REACT

rortegag

Importación/Exportación

Importación por defecto.

Importación exportación por defecto.

```
import ExampleComponent from 'path/to/file';
```

Puede importar las exportaciones por defecto desde un archivo utilizando la sintaxis anterior.

Exportación por defecto.

```
export default ExampleComponent;
```

Utilice las palabras clave export y default para exportar un componente.

Importación con nombre.

Exportaciones con nombre.

```
export const capitalize = word => {
 ...
};

export const fetchUserData = userID => {
 ...
};

export const getUserName = userObject => {
 ...
};
```

Puede exportar tantos elementos con nombre de un fichero como desee. Para ello, debe utilizar la palabra clave 'export' SIN la palabra clave 'default'.

Importación de exportaciones con nombre.

```
import { capitalize, fetchUserData, getUserName } from 'path/to/file';
```

Para importar las exportaciones con nombre, debe enumerarlas explícitamente entre llaves con los nombres con los que se han exportado.

```
import { ExampleComponent as AliasedExampleComponent } from 'path/to/file';
import { capitalize as cap } from 'path/to/file';
```

Para crear un alias puede utilizar la palabra clave 'as' seguida del nuevo nombre local, que será el nombre con el que se utilizará el componente importado en el contexto actual.

Importar todo

```
import * as AliasedAllImport from 'path/to/file';
```

Puede importar todas las exportaciones con nombre de un archivo utilizando la '* as sintasis'.

Componentes

Sintaxis del Class Component.

Definición de un Class Component.

```
class ExampleClassComponent extends React.Component {
  render() {
    return <h1>Example Class Component</h1>;
  }
}
```

Los Class Components eran una parte importante de React antes de introducir React Hooks. Aunque la mayoría de los nuevos proyectos prefieren Functional Components, todavía hay muchos proyectos basados en React que utilizan Class Components.

MÁS ↓

REACT

rortegag

Componentes

Sintaxis del Functional Component.

Componente funcional definido mediante la palabra clave function.

```
function ExampleFunctionalComponent() {
  return <h1>Example Class Component</h1>;
}
```

Componente funcional definido mediante la arrow function.

```
const ExampleFunctionalComponent = () => {
 return <h1>Example Class Component</h1>;
}
```

Los Functional Components son el futuro de React. A pesar de que anteriormente solo se utilizaban para componentes de interfaz de usuario sin estado, ahora son la forma preferida de construir nuevos componentes.

Pure Functional Component.

```
import { memo } from 'React';

const ExamplePureComponent = memo({ portal }) => {
  return (
    <h1>Welcome to {portal}!</h1>
  );
}
```

Los Pure (Functional) Components siempre generan la misma salida dada la misma entrada (estado y props).

Props

Uso de props en Class Components.

Utilizar props directamente desde el objeto props.

```
class PropsExample extends React.Component {
  render() {
    return <h1>Welcome {this.props.portalName}</h1>;
  }
}
```

La forma más sencilla de utilizar props en Class Components en React. Se leen directamente desde los props.

Utilizar props desestructurando el objeto props.

```
class PropsExample extends React.Component {
  render() {
    const { portalName } = this.props;

    return <h1>Welcome to {portalName}</h1>;
  }
}
```

Una forma más eficiente de utilizar props en Class Components es utilizar la asignación de desestructuración. Entonces podemos utilizar los props directamente.

Uso de props en Functional Components.

Utilizar props directamente desde el objeto props.

```
const PropsExample = (props) => {
  return <h1>Welcome to {props.portalName}</h1>;
}
```

Con Functional Components, tienes que esperar que el objeto props sea pasado desde el padre.
A continuación, puede acceder a los props en el objeto props.

Utilizar props desestructurando el objeto props.

```
const PropsExample = ({ portalName }) => {
  return <h1>Welcome to {portalName}</h1>;
}
```

Una forma más eficiente de utilizar props en Functional Components es utilizar la asignación de desestructuración. De este modo podemos utilizar los props directamente.

MÁS ↓

Para más información: https://reactjs.org/

REACT

rortegag

Props

Pasando props.

Pasar props directamente.

```
const PropsExample = () => {
  const userAge = 21;
  const isOverEighteen = userAge > 18;

  return (
    <TargetComponent
      portalName="Upmostly"
      userAge={userAge}
      isOverEighteen={isOverEighteen}
      beigeBackground
    />
  );
};
```

Puedes pasar props al componente hijo/destino asignándolos como atributos a los elementos JSX uno a uno.

Children props.

Pasando los children en JSX.

```
const App = () => {
  return (
    <>
      <ParentComponent>
        <ul>
          <li>This</li>
          <li>is</li>
          <li>children</li>
          <li>element</li>
        </ul>
      </ParentComponent>
    </>
  );
};
```

Puedes pasar los children a los componentes proporcionándolos a través de JSX. Estos deben estar situados entre las etiquetas de apertura y cierre de los elementos a los que desea pasarlos.

Utilización de children prop.

```
const ParentComponent = ({ children }) => {
  return (
    <div>
      {children}
    </div>
  );
};
```

Para usar children puedes usar el elemento children en el objeto props. En el ejemplo anterior, estamos desestructurando el objeto en la definición del componente y devolviendo los children en JSX.

MÁS

Pasando props eficazmente.

Expresiones JS.

```
import { useState } from 'react';
import Counter from './User';

function App() {
  const [year, setYear] = useState(1237);

  return (
    <div>
      <Counter
        currentCount={year + 3}
        maxCount={year + (year * 200)}
      />
    </div>
  );
}
```

Las expresiones JS serán evaluadas y su resultado será pasado como props.

Template Literals.

```
import { useState } from 'react';
import Counter from './User';

function App() {
  const [year, setYear] = useState(1237)
  return (
    <div>
      <Counter
        warning={`Current year is ${year}`}
      />
    </div>
  );
};
```

Puedes utilizar los template literals para construir cadenas de texto de forma eficiente.

Difundir.

```
import User from './User';

function App() {
  const user = {
    name: 'Tommy',
    surname: 'Smith'
  };

  return (
    <div>
      <User name={user.name} surname={user.surname} />
      <User {...user} />
    </div>
  );
}
```

Puedes extender objetos para pasar todas sus propiedades como props individuales.

Para más información: https://reactjs.org/

REACT

rortegag

Props

Default.

defaultProps.

```
ExampleComponent.defaultProps = {
  name: 'Default name',
  surname: 'Default surname',
  age: 'Default age',
};
```

Puedes definir los props por defecto asignando un objeto con los props relevantes a la propiedad defaultProps disponible en cada componente.

Pasando props eficazmente.

Inline.

```
const User = ({
  name = 'Default name',
  surname = 'Default surname',
  age = 'Default age',
  color = 'lightgreen'
}) => {
  return (
    <div style={{
      backgroundColor: color,
      padding: '10px',
      marginBottom: '10px'
    }}>
      <h2>Name: {name}</h2>
      <h2>Surname: {surname}</h2>
      <h2>Age: {age}</h2>
    </div>
  );
};
```

Aplicación sencilla que utiliza accesorios predeterminados en la definición del componente.

State en Functional Components.

State de lectura.

```
const UserAccount = () => {
  const [username, _] = useState('mike123');
  const [userAge, setUserAge] = useState(21);

  return (
    <div>
      <h2>User account</h2>
      <ul>
        <li>Username: {username}</li>
        <li>Age: {userAge}</li>
      </ul>
    </div>
  );
};
```

Con Functional Components, puedes gestionar el state utilizando el hook useState. Devuelve dos valores: el primero es la variable de estado de sólo lectura que se puede utilizar para acceder al valor de estado, y el segundo es la devolución de llamada que se utilizará para actualizar esa variable de estado con el valor pasado como su argumento.

State

State en Class Components.

State de lectura.

```
class UserAccount extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      username: 'mike123',
      userAge: 21
    };
  }

  render() {
    return (
      <div>
        <h2>User account</h2>
        <ul>
          <li>Username: {this.state.username}</li>
          <li>Age: {this.state.userAge}</li>
        </ul>
      </div>
    );
  }
}
```

En los componentes basados en clases, los valores del state se instancian como propiedades del objeto state dentro del cuerpo de la función del constructor.
A continuación, puede utilizar las variables de estado a través de su código accediendo al objeto this.state.

Modificación del state.

```
class UserAccount extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      username: '',
    };
  }

  render() {
    return (
      <div>
        <input
        type='text'
        value={this.state.username}
        onChange={(event) => this.setState({
          username: event.target.value
        })}
        />
      </div>
    );
  }
}
```

MÁS ↓

Para actualizar el estado basta con llamar al método setState accesible a través de esta palabra clave. Hay que pasarle un objeto con los nuevos valores de las propiedades que deben actualizarse.

Para más información: https://reactjs.org/

State

Modificación del state.

```
const UserAccount = () => {
  const [username, setUsername] = useState('')

  return (
    <div>
      <input
      type='text'
      value={username}
      onChange={(event) =>
        setUsername(event.target.value)}
      />
    </div>
  );
};
```

Para actualizar el valor del state puedes llamar al segundo valor de retorno del hook useState. Necesitas pasarle el nuevo valor del estado deseado como argumento para actualizar el actual.

Operaciones de State complejas.

Arrays.

```
const [users, setUsers] = useState([
  'Tom', 'Mike', 'Anna', 'Amy'
]);

const swapNames = () => {
  let usersCopy = [...users]
  usersCopy[0] = users[3]
  usersCopy[3] = users[0]
  setUsers(usersCopy)
};
```

Puede realizar operaciones complejas en los arrays almacenadas en el state creando una copia del array.

Objetos.

```
const [person, setPerson] = useState({
  name: 'Tom',
  surname: 'Smith',
  age: 22
});

const updateAge = () => {
  setPerson({ ...person, age: person.age - 1 })
};
```

Puede realizar operaciones complejas en los objetos almacenados en estado extendiendo el array y sobrescribiendo las propiedades con los cambios deseados, mientras mantiene los demás cambios almacenados en el state como anteriores.

Datos estáticos (NO ES TÉCNICAMENTE STATE).

Utilizar variables estáticas.

```
const USERNAME = 'mike123';
const UserAccount = () => {
  return (
    <div>
      <h2>Current user: {USERNAME}</h2>
    </div>
  );
};
```

Si tiene datos que no cambiarán con el tiempo, puede definirlos fuera del componente.

Enums.

```
const USER_TYPES = {
  ADMIN: 'Admin',
  BASIC_USER: 'Basic',
  PREMIUM_USER: 'Premium',
  VIP_USER: 'VIP',
};

function App() {
  const [userType, setUserType] = useState(
    USER_TYPES.ADMIN
  )

  return (
    <div>
      <h2>Current user type: {userType}</h2>
    </div>
  );
}
```

Los Enums son una gran manera de añadir más estructura a tu aplicación React. Los Enums se utilizan muy a menudo cuando una variable dada sólo puede tomar un valor que está limitado a un subconjunto. Pueden ayudarnos a reforzar la consistencia para evitar errores en tiempo de ejecución y aumentar la legibilidad del código.

REACT

rortegag

Hooks

useState.

Sintaxis.

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

const App = () => {
  const [<state-variable>, <setter-function>] =
    useState(<default-state>);

  return <div></div>;
}

export default App;
```

Cuando llames al hook useState necesitas pasarle el valor del state por defecto (estado inicial) para la inicialización. useState te devolverá una tupla que contiene la variable del state de sólo lectura y la función setter para actualizar el valor de la variable de estado de sólo lectura.

Ejemplo de uso:

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

const App = () => {
  const [counter, setCounter] = useState(0);

  return (
    <div>
      <h2>{counter}</h2>
        <button onClick={() =>
          setCounter(counter + 1)}>Increment</button>
        <button onClick={() =>
          setCounter(counter - 1)}>Decrement</button>
    </div>
  );
};

export default App;
```

En el ejemplo anterior, estamos definiendo la variable de estado que contiene nuestra variable contador. También estamos usando la función setter setCounter para manejar el incremento y decremento del valor de la variable de estado contador.

useContext.

Crear el context.

```
import React, { createContext } from 'react';
...
export const <context-name> = createContext();

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

Para definir un nuevo contexto necesitas importar createContext desde React como una importación con nombre. Luego necesitas guardar el valor de retorno de llamar a createContext. El siguiente paso es envolver la parte de la aplicación que deseas que sea accesible con el Context Provider. Por último, tienes que pasar el valor del contexto a la propiedad value.

Uso del context:

```
import { useContext } from 'react';

import { <context-name> } from './index';

...
const <context-variable> = useContext(<context-name>);
console.log(<context-variable>);
```

Para utilizar el context, primero tenemos que importar el hook useContext y la propia variable context. Podemos acceder al valor del context pasando el como argumento al useContext. El valor devuelto será el valor del propio context.

MÁS

REACT

rortegag

Hooks

useReducer.

Sintaxis.

```
import { useReducer } from 'react'

const App = () => {
 const [state, dispatch] =
   useReducer(<reducer-function>, <initial-state-object>);

 return <div></div>;
};
```

El hook useReducer toma dos funciones como agumentos; la primera es la función reducer utilizada para modificar el estado actual mantenido por el reducer y la segunda es el objeto de estado inicial. Devuelve la variable de estado que se puede utilizar para acceder a los valores de estado y el envío que desencadena las actualizaciones de estado.

Ejemplo de uso:

```
import { useReducer } from 'react';

const reducer = (state, action) => {
 switch (action.type) {
  case 'increment':
   return {count: state.count + 1};
  case 'decrement':
   return {count: state.count - 1};
 }
};

const App = () => {
 const [state, dispatch] = useReducer(reducer, {count: 0});

 return (
  <>
   <h2>Count: {state.count}</h2>
   <button onClick={() =>
    dispatch({type: 'decrement'})}>Plus one</button>
   <button onClick={() =>
    dispatch({type: 'increment'})}>Minus one</button>
  </>
 );
};
```

En el ejemplo anterior, hemos escrito un ejemplo de contador. Se inicializa con la función reducer y el objeto de estado inicial. Estamos mostrando el valor actual del contador a los usuarios y dejamos que lo cambien interactuando con los dos botones.
Están llamando al método dispatch con el objeto de acción correspondiente. La función reducer lee los valores del objeto de acción y devuelve el nuevo estado.

MÁS

useEffect.

Sintaxis.

```
import { useEffect } from 'react';

const App = () => {
 useEffect(<callback-function>, <dependency-array>);

 return <div></div>;
}
```

El hook useEffect toma 2 argumentos, el primero es una función callback y el otro es el array de dependencias. La función callback se ejecuta cada vez que React detecta una actualización de un elemento (ref o cualquier otro elemento de estado) proporcionado en el array de dependencias.

Ejemplo de uso 1:

```
import axios from 'axios';
import { useEffect } from 'react';

const App = () => {
 useEffect(() => {
  const fetchData = async () => {
   ...
  };

  fetchData()
 }, []);

 return <div></div>;
};
```

Estamos llamando a la función fetchData que es responsable de obtener los datos del servidor.
La ejecutamos cada vez que se vuelve a renderizar porque el array de dependencias está vacía.

Ejemplo de uso 2:

```
import axios from 'axios';
import { useEffect } from 'react';

const App = () => {
 const [userID, setUserID] = useState(23112);

 useEffect(() => {
  const fetchData = async (userID) => {
   ...
  };

  fetchData(userID)
 }, [userID]);

 return <div></div>;
}
```

Estamos llamando a la función fetchData que es responsable de obtener datos del servidor.
La ejecutamos sólo cuando hubo un cambio en la variable del state userID.

REACT

rortegag

Hooks

useCallback.

Sintaxis.

```
import { useCallback } from 'react';

const App = () => {
  const <memoized-callback> =
    useCallback(<callback-function>, <dependency-array>)

  return <div></div>;
};
```

El hook useCallback toma dos argumentos: el primero es la función callback y el segundo es el array de dependencias. useCallback activará la ejecución de su argumento callback cada vez que cambie alguno de los elementos del array de dependencias.

Ejemplo de uso:

```
import { useCallback } from 'react';

const App = () => {
  const [userID, setUserID] = useState(2423);
  const computeUserScore = () => {
  ...
  }

  const memoizedCallback =
    useCallback(() => computeUserScore(), [userID]);
  ...

};
```

Hemos escrito un componente sencillo que calcula la puntuación del usuario y la devuelve. memoizedCallback se actualizará cuando cambie el userID.

useRef.

Sintaxis.

```
import { useRef } from 'react';

const App = () => {
  const <ref-variable> = useRef();

  return <div ref={<ref-variable>} />;
};
```

Para crear una nueva referencia necesitas importar el hook useRef de React. Luego tienes que llamar al hook y guardar el valor de retorno como una variable. Finalmente tendrías que pasar el ref prop en el elemento JSX deseado.

Ejemplo de uso:

```
import { useRef } from 'react';

const App = () => {
  const ref = useRef();

  const onClickHandler = () => {
    ref.current.focus();
  };

  return (
    <>
      <input ref={ref} type="text" />
      <button onClick={onClickHandler}>Focus</button>
    </>
  );
}
```

En el ejemplo anterior, hemos definido una referencia y la hemos pasado al elemento input. También hemos creado un elemento botón. Estamos utilizando la referencia para llamar al método focus en él cada vez que hacemos clic en el botón.

useMemo.

Sintaxis.

```
import { useMemo } from 'react';

const App = () => {
  const <memoized–value> =
    useMemo(<callback-function>, <dependency-array>);

  return <div></div>;
};
```

useMemo toma dos argumentos, el primero es la función callback que calcula el valor deseado y el segundo es el array de dependencia. Funciona de forma similar al hook useEffect. useMemo recalculará el valor memoizado cada vez que alguno de los elementos pasados al array de dependencias sufra algún cambio.

MÁS ↓

Uso de ejemplo:

```
import { useMemo } from 'react';

const App = () => {
  const [userID, setUserID] = useState(2423);
  const computeUserScore = () => {
  ...
  };

  const userScore = useMemo(() =>
    computeUserScore(), [userID]);

  return <div>{userScore}</div>;
};
```

Un componente sencillo que calcula la puntuación del usuario y la devuelve. userScore se calcula mediante la función computeUserScore, que está memoizada. La puntuación se vuelve a calcular sólo si hay un cambio en la variable de estado userID.

REACT

rortegag

Hooks

useLayoutEffect.

Sintaxis.

```
import { useLayoutEffect } from 'react';

const App = () => {
  useLayoutEffect(<callback-function>,
  <dependency-array>);

  return <div></div>;
};
```

El hook useLayoutEffect toma 2 argumentos; el primero es una función callback y el otro es el array de dependencias. La función callback se ejecuta después de que todas las mutaciones necesarias se hayan aplicado al DOM.

useDeferredValue.

Sintaxis.

```
import { useDeferredValue } from 'react';

const App = ({ value }) => {
  const <deferred-value> = useDeferredValue(value);
  ...
};
```

El hook useDeferredValue acepta un valor como argumento y devuelve un valor diferido.

Ejemplo de uso:

```
import { useDeferredValue } from 'react';

const App = ({ value }) => {
  const deferredValue = useDeferredValue(value);
  return <div>{deferredValue}</div>;
};
```

useId.

Sintaxis.

```
import { useId } from 'react';
...
const id = useId();
```

Después de importar el hook useId de React puedes simplemente asignar el valor devuelto al llamar al hook a una variable a la que se le asignará un valor de id único generado.

Ejemplo de uso:

```
import { useId } from 'react';

const App = () => {
  const id = useId();
  return (
    <>
      <button id={id}>Hello</button>
    </>
  );
};
```

Generamos un nuevo ID y se lo pasamos al JSX.

useImperativeHandle.

Sintaxis.

```
import { useImperativeHandle } from 'react';

const App = () => {
  const inputRef = useRef();

  useImperativeHandle(ref, () => ({
    <property>: () => {
    ...
    }
  }));

  return <input ref={inputRef} />;
};
```

El hook useImperativeHandle nos da control sobre los valores expuestos al componente padre a través del ref handle. Desde el componente padre se puede llamar a inputRef.current().

useDebugValue.

Sintaxis.

```
import { useDebugValue } from 'react';
...
useDebugValue(<label>);
```

El gancho useDebugValue toma una etiqueta que se mostrará al depurar ganchos personalizados.

Ejemplo de uso:

```
import { useDebugValue } from 'react';

const useUserStatus = () => {
  const [premiumUser, setPremiumUser] = useState(true);
  useDebugValue(premiumUser ?
    'premium user' : 'basic user');

  return premiumUser;
};
```

En el hook personalizado useUserStatus estamos usando el hook useDebugValue para mostrar el estado relevante del usuario.

MÁS ↓

REACT

rortegag

Hooks

useTransition.

Sintaxis.

```
import { useTransition } from 'react';

function App() {
  const [<value>, <function>] = useTransition();
  ...
}
```

El hook useTransition devuelve una tupla que contiene un valor que indica si la transición se está ejecutando, así como una función para iniciar la transición.

Ejemplo de uso:

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

function App() {
  const [loading, startTransition] = useTransition();
  const [count, setCount] = useState(0);
  const handleClick = () => {
    startTransition(() => {
      setCount(count + 1);
    })
}

  return (
    <div>
      <h2>{count}</h2>
        <button onClick={handleClick}>
          Increment
        </button>
        {loading && <Loader />}
    </div>
  );
}
```

En el ejemplo del contador anterior estamos utilizando el valor de carga para indicar a los usuarios si se está produciendo una actualización. También estamos utilizando la función startTransaction para actualizar el estado en cada clic del botón.

componentDidUpdate.

```
componentDidMount(prevProps) {
  if (this.props.ID !== prevProps.ID) {
    axios.get(testEndpoint)
      .then(resp => this.setState({ testData: resp }));
  }
}
```

Este método se llama justo después de que el componente se vuelva a renderizar. Normalmente se utiliza para manipular el DOM o para realizar peticiones más eficientes a otras aplicaciones.

MÁS

↓

Métodos de ciclo de vida

constructor.

```
constructor(props) {
  super(props);
  this.state = {
    portalName: 'Upmostly'
  };

  this.handleLogin = this.handleLogin.bind(this);
}
```

Un constructor en React se utiliza para inicializar el estado y enlazar métodos dentro del componente de la clase.

render.

```
render() {
  return
  <h1>Welcome to
    {this.state.portalName}
  </h1>;
}
```

Render es un método requerido que tiende a devolver JSX.

componentDidMount.

```
componentDidMount() {
  axios.get(testEndpoint).then(resp =>
    this.setState({ testData: resp }));
}
```

Este método se ejecuta cuando el componente se monta por primera vez en el DOM. Normalmente se utiliza para obtener datos de otras aplicaciones.

componentWillUnmount.

```
componentWillUnmount() {
  document.removeEventListener("click",
    this.closeMenu);
}
```

Esto es lo que llamamos un método de "limpieza". Puedes tratarlo como lo opuesto al componentDidMount. Es el lugar en el que debes cancelar cualquier tarea que hayas podido inicializar cuando el componente se estaba montando. `componentWillUnmount` es llamado cuando el componente es removido del DOM.

shouldComponentUpdate.

```
shouldComponentUpdate(nextProps) {
  if (nextProps.portalName !== this.props.portalName)
    return true
  return false;
}
```

Este método se utiliza como algo que puede ayudar a mejorar el rendimiento de nuestros componentes. Es llamado antes de que el componente sea re-renderizado con nuevos props o estado.

REACT

rortegag

Métodos de ciclo de vida

getSnapshotBeforeUpdate.

```
getSnapshotBeforeUpdate() {
  return {
    tableBackground: this.tableRef.current.offsetHeight >= 85
  };
}
```

Este método se ejecuta poco antes de que la salida del componente se anexe al DOM real. Nos ayuda a capturar cierta información que puede ser útil al llamar a componentDidUpdate.

static getDerivedStateFromProps.

```
static getDerivedStateFromProps(props, state) {
  if(props.id !== state.id) {
    return {
      id: props.id
    }
  }

  return null;
}
```

Este método es llamado cada vez justo antes del método de renderizado.

PropTypes

Sintaxis.

```
import PropTypes from 'prop-types';
...
ExampleComponent.propTypes = {
  <prop-name>: PropTypes.<type>, ...
};
```

Tienes que importar PropTypes del paquete prop-types. A continuación, puede definir los tipos deseados escribiéndolos en una notación de objeto y asignándolos a la propiedad propTypes de su componente.

Ejemplo de uso 1:

```
import PropTypes from 'prop-types';

const User = ({ name, surname, age }) => {
  return (
    <div>
      <p>Name: {name}</p>
      <p>Surname: {surname}</p>
      <p>Age: {age}</p>
    </div>
  );
};
User.propTypes = {
  name: PropTypes.string,
  surname: PropTypes.string,
  age: PropTypes.number
};

export default User
```

Definimos nombre y apellidos como de tipo cadena y edad como de tipo número.

Ejemplo de uso 2:

```
import PropTypes from 'prop-types';

const App = ({ name, age, rightHanded }) => {
  return (
    <div>
      <p>Name: {name.firstName}</p>
      <p>Surname: {name.lastName}</p>
      <p>Age: {age}</p>
      <p>Dominant hand:
          {rightHanded ? 'right' : 'left'}
      </p>
    </div>
  );
};
App.propTypes = {
  name: PropTypes.shape({
    firstName: PropTypes.string,
    lastName: PropTypes.string
  }),
  age: PropTypes.number,
  rightHanded: PropTypes.oneOf([true, false]),
};

export default App;
```

También puedes gestionar props shapes más complejas utilizando PropTypes.

JSX

Return.

```
return (
  <div>Hello from Upmostly!</div>
);
```

Los componentes React están pensados para devolver un único elemento JSX de nivel superior.

MÁS ↓

Para más información: https://reactjs.org/

REACT

rortegag

JSX

Las funciones pueden utilizarse dentro de JSX y devolver JSX.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}

const App = () => {
  const user = {
    firstName: 'Mike',
    lastName: 'Smith',
  };
  return (
    <div>
      {getGreeting(user)}
    </div>
  );
};
```

JSX puede contener expresiones JavaScript, como llamadas a funciones. Las llamadas a funciones pueden devolver JSX que se renderizará en la pantalla.

Nesting.

```
return (
  <div>
    <h1>Hello from Upmostly!</h1>
    <button>Click me</button>
    <ul>
      <li>1</li>
      <li>2</li>
      <li>3</li>
    </ul>
  </div>
);
```

Los componentes React están pensados para devolver un único elemento JSX de nivel superior. Ese elemento puede tener más elementos anidados en su interior.

```
const App = () => {
  return (
    <div className="element-with-class-in-jsx" />
  );
};
```

En JSX, los elementos ya no pueden utilizar la palabra clave "class", ya que está restringida a JavaScript. En su lugar, utilizamos la palabra clave "className", que hace lo mismo que la palabra clave "class" en HTML.

Eventos en elementos JSX

onClick.

Actualizar state.

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

const App = () => {
  const [userName, setUserName] = useState('Tommy')
  const [showName, setShowName] = useState(false)

  return (
    <div>
      {showName && <h2>{userName}</h2>}
      <button onClick={() => setShowName(!showName)}>
        Show user name
      </button>
    </div>
  );
};
```

Puede utilizar el controlador onClick para actualizar el estado.

onSubmit.

Envío del formulario.

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

const App = () => {
  const [userInput, setUserInput] = useState('');
  const submitHandler = (event) => {
    event.preventDefault();
    console.log(userInput);
  }

  return(
    <div>
      <form onSubmit={submitHandler}>
        <input value={userInput} onChange={(event) =>
          setUserInput(event.target.value)}/>
        <button type='submit'>Submit</button>
      </form>
    </div>
  );
};
```

Puede utilizar el controlador onSubmit para enviar sus formularios. onSubmit está disponible en la etiqueta de formulario que tiene anidado un botón de tipo submit.

MÁS

Para más información: https://reactjs.org/

REACT

rortegag

Eventos en elementos JSX

onChange.

Entradas controladas.

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

const App = () => {
  const [userInput, setUserInput] = useState('');

  return (
    <div>
      <input value={userInput} onChange={(event) =>
        setUserInput(event.target.value)}/>
    </div>
  );
};
```

Puede utilizar el manejador onChange para controlar los datos de sus entradas.

Formulario abreviado del controlador de eventos.

Entradas controladas.

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

const App = () => {
  const [userInput, setUserInput] = useState('');
  const handleInputValueChange = (e) => {
    setUserInput(e.target.value);
  };

  return (
    <div>
      <input
      value={userInput}
      onChange={handleInputValueChange}
      />
    </div>
  );
};
```

Si la función del manejador de eventos que has definido sólo toma un parámetro de evento, puedes simplemente pasarlo a la prop del manejador de eventos y el objeto del evento será inferido.

Trucos

Patrones útiles de React.

Arrays de renderización.

```
const FIBONACCI = [1, 1, 2, 3, 5, 8];

const App = () => {
  return (
    <ul>
      {FIBONACCI.map((number) =>
        <li key={number}>{number}</li> )}
    </ul>
  );
}
```

Puedes renderizar arrays en React eficientemente usando el método map. Puedes mapear cada uno de los elementos del array en su elemento JSX correspondiente y renderizarlo directamente desde el JSX.

Pluralizar.

```
const pluralize = (number, singular, plural) => {
  if (number === 1) {
    return `${number} ${singular}`
  }

  return `${number} ${plural}`;
};
```

Sus cadenas pueden a veces depender de un valor numérico para mostrar una forma relevante del sustantivo. Puedes escribir un método pluralize que se encargue de ello automáticamente.

Fallback de array vacío.

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

const App = () => {
  const [users, setUsers] = useState([1]);

  return (
    <div>
      {
        users.length === 0
        ?
        <h2>There are no users :(</h2>
        :
        <h2>There are {users.length} users :)</h2>
      }
    </div>
  );
}
```

Si su aplicación depende de los datos almacenados en el array, puede proporcionar una alternativa en caso de que el array esté vacío.

Para más información: https://reactjs.org/

REACT

rortegag

Sintaxis útil

Operador condicional.

Sintaxis.

```
<condition> ? <if-true> : <if-false>
```

Los ternarios son una notación abreviada para el flujo de control con 2 condiciones. Si la condición se evalúa como verdadera, el bloque se ejecutará; de lo contrario, se ejecutará.

Ejemplo de uso:

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

const App = () => {
  const [age, setAge] = useState(19)

  return (
    <div>
      <h2>{`User is ${age > 18 ? 'over' : 'under'} 18`}</h2>
    </div>
  );
}
```

Usamos el operador ternario para mostrar la cadena correcta. Si el usuario es mayor de 18 años, se mostrará "El usuario es mayor de 18 años".

Encadenamiento condicional.

Sintaxis.

```
<object-name>?.[<property-name>]
```

Si existe, se accederá a él; de lo contrario, la expresión se evaluará como indefinida.

Ejemplo de uso:

```
const user = {
  personalDetails: {
    name: 'Tommy',
    surname: 'Smith',
    age: 21
  },
  subscriptionDetails: {
    type: 'premium',
    price: 12.99
  }
};

user?.subscriptionDetails?.type // premium

user?.subscriptionDetails?.age  // undefined
```

La primera expresión evalúa a la cadena 'premium' mientras que la otra a indefinido.

MÁS

Evaluación de cortocircuitos.

&&

```
import { useState } from 'react';
import MyComponent from './component';

const App = () => {
  const [age, setAge] = useState(19);

  return (
    <div>
      {age > 18 && <MyComponent />}
    </div>
  );
}
```

El operador && se utiliza a menudo para mostrar componentes si se cumple una condición dada. Si la condición no se cumple, se evaluará y devolverá null.

||

```
import MyComponent from './component';

const user = {
  personalDetails: {
    name: 'Tommy',
    surname: 'Smith',
    age: 21
  },
  subscriptionDetails: {
    type: 'premium',
    price: 12.99
  }
};

const App = () => {
  return (
    <div>
      {user?.userDetails || ‘Fallback value’}
    </div>
  );
};
```

El operador || se utiliza a menudo para mostrar un valor alternativo cuando la condición se evalúa como falsa. Si la condición es falsa, entonces se mostrará 'Valor alternativo'.

React 18.

ReactDOM.

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

const rootElement = document.getElementById('root');
const root = ReactDOM.createRoot(rootElement);

root.render(<App />);
```

El fragmento de código anterior te ayudará a adaptar tu código a los cambios de React 18.

Para más información: https://reactjs.org/

REACT

rortegag

Sintaxis útil

Profiler.

Sintaxis.

```
import { Profiler } from 'react';

function App() {
  return (
    <Profiler id=<string-id> onRender={<log-function>}>
    ...
    </Profiler>
  );
}
```

Para utilizar el Profiler necesitamos importarlo del paquete react. El siguiente paso es envolver la parte del árbol JSX que te gustaría que analizara.
Luego tienes que pasar el que se usa para identificar el Profiler y el que procesará tu log.

Ejemplo de uso:

```
import { Profiler } from 'react';
import Counter from './Counter'

const log = (id, phase, actualTime, baseTime, startTime,
                commitTime, interactions) => {
  console.table({ id, phase, actualTime, baseTime, startTime,
                commitTime, interactions });
};

function App() {
  return (
    <div style={{ margin: '50px' }}>
      <Profiler id="Counter" onRender={log}>
        <Counter />
      </Profiler>
    </div>
  );
}

export default App;
```

En este ejemplo, estamos utilizando el Profiler para inspeccionar los datos sobre nuestro componente Counter personalizado. A continuación, estamos registrando los datos en forma de una tabla para que sea más fácil de entender.

Estilos en línea computados.

Ejemplo de uso:

```
const App = ({ headerFontSize, headerColor }) => {
  const styles = {
    fontSize: headerFontSize,
    color: headerColor,
  };

  return <h2 style={styles}>Hello from Upmostly!</h2>;
};
```

MÁS ↓

Fragments.

Sintaxis.

```
import React from 'react';

const Numbers = () => {
  return (
    <React.Fragment>
      <li>1</li>
      <li>2</li>
      <li>3</li>
    </React.Fragment>
  )
}
```

Envuelve los hijos JSX dentro del componente Fragment disponible en el paquete React import from React por defecto.

Sintaxis abreviada.

```
const Numbers = () => {
  return (
    <>
      <li>1</li>
      <li>2</li>
      <li>3</li>
    </>
  );
}
```

Styling

En línea.

Ejemplo de uso:

```
const App = () => {
  return <h2 style={{ fontSize: '10px', color: 'yellow' }}>
    Hello from Upmostly!
  </h2>;
};
```

Puede utilizar el estilo en línea proporcionando el elemento JSX con la propiedad style. Acepta un objeto cuyas claves son propiedades CSS camelCased correspondientes a valores proporcionados como cadenas.

CSS.

Ejemplo de uso:

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

const App = () => {
  return <h2 className='header'>
    Hello from Upmostly!
  </h2>;
};
```

Puede utilizar el CSS importando el archivo CSS correspondiente en su componente.
A continuación, puede añadir las clases pertinentes utilizando la propiedad className en el elemento JSX correspondiente.

Para más información: https://reactjs.org/

REACT

rortegag

Styling

Styled-components.

Sintaxis.

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

const <styled-component-name> =
  styled.<element>`<css-properties>`;
```

Para utilizar componentes con estilo, primero tiene que importar styled del paquete styled-components. Puede definir el nuevo componente con estilo utilizando la sintaxis styled. Puede proporcionar las propiedades deseadas entre " como lo haría en cualquier otro archivo CSS.

Ejemplo de uso:

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

const Heading = styled.h2`font-size: 10px;`;

const App = () => {
  return <Heading>Hello from Upmostly!</Heading>;
};
```

El fragmento de código anterior define un nuevo componente con estilo llamado Heading. A continuación, utiliza el componente con estilo como cualquier otro componente en el JSX.

Módulos CSS.

Sintaxis.

```
import <styles-object> from './<name>.module.css';
...
.heading {
  font-size: 10px;
}
```

Puedes utilizar los módulos CSS nombrando tus archivos CSS con el formato .module.css. Puedes definir tus estilos como lo harías en un archivo CSS normal.

Ejemplo de uso:

```
import styles from './App.module.css';

const App = () => {
  return <h2 className={styles.heading}>
    Hello from Upmostly!
  </h2>;
};
```

Puede utilizar los módulos CSS importando los estilos del archivo correspondiente y pasando el estilo deseado a la propiedad className del elemento JSX correspondiente.

ReactDOM API

Strict Mode.

Sintaxis.

```
import React, { StrictMode } from 'react';
import ReactDOM from 'react-dom';

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

Uso: Importa StrictMode de React y luego envuelve la parte de la aplicación donde quieras usarlo.