitting.html#route-based-code-splitting>

<https://es.react.dev/reference/react-dom/components/input#input>