**React**

**Crea un elemento JSX simple**

React es una librería de vistas de código abierto creada y mantenida por Facebook. Es una gran herramienta para hacer la interfaz de usuario (UI) de aplicaciones web modernas.

React usa una extensión de sintaxis de JavaScript llamada JSX que te permite escribir HTML directamente dentro de JavaScript. Esto tiene muchos beneficios. Te permite utilizar el poder programático completo de JavaScript dentro de HTML, y ayuda a mantener tu código legible. En su mayor parte, JSX es similar al HTML que ya has aprendido. Sin embargo, hay algunas diferencias clave que se abordarán a lo largo de estos desafíos.

Por ejemplo, dado que JSX es una extensión sintáctica de JavaScript, se puede escribir JavaScript directamente dentro de JSX. Para hacer esto, simplemente incluye el código que deseas que sea tratado como JavaScript entre llaves: { 'this is treated as JavaScript code' }. Ten esto en cuenta, ya que se utiliza en varios desafíos futuros.

Sin embargo, debido a que JSX no es JavaScript válido, el código JSX debe ser compilado en JavaScript. El transpilador Babel es una herramienta muy popular para este proceso. Para tu comodidad, ya se ha añadido tras bambalinas para estos desafíos. Si escribes JSX no válido sintácticamente, verás que la primera prueba de estos desafíos falla.

Vale la pena señalar que por debajo los desafíos están llamando ReactDOM.render(JSX, document.getElementById('root')). Esta llamada de función es la que coloca tu JSX en la representación ligera del DOM de React. React entonces utiliza capturas instantáneas de su propio DOM para optimizar actualizando sólo partes específicas del DOM.

**const JSX = <h1>Hello JSX!</h1>;**

**Crea un elemento JSX complejo**

El último desafío fue un ejemplo sencillo de JSX, pero JSX también puede representar HTML más complejo.

Una cosa importante que debes saber sobre JSX anidado es que debe devolver un solo elemento.

Este elemento principal contendría a todos los demás niveles de elementos anidados.

Por ejemplo, varios elementos JSX escritos al mismo nivel sin elemento contenedor principal no se transpilarán.

Aquí va un ejemplo:

**JSX válido:**

**<div>**

**<p>Paragraph One</p>**

**<p>Paragraph Two</p>**

**<p>Paragraph Three</p>**

**</div>**

**JSX inválido:**

**<p>Paragraph One</p>**

**<p>Paragraph Two</p>**

**<p>Paragraph Three</p>**

**const JSX =<div>**

**<h1></h1>**

**<p></p>**

**<ul>**

**<li></li>**

**<li></li>**

**<li></li>**

**</ul>**

**</div>**

**Agrega comentarios en JSX**

JSX es una sintaxis que se compila en JavaScript válido. A veces, para facilitar la lectura, es necesario añadir comentarios al código. Como la mayoría de los lenguajes de programación, JSX tiene su propia manera de hacerlo.

Para agregar comentarios dentro de JSX, se utiliza la sintaxis {/\* \*/} para envolver el texto del comentario.

**const JSX = (**

**<div>**

**<h1>This is a block of JSX</h1>{/\* \*/}**

**<p>Here's a subtitle</p>**

**</div>**

**);**

**Renderiza elementos HTML al DOM**

Hasta ahora, has aprendido que JSX es una herramienta conveniente para escribir HTML legible dentro de JavaScript. Con React, podemos renderizar este JSX directamente al DOM HTML usando la API de renderizado de React conocida como ReactDOM.

ReactDOM ofrece un método simple para renderizar elementos React al DOM que se ve así: ReactDOM.render(componentToRender, targetNode), donde el primer argumento es el elemento o componente React que deseas renderizar, y el segundo argumento es el nodo DOM al que se quiere renderizar el componente.

Como era de esperarse, ReactDOM.render() debe llamarse después de las declaraciones de los elementos JSX, al igual que hay que declarar las variables antes de usarlas.

**const JSX = (**

**<div>**

**<h1>Hello World</h1>**

**<p>Lets render this to the DOM</p>**

**</div>**

**);**

**// Agrega tu código debajo de esta**

**ReactDOM.render(JSX, document.getElementById("challenge-node"));**

**Define una clase HTML en JSX**

Ahora que te sientes cómodo escribiendo JSX, te preguntarás cuanto difiere de HTML.

Hasta ahora, puede parecer que HTML y JSX son exactamente iguales.

Una diferencia clave en JSX es que ya no puedes usar la palabra class para definir clases HTML. Esto es debido a que class es una palabra reservada en JavaScript. En su lugar, JSX utiliza className.

De hecho, la convención de nomenclatura para todos los atributos HTML y referencias a eventos en JSX se convierte a camelCase. Por ejemplo, un evento de clic en JSX es onClick, en lugar de onclick. Del mismo modo, onchange se convierte en onChange. Si bien se trata de una diferencia sutil, es importante tenerlo en cuenta de ahora en adelante.

**const JSX = (**

**<div className='myDiv'>**

**<h1>Add a class to this div</h1>**

**</div>**

**);**

**Aprende sobre las etiquetas JSX auto-cerradas**

Hasta ahora, has visto cómo JSX difiere de HTML de una manera clave con el uso de className vs. class para definir clases HTML.

Otra forma importante en la que JSX difiere de HTML está en la idea de la etiqueta de auto-cierre.

En HTML, casi todas las etiquetas tienen una etiqueta de apertura y cierre: <div></div>; la etiqueta de cierre siempre tiene una barra inclinada antes del nombre de la etiqueta que está cerrando. Sin embargo, hay instancias especiales en HTML llamadas “etiquetas auto-cerradas”, o etiquetas que no requieren una etiqueta de apertura y cierre antes de que otra etiqueta pueda comenzar.

Por ejemplo, la etiqueta de salto de línea puede escribirse como <br> o como <br />, pero nunca debe escribirse como <br></br>, ya que no contiene ningún contenido.

En JSX, las reglas son un poco diferentes. Cualquier elemento JSX se puede escribir con una etiqueta de auto-cierre, y cada elemento debe ser cerrado. La etiqueta de salto de línea, por ejemplo, siempre debe escribirse como <br /> para ser un JSX válido que puede ser transpilado. Por otra parte, un <div> puede escribirse como <div /> o <div></div>. La diferencia es que en la primera versión de sintaxis no hay forma de incluir nada en la <div />. Verás en desafíos posteriores que esta sintaxis es útil al renderizar componentes de React.

**const JSX = (**

**<div>**

**<h2>Welcome to React!</h2> <br />**

**<p>Be sure to close all tags!</p>**

**<hr />**

**</div>**

**);**

**Crea un componente funcional sin estado**

Los componentes son el núcleo de React. Todo en React es un componente y aquí aprenderás a crear uno.

Hay dos maneras de crear un componente React. La primera forma es utilizar una función JavaScript. Definir un componente de esta manera crea un componente funcional sin estado. El concepto de estado en una solicitud se abordará en retos posteriores. Por ahora, piensa en un componente sin estado como uno que puede recibir datos y renderizarlos, pero no administra o rastrea los cambios en esos datos. (Cubriremos la segunda manera de crear un componente React en el siguiente desafío.)

Para crear un componente con una función, simplemente escribe una función JavaScript que devuelva ya sea JSX o null. Una cosa importante a tener en cuenta es que React requiere que tu nombre de función comience con una letra mayúscula. Aquí hay un ejemplo de un componente funcional sin estado que asigna una clase HTML en JSX:

**const DemoComponent = function() {**

**return (**

**<div className='customClass' />**

**);**

**};**

Después de ser transpilado, el <div> tendrá una clase CSS de customClass.

Debido a que un componente JSX representa HTML, podrías poner varios componentes juntos para crear una página HTML más compleja. Esta es una de las ventajas clave de la arquitectura de componentes que React proporciona. Te permite componer tu interfaz de usuario de muchos componentes separados y aislados. Esto hace más fácil construir y mantener complejas interfaces de usuario.

**return (**

**<div>Completed challenge!</div>**

**);**

**Crea un componente de React**

La otra forma de definir un componente React es con la sintaxis de clase de ES6 class. En el siguiente ejemplo, Kitten hereda de React.Component:

**class Kitten extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<h1>Hi</h1>**

**);**

**}**

**}**

Esto crea una clase de ES6 Kitten que hereda de la clase React.Component. Así que la clase Kitten ahora tiene acceso a muchas características útiles de React, como el estado local y el ciclo de vida de los "hooks". No te preocupes si aún no estás familiarizado con estos términos, ya que se tratarán con más detalle en los desafíos posteriores. También ten en cuenta que la clase Kitten tiene un constructor definido dentro de ella que llama a super(). Utiliza super() para llamar al constructor de la clase padre, en este caso React.Component. El constructor es un método especial utilizado durante la inicialización de objetos que se crean con la palabra clave class. Es una mejor práctica llamar al constructor de un componente con super, y pasar sus propiedades props a ambos. Esto asegura que el componente esté inicializado correctamente. Por ahora, sepan que es estándar que se incluya este código. Pronto verás otros usos para el constructor, así como las props.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**// Cambia el código debajo de esta línea**

**return (**

**<div>**

**<h1>Hello React!</h1>**

**</div>**

**);**

**// Cambia el código encima de esta línea**

**}**

**};**

**Crear un componente con composición**

Ahora veremos cómo podemos componer múltiples componentes de React juntos. Imagina que estás construyendo una aplicación y has creado tres componentes: un Navbar, Dashboard y Footer.

Para componer estos componentes juntos, se podría crear un componente App parent que renderiza cada uno de estos tres componentes como children. Para renderizar un componente como hijo en un componente React, se incluye el nombre del componente escrito como una etiqueta HTML personalizada en el JSX. Por ejemplo, en el método render se puede escribir:

**return (**

**<App>**

**<Navbar />**

**<Dashboard />**

**<Footer />**

**</App>**

**)**

Cuando React encuentra una etiqueta HTML personalizada que hace referencia a otro componente (un nombre de componente envuelto en < /> como en este ejemplo), renderiza el marcado de ese componente en la ubicación de la etiqueta. Esto debería ilustrar la relación padre/hijo entre el componente App y Navbar, Dashboard, y Footer.

**class ParentComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h1>I am the parent</h1>**

**<ChildComponent />**

**</div>**

**);**

**}**

**}**

**Utiliza React para procesar componentes anidados**

El último desafío mostró una manera simple de organizar dos componentes, pero hay muchas maneras diferentes de organizar componentes con React.

La composición de componentes es una de las características más poderosas de React. Cuando trabajas con React, es importante comenzar a pensar en tu interfaz de usuario en términos de componentes, como el ejemplo App del último desafío. Debes dividir tu UI en sus bloques básicos de construcción, y esas piezas se convierten en los componentes. Esto ayuda a separar el código responsable de la interfaz de usuario del código responsable de manejar la lógica de tu aplicación. Esto puede simplificar enormemente el desarrollo y el mantenimiento de proyectos complejos.

**const TypesOfFruit = () => {**

**return (**

**<div>**

**<h2>Fruits:</h2>**

**<ul>**

**<li>Apples</li>**

**<li>Blueberries</li>**

**<li>Strawberries</li>**

**<li>Bananas</li>**

**</ul>**

**</div>**

**);**

**};**

**const Fruits = () => {**

**return (**

**<div>**

**{ /\* Cambia el código debajo de esta línea \*/ }**

**<TypesOfFruit />**

**{ /\* Cambia el código encima de esta línea \*/ }**

**</div>**

**);**

**};**

**class TypesOfFood extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h1>Types of Food:</h1>**

**{ /\* Cambia el código debajo de esta línea \*/ }**

**<Fruits />**

**{ /\* Cambia el código encima de esta línea \*/ }**

**</div>**

**);**

**}**

**};**

**Compón componentes de React**

A medida que los desafíos continúan utilizando composiciones más complejas con componentes de React y JSX, hay un una cosa importante a tener en cuenta. Renderizar componentes de clase de estilo ES6 dentro de otros componentes es igual que renderizar los componentes simples que usaste en los últimos desafíos. Puedes renderizar elementos JSX, componentes funcionales sin estado y componentes de clase ES6, dentro de otros componentes.

**class Fruits extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h2>Fruits:</h2>**

**<NonCitrus />**

**<Citrus />**

**</div>**

**);**

**}**

**};**

**class TypesOfFood extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h1>Types of Food:</h1>**

**<Fruits />**

**<Vegetables />**

**</div>**

**);**

**}**

**};**

**Renderiza un componente de clase al DOM**

Puede que recuerdes haber usado la API ReactDOM en un desafío anterior para renderizar elementos JSX al DOM. El proceso para renderizar los componentes de React será muy similar. Los últimos desafíos se centraron en los componentes y la composición, por lo que el renderizado se ha realizado tras bambalinas. Sin embargo, ninguna parte de código de React que escribas se procesará en el DOM sin realizar una llamada a la API de ReactDOM.

Aquí va un recordatorio de la sintaxis: ReactDOM.render(componentToRender, targetNode). El primer argumento es el componente de React que deseas renderizar. El segundo argumento es el nodo del DOM en el que deseas renderizar ese componente.

Los componentes de React se pasan a ReactDOM.render() de manera un poco diferente a como se hace con los elementos JSX. Para los elementos JSX, pasas el nombre del elemento que deseas representar. Sin embargo, para los componentes de React, debes usar la misma sintaxis que si estuvieras renderizando un componente anidado, por ejemplo, ReactDOM.render(<ComponentToRender />, targetNode). Se utiliza esta sintaxis tanto para los componentes de clase ES6 como para los componentes funcionales.

**class TypesOfFood extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h1>Types of Food:</h1>**

**{/\* Cambia el código debajo de esta línea \*/}**

**<Fruits />**

**<Vegetables />**

**{/\* Cambia el código encima de esta línea \*/}**

**</div>**

**);**

**}**

**};**

**// Cambia el código debajo de esta línea**

**ReactDOM.render(<TypesOfFood />, document.getElementById("challenge-node"));**

**Escribe un componente React desde cero**

Ahora que has aprendido los conceptos básicos de JSX y componentes React, es el momento de escribir un componente por tu cuenta. Los componentes React son el bloque de construcción principal de las aplicaciones React, por lo que es importante familiarizarse con cómo escribirlos. Recuerda, un componente típico de React es una class ES6 que hereda de React.Component. Tiene un método de render que retorna HTML (de JSX) o null. Esta es la estructura básica de un componente React. Una vez que tengas un buen entendimiento de esto, estarás preparado para empezar a construir proyectos React más complejos.

**// change code below this line**

**class MyComponent extends React.Component{**

**constructor(props){**

**super(props);**

**}**

**render(){**

**return(**

**<div id="challenge-node">**

**<h1>My First React Component!</h1>**

**</div>**

**);**

**}**

**};**

**ReactDOM.render(<MyComponent/>, document.getElementById("challenge-node"));**

**Pasa "props" a un componente funcional sin estado**

Los desafíos anteriores cubrieron varios casos de creación y composición de elementos de JSX, componentes funcionales y componentes de clase estilo ES6 en React. Con estos cimientos, ha llegado la hora de observar otro patrón de uso muy común en React: props. En React, se pueden pasar props, o propiedades a componentes hijos. Digamos, que tienes un componente App que devuelve un componente hijo llamado Welcome, el cual es un componente funcional sin estado. Puedes pasarle una propiedad llamada user a Welcome escribiendo:

**<App>**

**<Welcome user='Mark' />**

**</App>**

Puedes utilizar atributos personalizados de HTML creados por ti y soportados por React para ser pasados por props a tu componente. En este caso, la propiedad creada user es pasada como atributo al componente Welcome. Dado que Welcome es un componente funcional sin estado, tiene acceso a este valor de la siguiente manera:

const Welcome = (props) => <h1>Hello, {props.user}!</h1>

Este valor es llamado props por convención y, cuando se trata de componentes funcionales sin estado, se lo considera como un argumento pasado a una función que retorna JSX. Puedes acceder el valor del argumento en el cuerpo de la función. En los componentes de clase, verás que esto es un poco diferente.

**const CurrentDate = (props) => {**

**return (**

**<div>**

**<p>The current date is: {props.date}</p>**

**</div>**

**);**

**};**

**class Calendar extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h3>What date is it?</h3>**

**<CurrentDate date={Date()} />**

**</div>**

**);**

**}**

**};**

**Pasa un arreglo como "props"**

El último desafío revela cómo pasar información desde un componente padre a un componente hijo como propspropiedades. Este desafío busca demostrar cómo se pueden pasar arreglos como props. Para pasar un arreglo a un elemento JSX, debe ser tratado como JavaScript y envolverlo entre llaves.

**<ParentComponent>**

**<ChildComponent colors={["green", "blue", "red"]} />**

**</ParentComponent>**

El componente hijo entonces tiene acceso a la propiedad del arreglo colors. Los métodos de arreglo, como join()pueden ser usados ​​al acceder a la propiedad.

**const ChildComponent = (props) => <p>{props.colors.join(', ')}</p>**

Esto unirá todos colorslos elementos de la matriz en una cadena separada por comas y producirá: <p>green, blue, red</p>. Más adelante, aprenderemos sobre otros métodos comunes para representar matrices de datos en React.

**const List = props => {**

**return <p>{props.tasks.join(", ")}</p>;**

**};**

**class ToDo extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h1>To Do Lists</h1>**

**<h2>Today</h2>**

**<List tasks={["Walk", "Cook", "Bake"]} />**

**<h2>Tomorrow</h2>**

**<List tasks={["Study", "Code", "Eat"]} />**

**</div>**

**);**

**}**

**}**

**Usa props predeterminados**

React también tiene una opción para establecer props predeterminados. Puede asignar props preestablecidos a un componente como si suceda una propiedad dentro del propio componente y React asigna la prop preestablecida si es necesario. Esto te permite especificar cuál debe ser el valor de una prop si no se proporciona un valor específico. Por ejemplo, si declara MyComponent.defaultProps = { location: 'San Francisco' }, has definido una prop de localización que se establece en la cadena San Francisco, a menos que indiques lo contrario. React asigna props predeterminados si estas no están definidas, pero si pasa el valor nullcomo valor para una prop, este realce null.

**const ShoppingCart = (props) => {**

**return (**

**<div>**

**<h1>Shopping Cart Component</h1>**

**</div>**

**)**

**};**

**// Cambia el código debajo de esta línea**

**ShoppingCart.defaultProps = {**

**items: 0**

**};**

**Reemplazar las "props" predeterminadas**

La capacidad de establecer valores por defecto para las props es una característica útil en React. La manera de reemplazar los accesorios predeterminados es establecer limpiamente los valores de los accesorios para un componente.

**const Items = (props) => {**

**return <h1>Current Quantity of Items in Cart: {props.quantity}</h1>**

**}**

**Items.defaultProps = {**

**quantity: 0**

**}**

**class ShoppingCart extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**{ /\* Cambia el código debajo de esta línea \*/ }**

**return <Items quantity={10}/>**

**{ /\* Cambia el código encima de esta línea \*/ }**

**}**

**};**

**Usa PropTypes para definir las props que esperas**

React proporciona características útiles de validación de tipos para verificar que los componentes reciben las props del tipo correcto. Por ejemplo, tu aplicación hace una llamada a una API para obtener datos que se esperan que sea un arreglo, el cual es pasado al componente como una prop. Puedes establecer propTypestus componentes para que los datos sean de tipo array. Esto arrojará una advertencia útil cuando los datos sean de otro tipo.

Se considera una muy buena práctica definir los propTypescuando se conoce el tipo de una propiedad con antelación. Puedes definir una propiedad propTypespara un componente de la misma manera que define defaultProps. Al hacer esto, se validará que los props de una clave determinada están presentes con un tipo específico. Aquí hay un ejemplo para validar el tipo functionpara una prop llamada handleClick:

MyComponent.propTypes = { handleClick: PropTypes.func.isRequired }

En el ejemplo anterior, la parte de PropTypes.funcverifica que handleClickes una función. Añadir isRequiredle dice a React que handleClickes una propiedad obligatoria para ese componente. Verás una advertencia si no se proporciona esa propiedad. También diez en cuenta que funcrepresenta function. Entre los siete tipos primitivos de JavaScript, functiony boolean(escrito como bool) son los únicos dos que utilizan una ortografía diferente. Además de los tipos primitivos, hay otros tipos disponibles. Por ejemplo, puedes validar si una prop es un elemento React. Por favor, consulte la documentación para todas las opciones.

Nota: A partir de React v15.5.0, PropTypesse importa de manera independiente de React, así:import PropTypes from 'prop-types';

**const Items = (props) => {**

**return <h1>Current Quantity of Items in Cart: {props.quantity}</h1>**

**};**

**// Cambia el código debajo de esta línea**

**Items.propTypes = {**

**quantity: PropTypes.number.isRequired**

**};**

**// Cambia el código encima de esta línea**

**Items.defaultProps = {**

**quantity: 0**

**};**

**class ShoppingCart extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return <Items />**

**}**

**};**

**Acceder a propiedades "props" usando this.props**

Los últimos desafíos cubrieron las formas básicas de pasar propiedades a un componente hijo. Pero, ¿qué pasa si el componente hijo al que se le pasa una propiedad es un componente de clase ES6, en lugar de un componente funcional sin estado? Los componentes de clase ES6 usan una convención un poco diferente para acceder a las propiedades.

Cada vez que se hace referencia a un componente de clase en sí mismo, se utiliza la palabra clave this. Para acceder a las propiedades dentro de un componente de clase, se antepone al código que se utiliza para acceder a él con this. Por ejemplo, si un componente de clase de ES6 tiene una propiedad llamada data, se escribirá {this.props.data}en JSX.

**class App extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**{ /\* Cambia el código debajo de esta línea \*/ }**

**<Welcome name="ulises" />**

**{ /\* Cambia el código encima de esta línea \*/ }**

**</div>**

**);**

**}**

**};**

**class Welcome extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**{ /\* Cambia el código debajo de esta línea \*/ }**

**<p>Hello, <strong>{this.props.name}</strong>!</p>**

**{ /\* Cambia el código encima de esta línea \*/ }**

**</div>**

**);**

**}**

**};**

**Revisa el uso de "props" con componentes funcionales sin estado**

Excepto por el último desafío, has estado pasando props a componentes funcionales sin estado. Estos componentes actúan como funciones puras. Aceptan props como entrada y devuelven la misma vista cada vez que se les pasa el mismo props. Tal vez te estes preguntando qué es estado, y el próximo desafío lo cubrirá con más detalle. Antes de eso, aquí hay una revisión de la terminología de los componentes.

Un componente funcional sin estado , es cualquier función que escribas que acepte props y devuelva JSX. Un componente sin estado , por otra parte, es una clase que se extiende React.Component, pero no usa el estado interno (que será cubierto en el siguiente desafío). Finalmente, un componente con estado es un componente de clase que mantiene su propio estado interno. Puedes ver componentes con estado referido simplemente como componentes de React.

Un patrón común es intentar minimizar el estado y crear componentes funcionales sin estado siempre que sea posible. Esto ayuda a contener su administración de estado a un área específica de su aplicación. A su vez, esto mejora el desarrollo y el mantenimiento de su aplicación al facilitar el seguimiento de cómo los cambios de estado surgen a su comportamiento.

**class CampSite extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<Camper/>**

**</div>**

**);**

**}**

**};**

**// Cambia el código debajo de esta línea**

**const Camper = props => <p>{props.name}</p>;**

**Camper.defaultProps = {**

**name: "CamperBot"**

**};**

**Camper.propTypes = {**

**name: PropTypes.string.isRequired**

**};**

**Crea un componente de estado**

**U**no de los temas más importantes en React es state. El estado consiste en cualquier dato que tu aplicación necesite conocer y que pueda cambiar con el tiempo. Quieres que tus aplicaciones respondan a los cambios de estado y presenten una interfaz de usuario actualizada cuando sea necesario. React ofrece una buena solución para el manejo del estado de las aplicaciones web modernas.

Crea un estado en un componente de React declarando una propiedad stateen la clase del componente en su constructor. Esto inicializa el componente con statecuando se crea. La propiedad statedebe establecerse en un objectJavaScript. Declararlo se ve asi:

**this.state = {**

**}**

Tienes acceso al objeto statea lo largo de la vida de tu componente. Puedes actualizarlo, renderizarlo en tu interfaz de usuario y pasarlo como propiedades a componentes hijos. El objeto statepuede ser tan complejo o simple como lo necesita. Ten en cuenta que debes crear un componente de clase heredando React.Componentpara crear un statecomo este.

**class StatefulComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**// Cambia solo el código debajo de esta línea**

**this.state = {**

**firstName : "ulises"**

**}**

**// Cambia solo el código encima de esta línea**

**}**

**render() {**

**return (**

**<div>**

**<h1>{this.state.firstName}</h1>**

**</div>**

**);**

**}**

**};**

**Renderiza el estado en la interfaz de usuario**

Una vez que se define el estado inicial de un componente, se puede mostrar cualquier parte del mismo en la interfaz de usuario que se renderiza. Si un componente tiene estado, siempre tendrá acceso a los datos en statesu método render(). Puedes acceder a los datos con this.state.

Si quieres acceder a un valor de estado dentro del returnmétodo de renderización, tienes que encerrar el valor entre llaves.

statees una de las características más poderosas de los componentes de React. Esto te permite realizar un seguimiento de los datos importantes en tu aplicación y generar una interfaz de usuario en respuesta a los cambios en estos datos. Si tus datos cambian, tu interfaz de usuario cambiará. React usa lo que se llama un DOM virtual, para realizar un seguimiento de los cambios detrás de escena. Cuando se actualicen los datos de estado, active un re-renderizado de los componentes usando esos datos: incluyendo componentes hijos que recibieron los datos como un prop. React actualiza el DOM actual, pero solo cuando sea necesario. Esto significa que no tienes que preocuparte por cambiar el DOM. Tú declaras simplemente cómo debe ver la interfaz de usuario.

Ten en cuenta que si creas un componente con estado, ningún otro componente es consciente de su state. Su stateestá completamente encapsulado, o local a ese componente, a menos que pase datos de estado a un componente hijo como props. Esta noción de stateencapsulado, es muy importante porque te permite escribir cierta lógica, luego tener esa lógica contenida y aislada en un lugar de tu código.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**name: 'freeCodeCamp'**

**}**

**}**

**render() {**

**return (**

**<div>**

**{ /\* Cambia el código debajo de esta línea \*/ }**

**<h1>{this.state.name}</h1>**

**{ /\* Cambia el código encima de esta línea \*/ }**

**</div>**

**);**

**}**

**};**

**Otra manera de renderizar el estado en la interfaz de usuario**

Hay otra manera de acceder a stateun componente. En el método render(), antes de la sentencia return, se puede escribir JavaScript directamente. Por ejemplo, puede declarar funciones, acceder a datos de stateo props, realizar cálculos sobre estos datos, etc. Luego, puede asignar cualquier dato a las variables, a las que tiene acceso en la sentencia return.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**name: 'freeCodeCamp'**

**}**

**}**

**render() {**

**// Cambia el código debajo de esta línea**

**const name = this.state.name;**

**// Cambia el código encima de esta línea**

**return (**

**<div>**

**{ /\* Cambia el código debajo de esta línea \*/ }**

**<h1>{name}</h1>**

**{ /\* Cambia el código encima de esta línea \*/ }**

**</div>**

**);**

**}**

**};**

**Establecer estado con this.setState**

Los desafíos anteriores cubrieron el componente statey cómo inicializar el estado en constructor. También hay una manera de cambiar el componente state. React proporciona un método para actualizar el componente statellamado setState. Llamas al setStatemétodo dentro de tu clase de componente así: this.setState(), pasando un objeto con pares clave-valor. Las claves son las propiedades de su estado y los valores son los datos de estado actualizados. Por ejemplo, si estuviéramos almacenando un usernameestado y quisiéramos actualizarlo, se vería así:

**this.setState({**

**username: 'Lewis'**

**});**

React espera que nunca modifiques statedirectamente, sino que lo utilices siempre this.setState()cuando se produzcan cambios de estado. Además, debe tener en cuenta que React puede agrupar varias actualizaciones de estado para mejorar el rendimiento. Lo que esto significa es que las actualizaciones de estado a través del setStatemétodo pueden ser asincrónicas. Existe una sintaxis alternativa para el setStatemétodo que proporciona una solución a este problema. Esto rara vez es necesario, ¡pero es bueno tenerlo en cuenta! Consulte nuestro artículo de React para obtener más detalles.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**name: 'Initial State'**

**};**

**this.handleClick = this.handleClick.bind(this);**

**}**

**handleClick() {**

**// Change code below this line**

**this.setState({**

**name: "React Rocks!"**

**});**

**// Change code above this line**

**}**

**render() {**

**return (**

**<div>**

**<button onClick={this.handleClick}>Click Me</button>**

**<h1>{this.state.name}</h1>**

**</div>**

**);**

**}**

**};**

**Vincular 'esto' a un método de clase**

Además de configurar y actualizar state, también puede definir métodos para su clase de componente. Un método de clase normalmente necesita usar la thispalabra clave para poder acceder a las propiedades de la clase (como statey props) dentro del alcance del método. Hay algunas formas de permitir el acceso a los métodos de su clase this.

Una forma común es vincularse explícitamente thisen el constructor para que thisquede vinculado a los métodos de la clase cuando se inicializa el componente. Es posible que hayas notado el último desafío utilizado this.handleClick = this.handleClick.bind(this)para su handleClickmétodo en el constructor. Luego, cuando llamas a una función como this.setState()dentro de tu método de clase, thisse refiere a la clase y no será undefined.

Nota: La thispalabra clave es uno de los aspectos más confusos de JavaScript, pero juega un papel importante en React. Aunque su comportamiento aquí es totalmente normal, estas lecciones no son el lugar para una revisión en profundidad, thisasí que consulte otras lecciones si lo anterior le resulta confuso.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**text: "Hello"**

**};**

**// Change code below this line**

**this.handleClick = this.handleClick.bind(this);**

**// Change code above this line**

**}**

**handleClick() {**

**this.setState({**

**text: "You clicked!"**

**});**

**}**

**render() {**

**return (**

**<div>**

**{ /\* Change code below this line \*/ }**

**<button onClick = {this.handleClick}>Click Me</button>**

**{ /\* Change code above this line \*/ }**

**<h1>{this.state.text}</h1>**

**</div>**

**);**

**}**

**};**

**Usar estado para alternar un elemento**

A veces es posible que necesite conocer el estado anterior al actualizar el estado. Sin embargo, las actualizaciones de estado pueden ser asincrónicas; esto significa que React puede agrupar varias setState()llamadas en una sola actualización. Esto significa que no puede confiar en el valor anterior this.stateni this.propsal calcular el siguiente valor. Por lo tanto, no deberías utilizar código como este:

**this.setState({**

**counter: this.state.counter + this.props.increment**

**});**

En su lugar, debes pasar setStateuna función que te permita acceder al estado y a los accesorios. Al utilizar una función con setStategarantías, está trabajando con los valores de estado y accesorios más actuales. Esto significa que lo anterior debería reescribirse como:

**this.setState((state, props) => ({**

**counter: state.counter + props.increment**

**}));**

También puede utilizar un formulario sin propssi solo necesita state:

**this.setState(state => ({**

**counter: state.counter + 1**

**}));**

Tenga en cuenta que debe envolver el objeto literal entre paréntesis; de lo contrario, JavaScript cree que es un bloque de código.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**visibility: false**

**};**

**this.toggleVisibility = this.toggleVisibility.bind(this);**

**}**

**toggleVisibility() {**

**this.setState(state => ({**

**visibility: !state.visibility**

**}));**

**}**

**render() {**

**if (this.state.visibility) {**

**return (**

**<div>**

**<button onClick = {this.toggleVisibility}>Click Me</button>**

**<h1>Now you see me!</h1>**

**</div>**

**);**

**} else {**

**return (**

**<div>**

**<button onClick = {this.toggleVisibility}>Click Me</button>**

**</div>**

**);**

**}**

**}**

**};**

**Escribe un contador simple**

Puede diseñar un componente con estado más complejo combinando los conceptos tratados hasta ahora. Estos incluyen inicializar state, escribir métodos que configuran statey asignar controladores de clic para activar estos métodos.

**class Counter extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**count: 0**

**};**

**this.increment = this.increment.bind(this);**

**this.decrement = this.decrement.bind(this);**

**this.reset = this.reset.bind(this);**

**}**

**reset() {**

**this.setState({**

**count: 0**

**});**

**}**

**increment() {**

**this.setState(state => ({**

**count: state.count + 1**

**}));**

**}**

**decrement() {**

**this.setState(state => ({**

**count: state.count - 1**

**}));**

**}**

**render() {**

**return (**

**<div>**

**<button className='inc' onClick={this.increment}>Increment!</button>**

**<button className='dec' onClick={this.decrement}>Decrement!</button>**

**<button className='reset' onClick={this.reset}>Reset</button>**

**<h1>Current Count: {this.state.count}</h1>**

**</div>**

**);**

**}**

**};**

**Crear una entrada controlada**

Su aplicación puede tener interacciones más complejas entre statela interfaz de usuario renderizada. Por ejemplo, los elementos de control de formulario para la entrada de texto, como inputy textarea, mantienen su propio estado en el DOM a medida que el usuario escribe. Con React, puedes mover este estado mutable al estado mutable de un componente de React state. La entrada del usuario pasa a formar parte de la aplicación state, por lo que React controla el valor de ese campo de entrada. Normalmente, si tiene componentes de React con campos de entrada en los que el usuario puede escribir, será un formulario de entrada controlado.

**class ControlledInput extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**input: ''**

**};**

**// Change code below this line**

**this.handleChange = this.handleChange.bind(this);**

**// Change code above this line**

**}**

**// Change code below this line**

**handleChange (event) {**

**this.setState({**

**input: event.target.value**

**})**

**}**

**// Change code above this line**

**render() {**

**return (**

**<div>**

**{ /\* Change code below this line \*/}**

**<input value={this.state.input} onChange={this.handleChange} />**

**{ /\* Change code above this line \*/}**

**<h4>Controlled Input:</h4>**

**<p>{this.state.input}</p>**

**</div>**

**);**

**}**

**};**

**Crear un formulario controlado**

El último desafío demostró que React puede controlar el estado interno de ciertos elementos como inputy textarea, lo que los convierte en componentes controlados. Esto también se aplica a otros elementos del formulario, incluido el formelemento HTML normal.

**class MyForm extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**input: '',**

**submit: ''**

**};**

**this.handleChange = this.handleChange.bind(this);**

**this.handleSubmit = this.handleSubmit.bind(this);**

**}**

**handleChange(event) {**

**this.setState({**

**input: event.target.value**

**});**

**}**

**handleSubmit(event) {**

**event.preventDefault()**

**this.setState({**

**submit: this.state.input**

**});**

**}**

**render() {**

**return (**

**<div>**

**<form onSubmit={this.handleSubmit}>**

**<input**

**value={this.state.input}**

**onChange={this.handleChange} />**

**<button type='submit'>Submit!</button>**

**</form>**

**<h1>{this.state.submit}</h1>**

**</div>**

**);**

**}**

**};**

**Pasar estado como accesorios para componentes secundarios**

Viste muchos ejemplos que pasaron accesorios a elementos JSX secundarios y componentes secundarios de React en desafíos anteriores. Quizás se pregunte de dónde vienen esos accesorios. Un patrón común es tener un componente con estado que contenga lo stateimportante para su aplicación, que luego representa los componentes secundarios. Quiere que estos componentes tengan acceso a algunas partes de eso state, que se pasan como accesorios.

Por ejemplo, tal vez tenga un Appcomponente que represente un Navbar, entre otros componentes. En su App, tiene stateuna gran cantidad de información del usuario, pero solo Navbarnecesita acceso al nombre de usuario del usuario para poder mostrarlo. Pasas esa pieza stateal Navbarcomponente como accesorio.

Este patrón ilustra algunos paradigmas importantes en React. El primero es el flujo de datos unidireccional.. El estado fluye en una dirección hacia abajo en el árbol de componentes de su aplicación, desde el componente principal con estado hasta los componentes secundarios. Los componentes secundarios solo reciben los datos de estado que necesitan. La segunda es que las aplicaciones con estado complejas se pueden dividir en unos pocos componentes con estado, o tal vez en uno solo. El resto de sus componentes simplemente reciben el estado del padre como accesorios y representan una interfaz de usuario desde ese estado. Comienza a crear una separación donde la gestión del estado se maneja en una parte del código y la representación de la interfaz de usuario en otra. Este principio de separar la lógica de estado de la lógica de la interfaz de usuario es uno de los principios clave de React. Cuando se usa correctamente, hace que el diseño de aplicaciones complejas y con estado sea mucho más fácil de administrar.

**class MyApp extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**name: "CamperBot"**

**};**

**}**

**render() {**

**return (**

**<div>**

**// Here we will call this.state.name in order to pass the value of**

**CamperBot // to the NavBar component**

**<Navbar name={this.state.name} />**

**</div>**

**);**

**}**

**}**

**class Navbar extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**// Since we passed in the CamperBot state value into the the NavBar**

**component above // the h1 element below will render the value passed**

**from state**

**<h1>Hello, my name is: {this.props.name}</h1>**

**</div>**

**);**

**}**

**}**

**Pasar una devolución de llamada como accesorios**

Puede pasar stateaccesorios a componentes secundarios, pero no está limitado a pasar datos. También puede pasar funciones de controlador o cualquier método definido en un componente de React a un componente secundario. Así es como permite que los componentes secundarios interactúen con sus componentes principales. Le pasas métodos a un niño como si fuera un accesorio normal. Se le asigna un nombre y usted tiene acceso al nombre de ese método this.propsen el componente secundario.

**class MyApp extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**inputValue: ''**

**}**

**this.handleChange = this.handleChange.bind(this);**

**}**

**handleChange(event) {**

**this.setState({**

**inputValue: event.target.value**

**});**

**}**

**render() {**

**return (**

**<div>**

**<GetInput**

**input={this.state.inputValue}**

**handleChange={this.handleChange}/>**

**<RenderInput**

**input={this.state.inputValue}/>**

**</div>**

**);**

**}**

**};**

**class GetInput extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h3>Get Input:</h3>**

**<input**

**value={this.props.input}**

**onChange={this.props.handleChange}/>**

**</div>**

**);**

**}**

**};**

**class RenderInput extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h3>Input Render:</h3>**

**<p>{this.props.input}</p>**

**</div>**

**);**

**}**

**};**

**Usar el componente Método de ciclo de vidaWillMount**

Los componentes de React tienen varios métodos especiales que brindan oportunidades para realizar acciones en puntos específicos del ciclo de vida de un componente. Estos se denominan métodos de ciclo de vida o ganchos de ciclo de vida y le permiten capturar componentes en ciertos puntos en el tiempo. Esto puede ser antes de que se rendericen, antes de que se actualicen, antes de que reciban accesorios, antes de que se desmonten, etc. Aquí hay una lista de algunos de los principales métodos de ciclo de vida: componentWillMount() componentDidMount() shouldComponentUpdate() componentDidUpdate() componentWillUnmount()Las próximas lecciones cubrirán algunos de los casos de uso básicos para estos métodos de ciclo de vida.

Nota: El componentWillMountmétodo Lifecycle quedará obsoleto en una versión futura de 16.X y se eliminará en la versión 17. Obtenga más información en este artículo

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**componentWillMount() {**

**console.log('Component being mounted');**

**}**

**render() {**

**return <div />**

**}**

**};**

**Usar el método de ciclo de vida componenteDidMount**

La mayoría de los desarrolladores web, en algún momento, necesitan llamar a un punto final API para recuperar datos. Si está trabajando con React, es importante saber dónde realizar esta acción.

La mejor práctica con React es realizar llamadas API o cualquier llamada a su servidor en el método del ciclo de vida componentDidMount(). Este método se llama después de montar un componente en el DOM. Cualquier llamada setState()aquí activará una nueva representación de su componente. Cuando llama a una API en este método y establece su estado con los datos que devuelve la API, se activará automáticamente una actualización una vez que reciba los datos.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**activeUsers: null**

**};**

**}**

**componentDidMount() {**

**setTimeout(() => {**

**this.setState({**

**activeUsers: 1273**

**});**

**}, 2500);**

**}**

**render() {**

**return (**

**<div>**

**{/\* Change code below this line \*/}**

**<h1>Active Users: { this.state.activeUsers }</h1>**

**{/\* Change code above this line \*/}**

**</div>**

**);**

**}**

**}**

**Agregar oyentes de eventos**

El componentDidMount()método también es el mejor lugar para adjuntar cualquier detector de eventos que necesite agregar para una funcionalidad específica. React proporciona un sistema de eventos sintético que envuelve el sistema de eventos nativo presente en los navegadores. Esto significa que el sistema de eventos sintéticos se comporta exactamente igual independientemente del navegador del usuario, incluso si los eventos nativos pueden comportarse de manera diferente entre diferentes navegadores.

Ya has estado usando algunos de estos controladores de eventos sintéticos, como onClick(). El sistema de eventos sintéticos de React es excelente para usar en la mayoría de las interacciones que administrará en elementos DOM. Sin embargo, si desea adjuntar un controlador de eventos al documento o a los objetos de la ventana, debe hacerlo directamente.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**message: ""**

**};**

**this.handleEnter = this.handleEnter.bind(this);**

**this.handleKeyPress = this.handleKeyPress.bind(this);**

**}**

**// change code below this line**

**componentDidMount() {**

**document.addEventListener("keydown", this.handleKeyPress);**

**}**

**componentWillUnmount() {**

**document.removeEventListener("keydown", this.handleKeyPress);**

**}**

**// change code above this line**

**handleEnter() {**

**this.setState((state) => ({**

**message: state.message + 'You pressed the enter key! '**

**}));**

**}**

**handleKeyPress(event) {**

**if (event.keyCode === 13) {**

**this.handleEnter();**

**}**

**}**

**render() {**

**return (**

**<div>**

**<h1>{this.state.message}</h1>**

**</div>**

**);**

**}**

**}**

**Optimice los renderizados con shouldComponentUpdate**

Hasta ahora, si algún componente recibe new stateo new props, se vuelve a renderizar a sí mismo y a todos sus hijos. Esto suele estar bien. Pero React proporciona un método de ciclo de vida al que puede llamar cuando los componentes secundarios reciben nuevos stateo propsy declarar específicamente si los componentes deben actualizarse o no. El método es shouldComponentUpdate(), y toma nextPropsy nextStatecomo parámetros.

Este método es una forma útil de optimizar el rendimiento. Por ejemplo, el comportamiento predeterminado es que su componente se vuelve a representar cuando recibe información nueva props, incluso si propsno ha cambiado. Puede utilizar shouldComponentUpdate()para evitar esto comparando el archivo props. El método debe devolver un booleanvalor que le indique a React si actualizar o no el componente. Puede comparar los accesorios actuales ( this.props) con los siguientes accesorios ( nextProps) para determinar si necesita actualizar o no, y regresar trueo falseen consecuencia.

El shouldComponentUpdate()método se agrega en un componente llamado OnlyEvens. Actualmente, este método regresa true, por lo que OnlyEvensse vuelve a representar cada vez que recibe nuevos archivos props. Modifique el método para que OnlyEvensse actualice solo si el valor valuede sus nuevos accesorios es par. Haga clic en el Addbotón y observe el orden de los eventos en la consola de su navegador a medida que se activan los enlaces del ciclo de vida.

**class OnlyEvens extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**shouldComponentUpdate(nextProps, nextState) {**

**console.log('Should I update?');**

**// change code below this line**

**if (nextProps.value % 2 == 0) {**

**return true;**

**}**

**return false;**

**// change code above this line**

**}**

**componentWillReceiveProps(nextProps) {**

**console.log('Receiving new props...');**

**}**

**componentDidUpdate() {**

**console.log('Component re-rendered.');**

**}**

**render() {**

**return <h1>{this.props.value}</h1>**

**}**

**};**

**class Controller extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**value: 0**

**};**

**this.addValue = this.addValue.bind(this);**

**}**

**addValue() {**

**this.setState({**

**value: this.state.value + 1**

**});**

**}**

**render() {**

**return (**

**<div>**

**<button onClick={this.addValue}>Add</button>**

**<OnlyEvens value={this.state.value}/>**

**</div>**

**);**

**}**

**};**

**Introducción a los estilos en línea**

Hay otros conceptos complejos que agregan capacidades poderosas a su código React. Pero quizás te preguntes sobre el problema más simple de cómo diseñar esos elementos JSX que creas en React. Probablemente sepas que no será exactamente lo mismo que trabajar con HTML debido a la forma en que aplicas las clases a los elementos JSX .

Si importa estilos desde una hoja de estilo, no es muy diferente. Aplica una clase a su elemento JSX usando el classNameatributo y aplica estilos a la clase en su hoja de estilo. Otra opción es aplicar estilos en línea, que son muy comunes en el desarrollo de ReactJS.

Aplica estilos en línea a elementos JSX de manera similar a como lo hace en HTML, pero con algunas diferencias JSX. A continuación se muestra un ejemplo de un estilo en línea en HTML:

<div style="color: yellow; font-size: 16px">Mellow Yellow</div>

Los elementos JSX usan el styleatributo, pero debido a la forma en que se transpila JSX, no se puede establecer el valor en string. En su lugar, lo estableces igual a JavaScript object. Aquí hay un ejemplo:

<div style={{color: "yellow", fontSize: 16}}>Mellow Yellow</div>

¿Observas cómo hacemos CamelCase en la fontSizepropiedad? Esto se debe a que React no aceptará claves de kebab-case en el objeto de estilo. React nos aplicará el nombre de propiedad correcto en el HTML.

**class Colorful extends React.Component {**

**render() {**

**return (**

**<div style={{ color: 'red', fontSize: '72'}}>**

**Big Red**

**</div>**

**);**

**}**

**};**

**Agregar estilos en línea en React**

Es posible que hayas notado en el último desafío que había otras diferencias de sintaxis con respecto a los estilos en línea HTML además del styleatributo establecido en un objeto JavaScript. Primero, los nombres de ciertas propiedades de estilo CSS usan mayúsculas y minúsculas. Por ejemplo, el último desafío estableció el tamaño de la fuente en fontSizelugar de font-size. Las palabras con guiones como font-sizetienen una sintaxis no válida para las propiedades de los objetos JavaScript, por lo que React usa mayúsculas y minúsculas. Como regla general, cualquier propiedad de estilo con guiones se escribe usando mayúsculas y minúsculas en JSX.

Se supone que todas las unidades de longitud del valor de propiedad (como height, widthy fontSize) están en pxa menos que se especifique lo contrario. Si desea utilizar em, por ejemplo, coloque el valor y las unidades entre comillas, como {fontSize: "4em"}. Aparte de los valores de longitud predeterminados px, todos los demás valores de propiedad deben estar entre comillas.

**const styles = {**

**color: 'purple',**

**fontSize: 40,**

**border: "2px solid purple",**

**};**

**class Colorful extends React.Component {**

**render() {**

**// change code below this line**

**return (**

**<div style={styles}>Style Me!</div>**

**);**

**// change code above this line**

**}**

**};**

**Renderizar con una condición If-Else**

Otra aplicación del uso de JavaScript para controlar la vista renderizada es vincular los elementos que se renderizan a una condición. Cuando la condición es verdadera, se representa una vista. Cuando es falso, es una visión diferente. Puede hacer esto con una if/elsedeclaración estándar en el render()método de un componente de React.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**display: true**

**}**

**this.toggleDisplay = this.toggleDisplay.bind(this);**

**}**

**toggleDisplay() {**

**this.setState({**

**display: !this.state.display**

**});**

**}**

**render() {**

**// change code below this line**

**if (this.state.display) {**

**return (**

**<div>**

**<button onClick={this.toggleDisplay}>Toggle Display</button>**

**<h1>Displayed!</h1>**

**</div>**

**);**

**} else {**

**return (**

**<div>**

**<button onClick={this.toggleDisplay}>Toggle Display</button>**

**</div>**

**);**

**}**

**}**

**};**

**Use JavaScript avanzado en el método React Render**

En desafíos anteriores, aprendió a inyectar código JavaScript en código JSX usando llaves, { }para tareas como acceder a accesorios, pasar accesorios, acceder al estado, insertar comentarios en su código y, más recientemente, diseñar sus componentes. Todos estos son casos de uso comunes para poner JavaScript en JSX, pero no son la única forma en que puede utilizar código JavaScript en sus componentes de React.

También puede escribir JavaScript directamente en sus rendermétodos, antes de la returndeclaración, sin insertarlo entre llaves. Esto se debe a que aún no está dentro del código JSX. Cuando desee utilizar una variable más adelante en el código JSX dentro de la returndeclaración, coloque el nombre de la variable entre llaves.

**const inputStyle = {**

**width: 235,**

**margin: 5**

**};**

**class MagicEightBall extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**userInput: '',**

**randomIndex: ''**

**};**

**this.ask = this.ask.bind(this);**

**this.handleChange = this.handleChange.bind(this);**

**}**

**ask() {**

**if (this.state.userInput) {**

**this.setState({**

**randomIndex: Math.floor(Math.random() \* 20),**

**userInput: ''**

**});**

**}**

**}**

**handleChange(event) {**

**this.setState({**

**userInput: event.target.value**

**});**

**}**

**render() {**

**const possibleAnswers = [**

**'It is certain',**

**'It is decidedly so',**

**'Without a doubt',**

**'Yes, definitely',**

**'You may rely on it',**

**'As I see it, yes',**

**'Outlook good',**

**'Yes',**

**'Signs point to yes',**

**'Reply hazy try again',**

**'Ask again later',**

**'Better not tell you now',**

**'Cannot predict now',**

**'Concentrate and ask again',**

**"Don't count on it",**

**'My reply is no',**

**'My sources say no',**

**'Most likely',**

**'Outlook not so good',**

**'Very doubtful'**

**];**

**const answer = possibleAnswers[this.state.randomIndex];**

**return (**

**<div>**

**<input**

**type='text'**

**value={this.state.userInput}**

**onChange={this.handleChange}**

**style={inputStyle}**

**/>**

**<br />**

**<button onClick={this.ask}>Ask the Magic Eight Ball!</button>**

**<br />**

**<h3>Answer:</h3>**

**<p>**

**{answer}**

**</p>**

**</div>**

**);**

**}**

**}**

**Utilice && para un condicional más conciso**

Las if/elsedeclaraciones funcionaron en el último desafío, pero hay una forma más concisa de lograr el mismo resultado. Imagine que está rastreando varias condiciones en un componente y desea que se representen diferentes elementos dependiendo de cada una de estas condiciones. Si escribe muchas else ifdeclaraciones para devolver UI ligeramente diferentes, puede repetir el código, lo que deja margen de error. En su lugar, puede utilizar el &&operador lógico para realizar la lógica condicional de una forma más concisa. Esto es posible porque desea verificar si una condición es truey, si lo es, devolver algún marcado. He aquí un ejemplo:

**{condition && <p>markup</p>}**

Si conditiones true, se devolverá el marcado. Si la condición es false, la operación regresará inmediatamente falsedespués de evaluar conditiony no devolverá nada. Puede incluir estas declaraciones directamente en su JSX y encadenar varias condiciones escribiendo &&después de cada una. Esto le permite manejar una lógica condicional más compleja en su render()método sin repetir mucho código.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**display: true**

**}**

**this.toggleDisplay = this.toggleDisplay.bind(this);**

**}**

**toggleDisplay() {**

**this.setState(state => ({**

**display: !state.display**

**}));**

**}**

**render() {**

**return (**

**<div>**

**<button onClick={this.toggleDisplay}>Toggle Display</button>**

**{this.state.display &&<h1>Displayed!</h1>}**

**</div>**

**);**

**}**

**};**

**Utilice una expresión ternaria para la representación condicional**

Antes de pasar a las técnicas de representación dinámica, hay una última forma de utilizar los condicionales integrados de JavaScript para representar lo que desea: el operador ternario . El operador ternario se utiliza a menudo como atajo para if/elsedeclaraciones en JavaScript. No son tan sólidas como if/elselas declaraciones tradicionales, pero son muy populares entre los desarrolladores de React. Una razón para esto es que, debido a cómo se compila JSX, if/elselas declaraciones no se pueden insertar directamente en el código JSX. Es posible que hayas notado esto hace un par de desafíos: cuando if/elsese requería una declaración, siempre estaba fuera delreturndeclaración. Las expresiones ternarias pueden ser una excelente alternativa si deseas implementar lógica condicional dentro de tu JSX. Recuerde que un operador ternario tiene tres partes, pero puede combinar varias expresiones ternarias. Aquí está la sintaxis básica:

**condition ? expressionIfTrue : expressionIfFalse;**

El editor de código tiene tres constantes definidas dentro del método CheckUserAgedel componente render(). Se llaman buttonOne, buttonTwoy buttonThree. A cada uno de ellos se le asigna una expresión JSX simple que representa un elemento de botón. Primero, inicialice el estado de CheckUserAgewith inputy userAgeambos establezcan valores de una cadena vacía.

Una vez que el componente presenta información a la página, los usuarios deberían tener una forma de interactuar con él. Dentro de la declaración del componente return, configure una expresión ternaria que implemente la siguiente lógica: cuando la página se carga por primera vez, muestre el botón de envío, buttonOneen la página. Luego, cuando un usuario ingresa su edad y hace clic en el botón, se genera un botón diferente según la edad. Si un usuario ingresa un número menor que 18, renderice buttonThree. Si un usuario ingresa un número mayor o igual a 18, renderiza buttonTwo.

**const inputStyle = {**

**width: 235,**

**margin: 5**

**};**

**class CheckUserAge extends React.Component {**

**constructor(props) {**

**super(props);**

**// Change code below this line**

**this.state = {**

**userAge: '',**

**input: ''**

**}**

**// Change code above this line**

**this.submit = this.submit.bind(this);**

**this.handleChange = this.handleChange.bind(this);**

**}**

**handleChange(e) {**

**this.setState({**

**input: e.target.value,**

**userAge: ''**

**});**

**}**

**submit() {**

**this.setState(state => ({**

**userAge: state.input**

**}));**

**}**

**render() {**

**const buttonOne = <button onClick={this.submit}>Submit</button>;**

**const buttonTwo = <button>You May Enter</button>;**

**const buttonThree = <button>You Shall Not Pass</button>;**

**return (**

**<div>**

**<h3>Enter Your Age to Continue</h3>**

**<input**

**style={inputStyle}**

**type='number'**

**value={this.state.input}**

**onChange={this.handleChange}**

**/>**

**<br />**

**{/\* Change code below this line \*/}**

**{**

**this.state.userAge === ''**

**? buttonOne**

**: this.state.userAge >= 18**

**? buttonTwo**

**: buttonThree**

**}**

**{/\* Change code above this line \*/}**

**</div>**

**);**

**}**

**}**

**Renderizar condicionalmente a partir de accesorios**

Hasta ahora, has visto cómo usar if/else, &&y el operador ternario ( condition ? expressionIfTrue : expressionIfFalse) para tomar decisiones condicionales sobre qué renderizar y cuándo. Sin embargo, queda un tema importante por discutir que le permite combinar cualquiera o todos estos conceptos con otra característica poderosa de React: los accesorios. El uso de accesorios para renderizar código condicionalmente es muy común entre los desarrolladores de React; es decir, utilizan el valor de un accesorio determinado para tomar decisiones automáticamente sobre qué renderizar.

En este desafío, configurará un componente secundario para tomar decisiones de representación basadas en accesorios. También utilizará el operador ternario, pero podrá ver cómo varios de los otros conceptos que se cubrieron en los últimos desafíos podrían ser igualmente útiles en este contexto.

**class Results extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<h1>**

**{this.props.fiftyFifty ? "You Win!" : "You Lose!"}**

**</h1>**

**)**

**};**

**};**

**class GameOfChance extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**counter: 1**

**}**

**this.handleClick = this.handleClick.bind(this);**

**}**

**handleClick() {**

**this.setState({**

**counter: this.state.counter + 1 // change code here**

**});**

**}**

**render() {**

**const expression = Math.random() >= 0.5;**

**return (**

**<div>**

**<button onClick={this.handleClick}>Play Again</button>**

**{ /\* change code below this line \*/ }**

**<Results fiftyFifty={expression} />**

**{ /\* change code above this line \*/ }**

**<p>{'Turn: ' + this.state.counter}</p>**

**</div>**

**);**

**}**

**};**

**Cambiar CSS en línea condicionalmente según el estado del componente**

Hasta este punto, has visto varias aplicaciones de renderizado condicional y el uso de estilos en línea. Aquí hay un ejemplo más que combina ambos temas. También puede renderizar CSS condicionalmente según el estado de un componente de React. Para hacer esto, verifica una condición y, si esa condición se cumple, modifica el objeto de estilos asignado a los elementos JSX en el método de representación.

Es importante comprender este paradigma porque supone un cambio espectacular con respecto al enfoque más tradicional de aplicar estilos modificando elementos DOM directamente (lo cual es muy común con jQuery, por ejemplo). En ese enfoque, debe realizar un seguimiento de cuándo cambian los elementos y también manejar la manipulación real directamente. Puede resultar difícil realizar un seguimiento de los cambios, lo que puede hacer que su interfaz de usuario sea impredecible. Cuando configura un objeto de estilo en función de una condición, describe cómo debería verse la interfaz de usuario en función del estado de la aplicación. Hay un flujo claro de información que sólo se mueve en una dirección. Este es el método preferido al escribir aplicaciones con React.

**class GateKeeper extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**input: ''**

**};**

**this.handleChange = this.handleChange.bind(this);**

**}**

**handleChange(event) {**

**this.setState({ input: event.target.value })**

**}**

**render() {**

**let inputStyle = {**

**border: '1px solid black'**

**};**

**// Change code below this line**

**if (this.state.input.length > 15) {**

**inputStyle.border = '3px solid red';**

**}**

**// Change code above this line**

**return (**

**<div>**

**<h3>Don't Type Too Much:</h3>**

**<input**

**type="text"**

**style={inputStyle}**

**value={this.state.input}**

**onChange={this.handleChange} />**

**</div>**

**);**

**}**

**};**

**Utilice Array.map() para representar elementos dinámicamente**

La representación condicional es útil, pero es posible que necesite que sus componentes representen una cantidad desconocida de elementos. A menudo, en la programación reactiva, un programador no tiene forma de saber cuál es el estado de una aplicación hasta el tiempo de ejecución, porque mucho depende de la interacción del usuario con ese programa. Los programadores deben escribir su código para manejar correctamente ese estado desconocido con anticipación. El uso Array.map()en React ilustra este concepto.

Por ejemplo, crea una aplicación sencilla de "Lista de tareas pendientes". Como programador, no tienes forma de saber cuántos elementos puede tener un usuario en su lista. Debe configurar su componente para representar dinámicamente la cantidad correcta de elementos de la lista mucho antes de que alguien que use el programa decida que hoy es el día de lavar la ropa.

**const textAreaStyles = {**

**width: 235,**

**margin: 5**

**};**

**class MyToDoList extends React.Component {**

**constructor(props) {**

**super(props);**

**// Change code below this line**

**this.state = {**

**userInput: '',**

**toDoList: []**

**}**

**// Change code above this line**

**this.handleSubmit = this.handleSubmit.bind(this);**

**this.handleChange = this.handleChange.bind(this);**

**}**

**handleSubmit() {**

**const itemsArray = this.state.userInput.split(',');**

**this.setState({**

**toDoList: itemsArray**

**});**

**}**

**handleChange(e) {**

**this.setState({**

**userInput: e.target.value**

**});**

**}**

**render() {**

**const items = this.state.toDoList.map(i => <li>{i}</li>);**

**return (**

**<div>**

**<textarea**

**onChange={this.handleChange}**

**value={this.state.userInput}**

**style={textAreaStyles}**

**placeholder='Separate Items With Commas'**

**/>**

**<br />**

**<button onClick={this.handleSubmit}>Create List</button>**

**<h1>My "To Do" List:</h1>**

**<ul>{items}</ul>**

**</div>**

**);**

**}**

**}**

**Dar a los elementos hermanos un atributo clave único**

El último desafío mostró cómo mapse utiliza el método para representar dinámicamente una serie de elementos según la entrada del usuario. Sin embargo, en ese ejemplo faltaba una pieza importante. Cuando crea una matriz de elementos, cada uno necesita un keyatributo establecido en un valor único. React usa estas claves para realizar un seguimiento de qué elementos se agregan, cambian o eliminan. Esto ayuda a que el proceso de reproducción sea más eficiente cuando la lista se modifica de alguna manera.

Nota: Las claves solo deben ser únicas entre elementos hermanos, no es necesario que sean globalmente únicas en su aplicación.

**const frontEndFrameworks = [**

**'React',**

**'Angular',**

**'Ember',**

**'Knockout',**

**'Backbone',**

**'Vue'**

**];**

**function Frameworks() {**

**const renderFrameworks = frontEndFrameworks.map((item) =>**

**<li key={item}>{item}</li>**

**);**

**return (**

**<div>**

**<h1>Popular Front End JavaScript Frameworks</h1>**

**<ul>**

**{renderFrameworks}**

**</ul>**

**</div>**

**);**

**};**

**Use Array.filter() para filtrar dinámicamente una matriz**

El mapmétodo de matriz es una herramienta poderosa que utilizará con frecuencia cuando trabaje con React. Otro método relacionado mapes filter, que filtra el contenido de una matriz según una condición y luego devuelve una nueva matriz. Por ejemplo, si tiene una serie de usuarios que tienen una propiedad onlineque se puede establecer en trueo false, puede filtrar solo aquellos usuarios que están en línea escribiendo:

**let onlineUsers = users.filter(user => user.online);**

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**users: [**

**{**

**username: 'Jeff',**

**online: true**

**},**

**{**

**username: 'Alan',**

**online: false**

**},**

**{**

**username: 'Mary',**

**online: true**

**},**

**{**

**username: 'Jim',**

**online: false**

**},**

**{**

**username: 'Sara',**

**online: true**

**},**

**{**

**username: 'Laura',**

**online: true**

**}**

**]**

**}**

**}**

**render() {**

**const usersOnline = this.state.users.filter(user => user.online === true); // change code here**

**const renderOnline = usersOnline.map(user => <li key={user.username}>{user.username}</li>); // change code here**

**return (**

**<div>**

**<h1>Current Online Users:</h1>**

**<ul>**

**{renderOnline}**

**</ul>**

**</div>**

**);**

**}**

**};**

**Render React en el servidor con renderToString**

Hasta ahora, has estado renderizando componentes de React en el cliente. Normalmente, esto es lo que siempre harás. Sin embargo, hay algunos casos de uso en los que tiene sentido renderizar un componente de React en el servidor. Dado que React es una biblioteca de vistas de JavaScript y puede ejecutar JavaScript en el servidor con Node, esto es posible. De hecho, React proporciona un renderToString()método que puedes utilizar para este propósito.

Hay dos razones clave por las que la renderización en el servidor se puede utilizar en una aplicación del mundo real. Primero, sin hacer esto, sus aplicaciones React consistirían en un archivo HTML relativamente vacío y un gran paquete de JavaScript cuando se carga inicialmente en el navegador. Puede que esto no sea ideal para los motores de búsqueda que intentan indexar el contenido de sus páginas para que las personas puedan encontrarlo. Si representa el marcado HTML inicial en el servidor y lo envía al cliente, la carga inicial de la página contiene todo el marcado de la página que los motores de búsqueda pueden rastrear. En segundo lugar, esto crea una experiencia de carga de página inicial más rápida porque el HTML renderizado es más pequeño que el código JavaScript de toda la aplicación. React aún podrá reconocer su aplicación y administrarla después de la carga inicial.

**class App extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return <div/>**

**}**

**};**

**// change code below this line**

**ReactDOMServer.renderToString(<App />);**