

<https://es.react.dev/learn/tutorial-tic-tac-toe>

Tutorial: Tres en línea

En este tutorial construirás un pequeño juego de Tres en línea. Este tutorial no asume ningún conocimiento previo de React. Las técnicas que aprenderás en el tutorial son fundamentales para crear cualquier aplicación de React, y comprenderlas por completo te dará una comprensión profunda de React.

Nota

Este tutorial fue diseñado para personas que prefieren aprender haciendo y quieren ver algo tangible de manera rápida. Si prefieres aprender cada concepto paso a paso, comienza con Describir la UI.

El tutorial se divide en varias secciones:

Configuración para el tutorial te dará un punto de partida para seguir el tutorial.

Descripción general te enseñará los fundamentos de React: componentes, props y estado.

Completar el juego te enseñará las técnicas más comunes en el desarrollo de React.

Agregar viajes en el tiempo te brindará una visión más profunda de las fortalezas únicas de React.

¿Qué estás construyendo?

En este tutorial, crearás un juego interactivo de Tres en línea con React.

Puedes ver cómo se verá cuando hayas terminado aquí:

App.js

Descargar

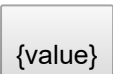
Reiniciar

Bifurcar

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21

22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

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

```
function Square({ value, onSquareClick }) {  
  return (  
    {value}  
  );  
}
```

```
function Board({ xIsNext, squares, onPlay }) {  
  function handleClick(i) {  
    if (calculateWinner(squares) || squares[i]) {  
      return;  
    }  
    const nextSquares = squares.slice();  
    if (xIsNext) {  
      nextSquares[i] = 'X';  
    } else {  
      nextSquares[i] = 'O';  
    }  
    onPlay(nextSquares);  
  }  
  
  const winner = calculateWinner(squares);  
  let status;  
  if (winner) {  
    status = 'Ganador: ' + winner;  
  } else {  
    status = 'Siguiente jugador: ' + (xIsNext ? 'X' : 'O');  
  }  
}
```

```
return (  
<>  
  
{status}
```

Mostrar más

Si el código aún no tiene sentido para ti, o si no estás familiarizado con la sintaxis del código, ¡no te preocupes! El objetivo de este tutorial es ayudarte a comprender React y su sintaxis.

Te recomendamos que consultes el juego de Tres en línea anterior antes de continuar con el tutorial. Una de las características que notarás es que hay una lista numerada a la derecha del tablero del juego. Esta lista te brinda un historial de todos los movimientos que se han producido en el juego y se actualiza a medida que avanza el juego.

Una vez que hayas jugado un poco con el juego Tres en línea terminado, sigue desplazándote. Comenzarás con una plantilla más simple en este tutorial. Nuestro siguiente paso es prepararte para que puedas comenzar a construir el juego.

Configuración para el tutorial

En el editor de código en vivo a continuación, haz clic en Bifurcar en la esquina superior derecha para abrir el editor en una nueva pestaña usando el sitio web CodeSandbox. CodeSandbox te permite escribir código en su navegador e inmediatamente ver cómo sus usuarios verán la aplicación que ha creado. La nueva pestaña debería mostrarte un cuadrado vacío y el código de inicio para este tutorial.

App.js

Descargar

Reiniciar

Bifurcar

```
1  
2  
3  
4  
export default function Square() {  
  return ;  
}
```

Nota

También puedes seguir este tutorial utilizando tu entorno de desarrollo local. Para hacer esto, necesitas:

Instalar Node.js

En la pestaña CodeSandbox que abriste anteriormente, presiona el botón de la esquina superior izquierda para abrir el menú y luego selecciona Descargar Sandbox en ese menú para descargar un archivo comprimido de los archivos que necesitaras para realizar este tutorial.

Descomprime el archivo, luego abre la terminal e introduce el comando `cd` en el directorio que descomprimiste

Instala las dependencias con el comando `npm install`

Ejecuta el comando `npm start` para iniciar un servidor local y sigue las indicaciones para ver el código que se ejecuta en un navegador

Si te quedas atascado, ¡no dejes que esto te detenga! Siga en línea en su lugar e intenta una configuración local nuevamente más tarde.

Descripción general

Ahora que está configurado, veamos una descripción general de React.

Inspeccionar el código de inicio

En CodeSandbox verás tres secciones principales:


CodeSandbox con código de inicio

La sección Files con una lista de archivos como App.js, index.js, styles.css y una carpeta llamada public

El code editor donde verás el código fuente de tu archivo seleccionado

La sección browser donde verás cómo se mostrará el código que has escrito

El archivo App.js debe seleccionarse en la sección Files. El contenido de ese archivo en el code editor debería ser:

```
export default function Square() {  
  return ;  
}
```


La sección del browser debería mostrarte un cuadrado con una X como esta:

Cuadrado lleno de x


Ahora echemos un vistazo a los archivos en el código de inicio.

App.js

El código en App.js crea un component. En React, un componente es una pieza de código reutilizable que representa una parte de una interfaz de usuario. Los componentes se utilizan para representar, administrar y actualizar los elementos de la interfaz de usuario en su aplicación. Miremos el componente línea por línea para ver qué está pasando:

```
export default function Square() {  
  return ;  
}
```

La primera línea define una función llamada Square. La palabra clave de JavaScript export hace que esta función sea accesible fuera de este archivo. La palabra clave default le dice a otros archivos que usan su código que es la función principal en su archivo.

```
export default function Square() {  
  return ;  
}
```

La segunda línea devuelve un botón. La palabra clave de JavaScript return significa que lo que viene después se devuelve como un valor a la persona que llama a la función.

es un elemento JSX. Un elemento JSX es una combinación de código JavaScript y etiquetas HTML que describe lo que te gustaría mostrar. className="square" es una propiedad de botón o prop que le dice a CSS cómo diseñar el botón. X es el texto que se muestra dentro del botón y

cierra el elemento JSX para indicar que ningún contenido siguiente debe colocarse dentro del botón.

styles.css

Haz clic en el archivo llamado styles.css en la sección Files de CodeSandbox. Este archivo define los estilos

para tu aplicación React. Los primeros dos selectores CSS (* y body) definen el estilo de grandes partes de su aplicación, mientras que el selector .square define el estilo de cualquier componente donde la propiedad className está establecida en square. En tu código, eso coincidiría con el botón de tu componente Square en el archivo App.js.

index.js

Haz clic en el archivo llamado index.js en la sección Files de CodeSandbox. No editarás este archivo durante el tutorial, pero es este el puente entre el componente que creaste en el archivo App.js y el navegador web.

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import './styles.css';
```

```
import App from './App';
```

Las líneas 1-5 reúnen todas las piezas necesarias:

React

Biblioteca de React para hablar con los navegadores web (React DOM)

los estilos para tus componentes

el componente que creaste en App.js.

El resto del archivo reúne todas las piezas e inyecta el producto final en index.html en la carpeta public.

Construir el tablero

Volvamos a App.js. Aquí es donde pasarás el resto del tutorial.

Actualmente, el tablero es solo un cuadrado, ¡pero necesitas nueve! Si solo intentas copiar y pegar tu cuadrado para hacer dos cuadrados como este:

```
export default function Square() {
  return ☐ ☐;
}
```

Obtendrás este error:

Console

```
/src/App.js: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment
<>...</>
```

(Traducción)

/src/App.js: Los elementos JSX adyacentes deben estar envueltos en una etiqueta contenedora. ¿Querías un Fragmento JSX <>...</>?

Los componentes de React deben devolver un solo elemento JSX y no múltiples elementos JSX adyacentes como dos botones. Para solucionar esto, puedes usar Fragmentos (<> y </>) para envolver múltiples elementos JSX adyacentes como este:

```
export default function Square() {
  return (
```

```
<>
```

```
☐
☐
```

```
</>
```

```
);
```

```
}
```

Ahora deberías ver:

dos cuadrados llenos de x

¡Excelente! Ahora solo necesitas copiar y pegar varias veces para agregar nueve cuadrados y...

nueve cuadrados llenos de x en una línea

¡Oh, no! Los cuadrados están todos en una sola línea, no en una cuadrícula como la que necesitamos para nuestro tablero. Para solucionar esto, deberás agrupar tus cuadrados en filas con divs y agregar algunas clases de CSS. Mientras lo haces, le darás a cada cuadrado un número para asegurarse de saber dónde se muestra cada cuadrado.

En el archivo App.js, actualiza el componente Square para que se vea así:

```
export default function Square() {
```

```
  return (
```

```
<>
```

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

</>); } El CSS definido en styles.css diseña los divs con className de board-row. Ahora que agrupaste tus componentes en filas con el estilo divs, tienes un tablero de tres en línea:

El tablero de Tres en línea esta lleno con números del 1 al 9

Pero ahora tienes un problema. Tu componente llamado Square, en realidad ya no es un cuadrado.

Arreglemos esto cambiando el nombre a Board:

```
export default function Board() {
```

```
//...
```

```
}
```

En este punto, tu código debería verse así:

App.js

Descargar

Reiniciar

Bifurcar

1

2

3

4

5

6

7

8

9

10
11
12
13
14
15
16
17
18
19
20
21
22

```
export default function Board() {  
  return (  
    <>
```

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

```
</> ); }
```

Mostrar más

Nota

Pssst... ¡Eso es mucho para escribir! Está bien copiar y pegar el código de esta página. Sin embargo, si estás preparado para un pequeño desafío, te recomendamos que solo copies el código que haz escrito manualmente al menos una vez.

Pasar datos a través de props

A continuación, querrás cambiar el valor de un cuadrado de vacío a «X» cuando el usuario haga clic en el cuadrado. Con la forma en que ha construido el tablero hasta ahora, necesitarías copiar y pegar el código que actualiza el cuadrado nueve veces (¡una vez por cada cuadrado que tengas)! En lugar de copiar y pegar, la arquitectura de componentes de React te permite crear un componente reutilizable para evitar el código duplicado desordenado.

Primero, ve a copiar la línea que define el primer cuadrado () de tu componente Board en un nuevo componente Square:

```
function Square() {  
  return ;  
}
```

```
export default function Board() {  
  // ...  
}
```

Luego, actualiza el componente Board para renderizar ese componente Square usando la sintaxis JSX:

```
// ...
```

```
export default function Board() {  
  return (  
    <>
```

```
</> ); } Observa cómo, a diferencia de los divs del navegador, tus propios componentes Board y Square  
deben comenzar con una letra mayúscula.
```

Vamos a ver:

tablero lleno

¡Oh, no! Perdiste los cuadrados numerados que tenías antes. Ahora cada cuadrado dice «1». Para arreglar esto, utiliza las props para pasar el valor que debe tener cada cuadrado del componente principal (Board) al componente secundario (Square).

Actualiza el componente Square para leer la propiedad value que pasarás desde el Tablero:

```
function Square({ value }) {  
  return 1;  
}
```

function Square({ value }) indica que al componente Square se le puede pasar un objeto llamado value.

Ahora deseas mostrar ese value en lugar de 1 dentro de cada cuadrado. Intenta hacerlo así:

```
function Square({ value }) {  
  return value;  
}
```

Vaya, esto no es lo que querías:

tablero lleno de valores

Querías representar la variable de JavaScript llamada value de tu componente, no la palabra «valor». Para «escapar a JavaScript» desde JSX, necesitas llaves. Agrega llaves alrededor de value en JSX de esta manera:

```
function Square({ value }) {  
  return {value};  
}
```

Por ahora, deberías ver un tablero vacío:

tablero vacío

Esto se debe a que el componente Board aún no ha pasado la prop value a cada componente Square que representa. Para solucionarlo, agrega el complemento value a cada componente Square representado por el componente Board:

```
export default function Board() {  
  return (  
    <>
```

```
</> ); } Ahora deberías ver una cuadrícula de números nuevamente:
```


tablero de Tres en línea lleno de números del 1 al 9

Tu código actualizado debería verse así:

App.js

Descargar

Reiniciar

Bifurcar

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

```
function Square({ value }) {  
  return {value};  
}
```

```
export default function Board() {  
  return (  
<>  
  
</> ); }  

```

Mostrar más

Hacer un componente interactivo.

Rellenemos el componente Square con una X al hacer clic en él. Declara una función llamada handleClick

dentro del Square. Luego, agrega onClick a las props del elemento

```
JSX devuelto por el componente Square:
function Square({ value }) {
  function handleClick() {
    console.log('¡hiciste clic!');
  }

  return (
    <div>
      <div>{value}</div>
      <button onClick={handleClick}>Click</button>
    </div>
  );
}
```

Si haces clic en un cuadrado ahora, deberías ver un registro que dice "¡hiciste clic!" en la pestaña Consola en la parte inferior de la sección Navegador de CodeSandbox. Al hacer clic en el cuadrado más de una vez, se registrará "¡hiciste clic!" de nuevo. Los registros repetidos en la consola con el mismo mensaje no crearán más líneas en la consola. En su lugar, verás un contador incremental al lado de su primer registro "¡hiciste clic!".

Nota

Si estás siguiendo este tutorial utilizando tu entorno de desarrollo local, debes abrir la consola de tu navegador. Por ejemplo, si usas el navegador Chrome, puedes ver la Consola con el método abreviado de teclado Shift + Ctrl + J (en Windows/Linux) u Option + ⌘ + J (en macOS).

Como siguiente paso, deseas que el componente Square «recuerde» que hiciste clic y lo rellene con una marca «X». Para «recordar» cosas, los componentes usan estado.

React proporciona una función especial llamada useState que puedes llamar desde tu componente para permitirle «recordar» cosas. Almacenemos el valor actual del Square en el estado, y cambiémoslo cuando se haga clic en Square.

Importa useState en la parte superior del archivo. Elimina la propiedad value del componente Square. En su lugar, agrega una nueva línea al comienzo del componente Square que llame a useState. Haz que este devuelva una variable de estado llamada value:

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

function Square() {
  const [value, setValue] = useState(null);

  function handleClick() {
    //...
  }

  value almacena el valor y setValue es una función que se puede usar para cambiar el valor. El null pasado a
  useState se usa como valor inicial para esta variable de estado, por lo que value aquí comienza igual a
  null.
```

Dado que el componente Square ya no acepta props, elimina la prop value de los nueve componentes Square creados por el componente Board:

```
// ...
export default function Board() {
```

```
return (
```

```
<>
```

`</>); }` Ahora cambia Square para mostrar una «X» cuando se haga clic. Reemplaza el controlador de evento `console.log("¡hiciste clic!");` con `setValue('X');`. Ahora tu componente Square se ve así:

```
function Square() {  
  const [value, setValue] = useState(null);
```

```
  function handleClick() {  
    setValue('X');  
  }
```

```
  return (
```

```
     {value}
```

```
  );
```

```
}
```

Al llamar a esta función `set` desde un controlador `onClick`, le estás diciendo a React que vuelva a renderizar ese Square cada vez que se hagas clic en

. Después de la actualización, el value del Square será 'X', por lo que verás la «X» en el tablero de juego.

Si haces clic en cualquier cuadrado, debería aparecer una «X»:

adicionando x al tablero

Ten en cuenta que cada Square tiene su propio estado: el value almacenado en cada Square es completamente independiente de los demás. Cuando llamas a una función set en un componente, React también actualiza automáticamente los componentes secundarios dentro de él.

Después de realizar los cambios anteriores, tu código se verá así:

App.js
Descargar
Reiniciar

Bifurcar

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

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

function Square() {
  const [value, setValue] = useState(null);

  function handleClick() {
    setValue('X');
  }

  return (
```

{value}

```
);  
}  
  
export default function Board() {  
  return (  
<>
```

Mostrar más

React Developer Tools

React DevTools te permite verificar las props y el estado de tus componentes React. Puedes encontrar la pestaña React DevTools en la parte inferior de la sección navegador en CodeSandbox:

React DevTools en CodeSandbox

Para inspeccionar un componente en particular en la pantalla, usa el botón en la esquina superior izquierda de React DevTools:

Seleccionar componentes en la página con React DevTools

Nota

Para el desarrollo local, React DevTools está disponible como Chrome, Firefox, y Edge extensión del navegador. Después de instalarlo, la pestaña Componentes aparecerá en las Herramientas de desarrollo de tu navegador para los sitios que utilizan React.

Completar el juego

En este punto, tienes todos los componentes básicos para tu juego de tres en línea. Para tener un juego completo, ahora necesitas alternar la colocación de «X» y «O» en el tablero, y necesitas una forma de determinar un ganador.

Levantar el estado

Actualmente, cada componente Square mantiene una parte del estado del juego. Para comprobar si hay un ganador en un juego de tres en línea, el Board necesitaría saber de alguna manera el estado de cada uno de los 9 componentes del Square.

¿Cómo abordarías eso? Al principio, puedes suponer que el Board necesita «preguntar» a cada Square por el estado de ese Square. Aunque este enfoque es técnicamente posible en React, no lo aconsejamos porque el código se vuelve difícil de entender, susceptible a errores y difícil de refactorizar. En cambio, el mejor enfoque es almacenar el estado del juego en el componente Board principal en lugar de en cada Square. El componente Board puede decirle a cada Square qué mostrar al pasar una prop, como lo hizo cuando pasó un número a cada Cuadrado.

Para recopilar datos de varios elementos secundarios o para que dos componentes secundarios se comuniquen entre sí, declara el estado compartido en tu componente principal. El componente padre puede devolver ese estado a los hijos a través de props. Esto mantiene los componentes secundarios sincronizados entre sí y con el componente principal.

Levantar el estado a un componente principal es común cuando se refactorizan los componentes de React.

Aprovechemos esta oportunidad para probarlo. Edita el componente Board para que declare una variable de estado llamada Square que por defecto sea una matriz de 9 valores nulos correspondientes a los 9

cuadrados:

```
// ...
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));
  return (
    // ...
  );
}
```

`Array(9).fill(null)` crea una matriz con nueve elementos y establece cada uno de ellos en `null`. La llamada `useState()` a su alrededor declara una variable de estado `squares` que inicialmente se establece en esa matriz. Cada entrada en la matriz corresponde al valor de un cuadrado. Cuando llenes el tablero más tarde, la matriz de «cuadrados» se verá así:

```
['O', null, 'X', 'X', 'X', 'O', 'O', null, null]
```

Ahora tu componente `Board` necesita pasar la prop `value` a cada uno de los componentes `Square` que representa:

```
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));
  return (
    <>
```

```
</> ); } A continuación, edita el componente Square para recibir la prop value del componente Board. Esto requerirá eliminar el propio seguimiento con estado del value del componente Square y la propiedad onClick del botón:
```

```
function Square({value}) {
  return {value};
}
```

En este punto, deberías ver un tablero de Tres en línea vacío:

tablero vacío

Y tu código debería verse así:

App.js

Descargar

Reiniciar

Bifurcar

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

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

```
function Square({ value }) {  
  return {value};  
}
```

```
export default function Board() {  
  const [squares, setSquares] = useState(Array(9).fill(null));  
  return (  
    <>  
  
    </> ); }  
}
```

Mostrar más

Cada Cuadrado ahora recibirá una prop de value que será 'X', 'O', o null para los cuadrados vacíos.

A continuación, debes cambiar lo que sucede cuando se hace clic en un Square. El componente Board ahora mantiene qué casillas están llenas. Necesitarás crear una forma para que Square actualice el estado de Board. Dado que el estado es privado para el componente que lo define, no puedes actualizar el estado de Board directamente desde Square.

En su lugar, pasa una función del componente Board al componente Square, y Square llamará a esa función cuando se haga clic en un cuadrado. Comienza con la función que llamará el componente Square cuando se haga clic en él. Llama a esa función onSquareClick:

```
function Square({ value }) {  
  return (  

```

```
{value}
```

```
);  
}
```

A continuación, agrega la función `onSquareClick` a las props del componente `Square`:

```
function Square({ value, onSquareClick }) {  
  return (  

```

```
{value}
```

```
);  
}
```

Ahora conecta la prop `onSquareClick` a una función en el componente `Board` que llamarás `handleClick`. Para conectar `onSquareClick` a `handleClick`, pasa una función a la prop `onSquareClick` del primer componente `Square`:

```
export default function Board() {  
  const [squares, setSquares] = useState(Array(9).fill(null));  
  
  return (  
    <>
```

```
//... ); } Por último, define la función handleClick dentro del componente Board para actualizar la matriz  
squares que contiene el estado de tu tablero:
```

```
export default function Board() {  
  const [squares, setSquares] = useState(Array(9).fill(null));  
  
  function handleClick() {  
    const nextSquares = squares.slice();  
    nextSquares[0] = "X";  
    setSquares(nextSquares);  
  }  
  
  return (  
    // ...  
  )  
}
```

La función `handleClick` crea una copia de la matriz `squares` (`nextSquares`) con el método JavaScript `slice()` `Array`. Luego, `handleClick` actualiza la matriz `nextSquares` para agregar `X` al primer cuadrado (índice `[0]`).

Llamar a la función `setSquares` le permite a React saber que el estado del componente ha cambiado. Esto activará una nueva representación de los componentes que usan el estado de `squares` (`Boards`), así como sus componentes secundarios (los componentes `squares` que forman el tablero).

Nota

JavaScript admite cierres, lo que significa que una función interna (por ejemplo, `handleClick`) tiene acceso a variables y funciones definidas en una función externa (por ejemplo, `Board`). La función `handleClick`

puede leer el estado squares y llamar al método setSquares porque ambos están definidos dentro de la función Board.

Ahora puedes agregar X al tablero... pero solo al cuadrado superior izquierdo. Tu función handleClick está codificada para actualizar el índice del cuadrado superior izquierdo (0). Actualicemos handleClick para poder actualizar cualquier cuadrado. Agrega un argumento i a la función handleClick que toma el índice del cuadrado que debe actualizarse:

```
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));

  function handleClick(i) {
    const nextSquares = squares.slice();
    nextSquares[i] = "X";
    setSquares(nextSquares);
  }

  return (
    // ...
  )
}
¡Ahora hay un nuevo problema!
```

Intenta configurar la prop onSquareClick del cuadrado para que sea handleClick(0) directamente en el JSX de esta manera:

La llamada a handleClick(0) será parte de la renderización del componente del tablero. Debido a que handleClick(0) altera el estado del componente del tablero llamando a setSquares, todo el componente del tablero se volverá a renderizar. Pero handleClick(0) ahora es parte de la renderización del componente del tablero, por lo que haz creado un bucle infinito:

Console

Too many re-renders. React limits the number of renders to prevent an infinite loop.

(Traducción)

Demasiados re-renderizados. React limita el número de renderizados para evitar un bucle infinito.

¿Por qué este problema no sucedió antes?

Cuando estabas pasando onSquareClick={handleClick}, estabas pasando la función handleClick como una prop. ¡No lo estabas llamando! Pero ahora estás llamando a esa función de inmediato, observa los paréntesis en handleClick(0), y es por eso que se ejecuta demasiado rápido. ¡No quieres llamar a handleClick hasta que el usuario haga clic!

Podrías arreglar esto creando una función como handleFirstSquareClick que llame a handleClick(0), una función como handleSecondSquareClick que llame a handleClick(1), y así sucesivamente. Tú podrías pasar (en lugar de llamar) a estas funciones como props de la forma onSquareClick={handleFirstSquareClick}. Esto resolvería el bucle infinito.

Sin embargo, definir nueve funciones diferentes y darles un nombre a cada una de ellas es demasiado detallado. En cambio, hagamos esto:

```
export default function Board() {  
  // ...  
  return (  
    <>
```

handleClick(0)} /> // ...); } Observa la nueva sintaxis () =>. Aquí, () => handleClick(0) es una función de flecha, que es una forma más corta de definir funciones. Cuando se hace clic en el cuadrado, se ejecutará el código después de la «flecha» =>, llamando a handleClick(0).

Ahora necesitas actualizar los otros ocho cuadrados para llamar a handleClick desde las funciones de flecha que pasas. Asegúrate de que el argumento para cada llamada handleClick corresponda al índice del cuadrado correcto:

```
export default function Board() {  
  // ...  
  return (  
    <>
```

```
    handleClick(0)} /> handleClick(1)} /> handleClick(2)} />  
    handleClick(3)} /> handleClick(4)} /> handleClick(5)} />  
    handleClick(6)} /> handleClick(7)} /> handleClick(8)} />
```

```
</> ); }; Ahora puedes volver a agregar X a cualquier casilla del tablero haciendo clic en ellos:
```

llenado el tablero con X

¡Pero esta vez toda el manejo del estado está a cargo del componente Board!

Así es como debería verse tu código:

App.js

Descargar

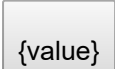
Reiniciar

Bifurcar

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15

16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

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

```
function Square({ value, onSquareClick }) {  
  return (  
     {value}  
  );  
}
```

```
export default function Board() {  
  const [squares, setSquares] = useState(Array(9).fill(null));
```

```
  function handleClick(i) {  
    const nextSquares = squares.slice();  
    nextSquares[i] = 'X';  
    setSquares(nextSquares);  
  }
```

```
  return (  
    <>
```

```
    handleClick(0) /> handleClick(1) /> handleClick(2) />  
    handleClick(3) /> handleClick(4) /> handleClick(5) />  
    handleClick(6) /> handleClick(7) /> handleClick(8) />
```

Mostrar más

Ahora que tu manejo de estado está en el componente Board, el componente padre Board pasa props a

los componentes hijos Square para que puedan mostrarse correctamente. Al hacer clic en un Square, el componente secundario Square ahora le pide al componente principal Board que actualice el estado del tablero. Cuando el estado de Board cambia, tanto el componente Board como todos los componentes secundarios Square se vuelven a renderizar automáticamente. Mantener el estado de todos los cuadrados en el componente Board te permite determinar el ganador en el futuro.

Recapitulemos lo que sucede cuando un usuario hace clic en el cuadrado superior izquierdo de su tablero para agregarle una X:

Al hacer clic en el cuadrado superior izquierdo, se ejecuta la función que el button recibe como prop onClick del Square. El componente Square recibe esa función como una prop onSquareClick del Board. El componente Board define esa función directamente en el JSX. Llama a handleClick con un argumento de 0.

handleClick usa el argumento (0) para actualizar el primer elemento de la matriz squares de null a X.

El estado squares del componente Board se actualiza, por lo que Board y todos sus elementos secundarios se vuelven a renderizar. Esto hace que la prop value del componente Square con el índice 0 cambie de null a X.

Al final, el usuario ve que el cuadrado superior izquierdo ha pasado de estar vacío a tener una X después de hacer clic en él.

Nota

El atributo onClick del elemento DOM

tiene un significado especial para React porque es un componente integrado. Para componentes personalizados como Square, el nombre depende de ti. Podrías dar cualquier nombre a la prop onSquareClick de Square o handleClick de Board, y el código funcionaría de la misma manera. En React, es convencional usar nombres on[Event] para props que representan eventos y handle[Event] para las definiciones de funciones que controlan los eventos.

¿Por qué es importante la inmutabilidad?

Observa cómo en handleClick, llama a .slice() para crear una copia de la matriz squares en lugar de modificar la matriz existente. Para explicar por qué, necesitamos discutir la inmutabilidad y por qué es importante aprender la inmutabilidad.

En general, hay dos enfoques para cambiar los datos. El primer enfoque es mutar los datos cambiando directamente los valores de los datos. El segundo enfoque es reemplazar los datos con una nueva copia que tenga los cambios deseados. Así es como se vería si mutaras la matriz squares:

```
const squares = [null, null, null, null, null, null, null, null, null];
squares[0] = 'X';
// Ahora squares es ["X", null, null, null, null, null, null, null, null];
Y así es como se vería si cambiaras los datos sin mutar de la matriz squares:
```

```
const squares = [null, null, null, null, null, null, null, null, null];
const nextSquares = ['X', null, null, null, null, null, null, null, null];
// Ahora squares no ha cambiado, pero el primer elemento de nextSquares es 'X' en lugar de null
El resultado final es el mismo, pero al no mutar (cambiar los datos subyacentes) directamente, obtienes varios beneficios.
```

La inmutabilidad hace que las características complejas sean mucho más fáciles de implementar. Más adelante en este tutorial, implementarás una característica de «viaje en el tiempo» que te permitirá revisar el historial del juego y «saltar atrás» a movimientos pasados. Esta funcionalidad no es específica de los juegos: la habilidad de deshacer y rehacer ciertas acciones es un requisito común para las aplicaciones. Evitar la mutación directa de los datos permite mantener intactas las versiones previas de los datos y reutilizarlas más adelante.

También hay otro beneficio de la inmutabilidad. De forma predeterminada, todos los componentes secundarios se vuelven a renderizar automáticamente cuando cambias el estado de un componente principal. Esto incluye incluso los componentes secundarios que no se vieron afectados por el cambio. Aunque el usuario no nota la renderización en sí misma (no debes tratar de evitarla de forma activa), es

Aunque el usuario no nota la renderización en sí misma (¡no debes tratar de evitarla de forma activa!), es posible que desees omitir la renderización de una parte del árbol que claramente no se vio afectada por razones de rendimiento. La inmutabilidad hace que sea muy barato para los componentes comparar si sus datos han cambiado o no. Puedes obtener más información sobre cómo React elige cuándo volver a renderizar un componente en la documentación de referencia de la API memo.

Tomar turnos

Ahora es el momento de corregir un defecto importante en este juego de tres en línea: las «O» no se pueden marcar en el tablero.

Establece que el primer movimiento sea «X» de forma predeterminada. Hagamos un seguimiento de esto agregando otra parte del estado al componente Board:

```
function Board() {
  const [xlsNext, setXlsNext] = useState(true);
  const [squares, setSquares] = useState(Array(9).fill(null));

  // ...
}
```

Cada vez que un jugador se mueve, xlsNext (un valor booleano) se invertirá para determinar qué jugador es el siguiente y se guardará el estado del juego. Actualiza la función handleClick de Board para cambiar el valor de xlsNext:

```
export default function Board() {
  const [xlsNext, setXlsNext] = useState(true);
  const [squares, setSquares] = useState(Array(9).fill(null));

  function handleClick(i) {
    const nextSquares = squares.slice();
    if (xlsNext) {
      nextSquares[i] = "X";
    } else {
      nextSquares[i] = "O";
    }
    setSquares(nextSquares);
    setXlsNext(!xlsNext);
  }

  return (
    //...
  );
}
```

Ahora, al hacer clic en diferentes cuadrados, se alternarán entre X y O, ¡como deberían!

Pero espera, hay un problema. Intenta hacer clic en el mismo cuadrado varias veces:

O sobrescribiendo una X

¡La X se sobrescribe con una O! Si bien esto agregaría un giro muy interesante al juego, por ahora nos apegaremos a las reglas originales.

Cuando marcas un cuadrado con una X o una O, no estás comprobando primero si el cuadrado ya tiene un valor X u O. Puedes arreglar esto regresando rápidamente en el estado. Verifica si el cuadrado ya tiene una X o una O. Si el cuadrado ya está lleno, genera un return en la función handleClick, antes de que intente actualizar el estado del tablero.

```
function handleClick(i) {
  if (squares[i]) {
    return;
  }
  const nextSquares = squares.slice();
  //...
}
```

¡Ahora solo puedes agregar X u O a los cuadrados vacíos! Así es como debería verse tu código en este punto:

App.js
Descargar
Reiniciar

Birurcar

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

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

function Square({value, onSquareClick}) {
  return (
```

{value}

```
);
}
```

```
export default function Board() {
  const [xIsNext, setXIsNext] = useState(true);
  const [squares, setSquares] = useState(Array(9).fill(null));

  function handleClick(i) {
    if (squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
      nextSquares[i] = 'X';
    } else {
      nextSquares[i] = 'O';
    }
  }
}
```

```

}
setSquares(nextSquares);
setXIsNext(!xIsNext);
}

return (
  <>

  handleClick(0)} /> handleClick(1)} /> handleClick(2)} />

```

Mostrar más

Declarar un ganador

Ahora que muestras el turno del siguiente jugador, también debes mostrar cuándo se gana el juego y cuando no hay más turnos. Para hacer esto, agrega una función de ayuda llamada `calculateWinner` que toma una matriz de 9 cuadrados, busca un ganador y devuelve 'X', 'O' o null según corresponda. No te preocupes demasiado por la función `calculateWinner`; no es específica de React:

```

export default function Board() {
  //...
}

function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6]
  ];
  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
      return squares[a];
    }
  }
  return null;
}

```

Nota

No importa si defines `calculateWinner` antes o después del `Board`. Pongámoslo al final para que no tengas que desplazarte cada vez que edites tus componentes.

Lamarás a `calculateWinner(squares)` en la función `handleClick` del componente `Board` para comprobar si un jugador ha ganado. Puedes realizar esta verificación al mismo tiempo que verificas si un usuario ha hecho clic en un cuadrado que ya tiene una X o una O. Nos gustaría regresar rápido en el estado en ambos casos:

```
function handleClick(i) {
  if (squares[i] || calculateWinner(squares)) {
    return;
  }
  const nextSquares = squares.slice();
  //...
}
```

Para que los jugadores sepan cuándo termina el juego, puedes mostrar un texto como «Ganador: X» o «Ganador: O». Para hacerlo, agrega una sección status al componente Board. El estado mostrará el ganador si el juego termina y si el juego está en curso, se mostrará el turno del siguiente jugador:

```
export default function Board() {
  // ...
  const winner = calculateWinner(squares);
  let status;
  if (winner) {
    status = "Ganador: " + winner;
  } else {
    status = "Siguiente jugador: " + (xIsNext ? "X" : "O");
  }
}
```

```
return (
```

```
<>
```

```
{status}
```

```
// ... ) } ¡Felicidades! Ahora tienes un juego de Tres en línea que funciona. Y acabas de aprender los
conceptos básicos de React también. Así que tú eres el verdadero ganador aquí. Así es como debería
verse el código:
```

App.js

Descargar

Reiniciar

Bifurcar

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13

14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

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

```
function Square({value, onSquareClick}) {  
  return (  

```

```
     {value}
```

```
  );  
}
```

```
export default function Board() {  
  const [xIsNext, setXIsNext] = useState(true);  
  const [squares, setSquares] = useState(Array(9).fill(null));
```

```
  function handleClick(i) {  
    if (calculateWinner(squares) || squares[i]) {  
      return;  
    }  
    const nextSquares = squares.slice();  
    if (xIsNext) {  
      nextSquares[i] = 'X';  
    } else {  
      nextSquares[i] = 'O';  
    }  
    setSquares(nextSquares);
```

```

setXlsNext(!xlsNext);
}

const winner = calculateWinner(squares);
let status;
if (winner) {
  status = 'Ganador: ' + winner;
} else {
  status = 'Siguiente jugador: ' + (xlsNext ? 'X' : 'O');
}

```

Mostrar más

Agregar viajes en el tiempo

Como ejercicio final, hagamos posible «retroceder en el tiempo» a los movimientos anteriores del juego.

Almacenar un historial de movimientos

Si mutaras la matriz squares, implementar el viaje en el tiempo sería muy difícil.

Sin embargo, usas slice() para crear una nueva copia de la matriz squares después de cada movimiento, y la tratas como inmutable. Esto te permite almacenar todas las versiones anteriores de la matriz squares y navegar entre los giros que ya han ocurrido.

Almacena las matrices anteriores de squares en otra matriz llamada history, que almacenarás como una nueva variable de estado. La matriz history representa todos los estados del tablero, desde el primero hasta el último movimiento, y tiene una forma como esta:

```

[
  // Antes del primer movimiento
  [null, null, null, null, null, null, null, null, null],
  // Después del primer movimiento
  [null, null, null, null, 'X', null, null, null, null],
  // Después del segundo movimiento
  [null, null, null, null, 'X', null, null, null, 'O'],
  // ...
]

```

Levantar el estado, otra vez

Ahora escribe un nuevo componente de nivel superior llamado Game para mostrar una lista de movimientos anteriores. Ahí es donde colocarás el estado de history que contiene todo el historial del juego.

Colocar el estado history en el componente Game te permite eliminar el estado squares de tu componente hijo Board. Al igual que «levantar el estado» del componente Square al componente Board, ahora puedes levantarlo del Board al componente Game de nivel superior. Esto le da al componente Game control total sobre los datos del Board y le permite instruir al Game para renderizar los turnos anteriores del history.

Primero, agrega un componente Game con export default. Haz que represente el componente Board dentro de algunas marcas JSX:

```
function Board() {
  // ...
}
```

```
export default function Game() {
  return (
```

```
    {/_TODO_/}
```

); } Ten en cuenta que estás eliminando las palabras claves export default antes de la declaración function Board() { y agregándolas antes de la declaración function Game() {. Esto le dice a tu archivo index.js que use el componente Game como el componente de nivel superior en lugar de su componente Board. Los 'div' adicionales devueltos por el componente Game están dejando espacio para la información del juego que agregarás al tablero más adelante.

Agrega algún estado al componente Game para rastrear qué jugador es el siguiente y el historial de movimientos:

```
export default function Game() {
  const [xlsNext, setXlsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  // ...
```

Observa cómo [Array(9).fill(null)] es una matriz con un solo elemento, que a su vez es una matriz de 9 nulls.

Para renderizar los cuadrados en el movimiento actual, debes leer la matriz de los últimos cuadrados del history. No necesitas useState para esto; ya tienes suficiente información para calcularlo durante el renderizado:

```
export default function Game() {
  const [xlsNext, setXlsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const currentSquares = history[history.length - 1];
  // ...
```

A continuación, crea una función handlePlay dentro del componente Game que será llamada por el componente Board para actualizar el juego. Pasa xlsNext, currentSquares y handlePlay como props al componente Board:

```
export default function Game() {
  const [xlsNext, setXlsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const currentSquares = history[history.length - 1];

  function handlePlay(nextSquares) {
    // TODO
  }

  return (
```

//...) } Hagamos que el componente Board esté completamente controlado por las props que recibe. Cambia el componente Board para que tome tres props: xlsNext, squares y una nueva función onPlay que Board puede llamar con la matriz de cuadrados actualizada cada vez que un jugador hace un movimiento. A continuación, elimina las dos primeras líneas de la función Board que llama a useState:

```
function Board({ xlsNext, squares, onPlay }) {  
  function handleClick(i) {  
    //...  
  }  
  // ...  
}
```

Ahora reemplaza las llamadas setSquares y setXlsNext en handleClick en el componente Board con una sola llamada a su nueva función onPlay para que el componente Game pueda actualizar el componente Board cuando el usuario hace clic en un cuadrado:

```
function Board({ xlsNext, squares, onPlay }) {  
  function handleClick(i) {  
    if (calculateWinner(squares) || squares[i]) {  
      return;  
    }  
    const nextSquares = squares.slice();  
    if (xlsNext) {  
      nextSquares[i] = "X";  
    } else {  
      nextSquares[i] = "O";  
    }  
    onPlay(nextSquares);  
  }  
  //...  
}
```

El componente Board está totalmente controlado por las props que le pasa el componente Game. Necesitas implementar la función handlePlay en el componente Game para que el juego vuelva a funcionar.

¿Qué debería hacer handlePlay cuando se llama? Recuerda que Board solía llamar a setSquares con una matriz actualizada; ahora pasa la matriz squares actualizada a onPlay.

La función handlePlay necesita actualizar el estado de Game para activar una nueva representación, pero ya no tienes una función setSquares a la que puedas llamar; ahora estás usando el estado variable history para almacenar esta información. Actualiza el history agregando la matriz squares actualizada como una nueva entrada en el historial. También puedes alternar xlsNext, tal como solía hacer el componente Board:

```
export default function Game() {  
  //...  
  function handlePlay(nextSquares) {  
    setHistory([...history, nextSquares]);  
    setXlsNext(!xlsNext);  
  }  
}
```

```
}  
//...  
}
```

Aquí, [...history, nextSquares] crea una nueva matriz que contiene todos los elementos en history, seguido de nextSquares. (Puedes leer la sintaxis extendida ...history como «enumerar todos los elementos en history».)

Por ejemplo, si history es igual a [[null,null,null], ["X",null,null]] y nextSquares es igual a ["X",null,"O"], entonces el nuevo arreglo[...history, nextSquares] será [[null,null,null], ["X",null,null], ["X",null,"O"]].

En este punto, haz movido el estado para vivir en el componente Game, y la interfaz de usuario debería estar funcionando completamente, tal como estaba antes de la refactorización. Así es como debería verse tu código en este punto:

App.js

Descargar

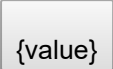
Reiniciar

Bifurcar

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28

29
30
31
32
33
34
35
36

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

```
function Square({ value, onSquareClick }) {  
  return (  
    {value}  
  );  
}
```

```
function Board({ xIsNext, squares, onPlay }) {  
  function handleClick(i) {  
    if (calculateWinner(squares) || squares[i]) {  
      return;  
    }  
    const nextSquares = squares.slice();  
    if (xIsNext) {  
      nextSquares[i] = 'X';  
    } else {  
      nextSquares[i] = 'O';  
    }  
    onPlay(nextSquares);  
  }  
  
  const winner = calculateWinner(squares);  
  let status;  
  if (winner) {  
    status = 'Ganador: ' + winner;  
  } else {  
    status = 'Siguiente Jugador: ' + (xIsNext ? 'X' : 'O');  
  }  
}
```

```
  return (  
    <>  
  
    {status}  
  )  
}
```

Mostrar más

Mostrar los movimientos anteriores

Dado que estás grabando el historial del juego de tres en línea, ahora puede mostrárselo al jugador como una lista de movimientos anteriores.

Los elementos de React como

son objetos regulares de JavaScript; puedes pasarlos en tu aplicación. Para representar varios elementos en React, puedes usar una matriz de elementos de React.

Ya tienes una matriz de movimientos history en el estado, por lo que ahora necesitas transformarla en una matriz de elementos React. En JavaScript, para transformar una matriz en otra, puede usar el método de array map:

```
[1, 2, 3].map((x) => x * 2) // [2, 4, 6]
```

Utilizarás map para transformar tu matriz de movimientos history en elementos React que representen botones en la pantalla, y mostrarás una lista de botones para «saltar» a movimientos anteriores. Hagamos un map sobre la matriz history en el componente Game:

```
export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const currentSquares = history[history.length - 1];

  function handlePlay(nextSquares) {
    setHistory([...history, nextSquares]);
    setXIsNext(!xIsNext);
  }

  function jumpTo(nextMove) {
    // TODO
  }

  const moves = history.map((squares, move) => {
    let description;
    if (move > 0) {
      description = 'Ir al movimiento #' + move;
    } else {
      description = 'Ir al inicio del juego';
    }
    return (
      •

```

```
jumpTo(move))>{description} ); });
```

```
return (
```

```
  {moves}
```

```
); } Puedes ver cómo debería verse tu código a continuación. Ten en cuenta que deberías ver un error en la consola de herramientas para desarrolladores que dice:
```

Console

Warning: Each child in an array or iterator should have a unique «key» prop. Check the render method of `Game`.

Resolverás este error en la siguiente sección.

App.js

Descargar

Reiniciar

Bifurcar

1

2

3

4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

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

```
function Square({ value, onSquareClick }) {  
  return (  

```

```
    {value}
```

```
  );  
}
```

```
function Board({ xIsNext, squares, onPlay }) {  
  function handleClick(i) {  
    if (calculateWinner(squares) || squares[i]) {  
      return;
```



```

}
const nextSquares = squares.slice();
if (xIsNext) {
  nextSquares[i] = 'X';
} else {
  nextSquares[i] = 'O';
}
onPlay(nextSquares);
}

const winner = calculateWinner(squares);
let status;
if (winner) {
  status = 'Ganador: ' + winner;
} else {
  status = 'Siguiente jugador: ' + (xIsNext ? 'X' : 'O');
}

return (
  <>


  {status}

```

Console (1)

Warning: Each child in a list should have a unique "key" prop.

Check the render method of `Game`. See <https://reactjs.org/link/warning-keys> for more information.
 at li

at Game (<https://786946de.sandpack-bundler-4bw.pages.dev/src/App.js>  53)

Mostrar más

Cuando recorres la matriz `history` dentro de la función que has pasado a `map`, el argumento `squares` recorre cada elemento de `history`, y el argumento `move` recorre cada índice de la matriz: 0, 1, 2, (En la mayoría de los casos, necesitarías los elementos concretos de la matriz, pero para representar una lista de movimientos sólo necesitarás los índices).

Para cada movimiento en el historial del juego de tres en línea, creas un elemento de lista

- que contiene un botón

. El botón tiene un controlador onClick que llama a una función llamada jumpTo (que aún no has implementado).

Por ahora, deberías ver una lista de los movimientos que ocurrieron en el juego y un error en la consola de herramientas del desarrollador.

Analicemos qué significa el error «key».

Elegir una key

Cuando renderizas una lista, React almacena cierta información sobre cada elemento renderizado. Cuando actualizas la lista, React necesita determinar qué ha cambiado. Podrías haber agregado, eliminado, reorganizado o actualizado los elementos de la lista.

Imagina la transición de

- Alexa: Quedan 7 tareas
- Ben: Quedan 5 tareas hacia
- Ben: Quedan 9 tareas pendientes
- Claudia: Quedan 8 tareas pendientes
- Alexa: Quedan 5 tareas pendientes

Además de los recuentos actualizados, un humano que leyera esto probablemente diría que has intercambiado el orden de Alexa y Ben e insertado a Claudia entre Alexa y Ben. Sin embargo, React es un programa de computadora y no puede saber lo que pretendías, por lo que necesitas especificar una propiedad key para cada elemento de la lista y así diferenciar cada elemento de la lista de sus hermanos. Si tus datos provinieran de una base de datos, los IDs de Alexa, Ben y Claudia podrían usarse como keys.

- {user.name}: {user.taskCount} tareas pendientes

Cuando se vuelve a representar una lista, React toma la key de cada elemento de la lista y busca en los elementos de la lista anterior una key coincidente. Si la lista actual tiene una key que no existía antes, React crea un componente. Si a la lista actual le falta una key que existía en la lista anterior, React destruye el componente anterior. Si dos keys coinciden, se mueve el componente correspondiente.

Las key informan a React sobre la identidad de cada componente, lo que permite a React mantener el estado entre renderizaciones. Si la key de un componente cambia, el componente se destruirá y se volverá a crear con un nuevo estado.

key es una propiedad especial y reservada en React. Cuando se crea un elemento, React extrae la propiedad key y almacena la key directamente en el elemento devuelto. Aunque puede parecer que key se pasa como prop, React usa automáticamente key para decidir qué componentes actualizar. No hay forma de que un componente pregunte qué key especificó su padre.

Se recomienda encarecidamente que asignes las key adecuadas cada vez que crees listas dinámicas. Si no tienes una key adecuada, puedes considerar reestructurar tus datos para que las tenga.

Si no se especifica ninguna key, React informará un error y utilizará el índice de matriz como key de forma predeterminada. El uso del índice de la matriz como key es problemático cuando se intenta reordenar los elementos de una lista o al insertar/eliminar elementos de la lista. Pasar explícitamente key={i} silencia el error pero tiene los mismos problemas que los índices de matriz y no se recomienda en la mayoría de los casos.

Las keys no necesitan ser globalmente únicas; solo necesitan ser únicas entre los componentes y sus hermanos.

Implementación de viajes en el tiempo

En la historia del juego de tres en línea, cada movimiento pasado tiene una identificación única asociada: es el número secuencial del movimiento. Los movimientos nunca se reordenarán, eliminarán o insertarán en el medio, por lo que es seguro usar el índice de movimiento como key.

En la función Game, puedes agregar la key como

- , y si vuelves a cargar el juego renderizado, el error de «key» de React debería desaparecer:

```
const moves = history.map((squares, move) => {
  //...
  return (
```

•

```
jumpTo(move)}>{description}
```

```
); });
```

App.js

Descargar

Reiniciar

Bifurcar

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

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

```

function Square({ value, onSquareClick }) {
  return (
    

{value}


  );
}

function Board({ xIsNext, squares, onPlay }) {
  function handleClick(i) {
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
      nextSquares[i] = 'X';
    } else {
      nextSquares[i] = 'O';
    }
    onPlay(nextSquares);
  }

  const winner = calculateWinner(squares);
  let status;
  if (winner) {
    status = 'Ganador: ' + winner;
  } else {
    status = 'Siguiente jugador: ' + (xIsNext ? 'X' : 'O');
  }

  return (
    <>

    {status}
  );
}

```

Mostrar más

Antes de que puedas implementar `jumpTo`, necesitas que el componente `Game` lleve la cuenta del paso que el usuario está viendo en ese momento. Para ello, define una nueva variable de estado llamada `currentMove`, que reciba como valor por defecto a 0:

```

export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const [currentMove, setCurrentMove] = useState(0);
  const currentSquares = history[history.length - 1];
  //...
}

```

A continuación, actualiza la función `jumpTo` dentro de `Game` para actualizar ese `currentMove`. También establece `xIsNext` en `true` si el número al que estás cambiando `currentMove` es par.

```
export default function Game() {
  // ...
  function jumpTo(nextMove) {
    setCurrentMove(nextMove);
    setXIsNext(nextMove % 2 === 0);
  }
  //...
}
```

Ahora haz dos cambios en la función `handlePlay` del `Game` que se llama cuando haces clic en un cuadrado.

Si «retrocedes en el tiempo» y luego haces un nuevo movimiento desde ese punto, sólo querrás mantener el historial hasta ese punto. En lugar de añadir `nextSquares` después de todos los elementos (sintaxis extendida ...) en `history`, lo agregarás después de todos los elementos en `history.slice(0, currentMove + 1)` para que sólo estés manteniendo esa porción de la historia antigua.

Cada vez que se realiza un movimiento, es necesario actualizar `currentMove` para que apunte a la última entrada del historial.

```
function handlePlay(nextSquares) {
  const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
  setHistory(nextHistory);
  setCurrentMove(nextHistory.length - 1);
  setXIsNext(!xIsNext);
}
```

Finalmente, modifica el componente `Game` para representar el movimiento seleccionado actualmente, en lugar de representar siempre el movimiento final:

```
export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const [currentMove, setCurrentMove] = useState(0);
  const currentSquares = history[currentMove];

  // ...
}
```

Si haces clic en cualquier paso en el historial del juego, el tablero de tres en línea debería actualizarse inmediatamente para mostrar cómo se veía el tablero después de que ocurriera ese paso.

App.js

Descargar


Reiniciar

Bifurcar

- 1
- 2
- 3
- 4
- 5

6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

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

```
function Square({value, onSquareClick}) {  
  return (  
     {value}  
  );  
}
```

```
function Board({ xIsNext, squares, onPlay }) {  
  function handleClick(i) {  
    if (calculateWinner(squares) || squares[i]) {  
      return;  
    }  
    const nextSquares = squares.slice();
```

```

if (xlsNext) {
  nextSquares[i] = 'X';
} else {
  nextSquares[i] = 'O';
}
onPlay(nextSquares);
}

const winner = calculateWinner(squares);
let status;
if (winner) {
  status = 'Ganador: ' + winner;
} else {
  status = 'Siguiente Jugador: ' + (xlsNext ? 'X' : 'O');
}

return (
  <>

  {status}

```

Mostrar más

Limpieza final

Si miras el código muy de cerca, puedes notar que `xlsNext === true` cuando `currentMove` es par y `xlsNext === false` cuando `currentMove` es impar. En otras palabras, si conoces el valor de `movimientoActual`, entonces siempre puedes averiguar cuál debería ser `xlsNext`.

No hay ninguna razón para que almacenes ambos en el estado. De hecho, intenta siempre evitar el estado redundante. Simplificar lo que almacenas en el estado reduce los errores y hace que tu código sea más fácil de entender. Cambia `Game` para que no almacene `xlsNext` como una variable de estado separada y en su lugar lo calcule basándose en el `currentMove`:

```

export default function Game() {
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const [currentMove, setCurrentMove] = useState(0);
  const xlsNext = currentMove % 2 === 0;
  const currentSquares = history[currentMove];

  function handlePlay(nextSquares) {
    const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
    setHistory(nextHistory);
    setCurrentMove(nextHistory.length - 1);
  }

  function jumpTo(nextMove) {
    setCurrentMove(nextMove);
  }

  // ...

```

}

Ya no necesitas la declaración de estado `xlsNext` ni las llamadas a `setXlsNext`. Ahora, no hay posibilidad de que `xlsNext` no esté sincronizado con `currentMove`, incluso si cometes un error al codificar los componentes.

Finalizar

¡Felicidades! Has creado un juego de Tres en línea que:

Te permite jugar Tres en línea,

Indica cuando un jugador ha ganado el juego,

Almacena el historial de un juego a medida que avanza un juego,

Permite a los jugadores revisar el historial de un juego y ver versiones anteriores del tablero de un juego.

¡Buen trabajo! Esperamos que ahora sientas que tiene una comprensión decente de cómo funciona React.

Mira el resultado final aquí:

[App.js](#)

[Descargar](#)


[Reiniciar](#)

[Bifurcar](#)

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26

27
28
29
30
31
32
33
34
35
36

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

```
function Square({ value, onSquareClick }) {  
  return (  
     {value}  
  );  
}
```

```
function Board({ xIsNext, squares, onPlay }) {  
  function handleClick(i) {  
    if (calculateWinner(squares) || squares[i]) {  
      return;  
    }  
    const nextSquares = squares.slice();  
    if (xIsNext) {  
      nextSquares[i] = 'X';  
    } else {  
      nextSquares[i] = 'O';  
    }  
    onPlay(nextSquares);  
  }  
}
```

```
const winner = calculateWinner(squares);  
let status;  
if (winner) {  
  status = 'Ganador: ' + winner;  
} else {  
  status = 'Siguiente jugador: ' + (xIsNext ? 'X' : 'O');  
}  
  
return (  
  <>  
  
  {status}  
)
```

Mostrar más

Si tienes tiempo extra o quieres practicar tus nuevas habilidades de React, aquí hay algunas ideas de

mejoras que podría hacer al juego de tres en línea, enumeradas en orden de dificultad creciente:

Solo para el movimiento actual, muestra «Estás en el movimiento #...» en lugar de un botón

Vuelve a escribir Board para usar dos bucles para crear los cuadrados en lugar de codificarlos.

Agrega un botón de alternancia que te permita ordenar los movimientos en orden ascendente o descendente.

Cuando alguien gane, resalta los tres cuadrados que causaron la victoria (y cuando nadie gane, muestra un mensaje indicando que el resultado fue un empate).

Muestra la ubicación de cada movimiento en el formato (columna, fila) en la lista del historial de movimientos.

A lo largo de este tutorial, haz tocado los conceptos de React, incluidos los elementos, los componentes, las props y el estado. Ahora que haz visto cómo funcionan estos conceptos al crear un juego, consulta Pensando en React para ver cómo funcionan los mismos conceptos de React al crear la interfaz de usuario de una aplicación.

[Anterior](#)

[Inicio rápido](#)

[Siguiendo](#)

[Pensar en React](#)

©2024

uwu?