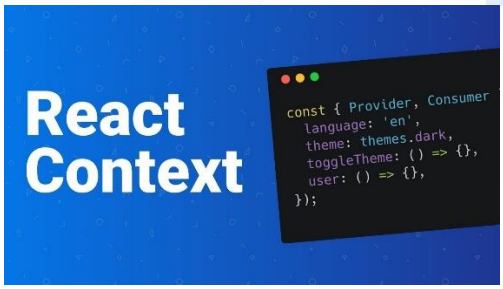

Ing. Luis Guillermo Molero Suárez

Context

Context provee una forma de pasar datos a través del árbol de componentes sin tener que pasar *props* manualmente en cada nivel.

En una aplicación típica de React, los datos se pasan de arriba hacia abajo (de padre a hijo) a través de *props*, pero esta forma puede resultar incómoda para ciertos tipos de *props* (por ejemplo, localización, el tema de la interfaz) que son necesarias para muchos componentes dentro de una aplicación. Context proporciona una forma de compartir valores como estos entre componentes sin tener que pasar explícitamente una *prop* a través de cada nivel del árbol.

Cuando usar Context

Context está diseñado para compartir datos que pueden considerarse “globales” para un árbol de componentes en React, como el usuario autenticado actual, el tema o el idioma preferido. Por ejemplo, en el código a continuación, pasamos manualmente una *prop* de “tema” para darle estilo al componente *Button*:

```
class App extends React.Component {
  render() {
    return <Toolbar theme="dark" />;
  }
}

function Toolbar(props) {
  // El componente Toolbar debe tener un prop adicional "theme"
// y pasarlo a ThemedButton. Esto puede llegar a ser trabajoso  //
si cada botón en la aplicación necesita saber el tema,  // porque
tendría que pasar a través de todos los componentes.  return (
    <div>
      <ThemedButton theme={props.theme} />    </div>
```

```
  );
}

class ThemedButton extends React.Component {
  render() {
    return <Button theme={this.props.theme} />;
  }
}
```

Usando Context podemos evitar pasar *props* a través de elementos intermedios:

```
// Context nos permite pasar un valor a lo profundo del árbol de componentes// sin pasarlo explícitamente a través de cada componente.// Crear un Context para el tema actual (con "light" como valor predeterminado).const ThemeContext = React.createContext('light');
class App extends React.Component {
  render() {
    // Usa un Provider para pasar el tema actual al árbol de abajo.    // Cualquier componente puede leerlo, sin importar qué tan profundo se encuentre.    // En este ejemplo, estamos pasando "dark" como valor actual.    return (
      <ThemeContext.Provider value="dark">        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}

// Un componente en el medio no tiene que// pasar el tema hacia abajo explícitamente.function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

class ThemedButton extends React.Component {
  // Asigna un contextType para leer el contexto del tema actual. // React encontrará el Provider superior más cercano y usará su valor.  // En este ejemplo, el tema actual es "dark".  static contextType = ThemeContext;
  render() {
```

```
    return <Button theme={this.context} />;  }
}
```

Antes de usar Context

Context se usa principalmente cuando algunos datos tienen que ser accesibles por *muchos* componentes en diferentes niveles de anidamiento. Aplícalo con moderación porque hace que la reutilización de componentes sea más difícil.

Si solo deseas evitar pasar algunos props a través de muchos niveles, la composición de componentes **suele ser una solución más simple que Context.**

Por ejemplo, considera un componente Page que pasa una prop user y avatarSize varios niveles hacia abajo para que los componentes anidados Link y Avatar puedan leerlo:

```
<Page user={user} avatarSize={avatarSize} />
// ... que renderiza ...
<PageLayout user={user} avatarSize={avatarSize} />
// ... que renderiza ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... que renderiza ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

Puede parecer redundante pasar los props de user yavatarSize a través de muchos niveles si al final solo el componente Avatar realmente lo necesita. También es molesto que cada vez que el componente Avatar necesite más props, también hay que agregarlos en todos los niveles intermedios.

Una forma de resolver este problema sin usar Context es pasar el mismo componente Avatar para que los componentes intermedios no tengan que saber sobre los props usuario o avatarSize:

```
function Page(props) {
  const user = props.user;
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}
```

```
// Ahora tenemos:
<Page user={user} avatarSize={avatarSize} />
// ... que renderiza ...
<PageLayout userLink={...} />
// ... que renderiza ...
<NavigationBar userLink={...} />
// ... que renderiza ...
{props.userLink}
```

Con este cambio, solo el componente más importante Page necesita saber sobre el uso de user y avatarSize de los componentes Link y Avatar.
Esta *inversión de control* puede hacer que tu código, en muchos casos, sea más limpio al reducir la cantidad de props que necesitas pasar a través de tu aplicación y dar más control a los componentes raíz. Sin embargo, esta inversión no es la opción correcta en todos los casos. Al mover más complejidad a niveles superiores del árbol, esos componentes en los niveles superiores resultan más complicados y obliga a los componentes en niveles inferiores a ser más flexibles de lo que podría ser deseable.

No estás limitado a un solo hijo por componente. Puede pasar varios hijos, o incluso tener varios “huecos” (slots) separados para los hijos, como se documenta aquí:

```
function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
    <NavigationBar>
      <Link href={user.permalink}>
        <Avatar user={user} size={props.avatarSize} />
      </Link>
    </NavigationBar>
  );
  return (
    <PageLayout
      topBar={topBar}
      content={content}
    />
  );
}
```

Este patrón es suficiente para muchos casos cuando necesitas separar a un componente hijo de sus componentes padres inmediatos. Puedes llevarlo aún más lejos con render props si el hijo necesita comunicarse con el padre antes de renderizar.

Sin embargo, a veces, los mismos datos deben ser accesibles por muchos componentes en el árbol y a diferentes niveles de anidamiento. Context te permite “transmitir” dichos datos, y los cambios, a todos los componentes de abajo. Los ejemplos comunes en los que el uso de Context podría ser más simple que otras alternativas incluyen la administración de la configuración de localización, el tema o un caché de datos.

API

React.createContext

```
const MyContext = React.createContext(defaultValue);
```

Crea un objeto Context. Cuando React renderiza un componente que se suscribe a este objeto Context, este leerá el valor de contexto actual del Provider más cercano en el árbol.

El argumento defaultValue es usado **únicamente** cuando un componente no tiene un Provider superior a él en el árbol. Este valor por defecto puede ser útil para probar componentes de forma aislada sin contenerlos. Nota: pasar undefined como valor al Provider no hace que los componentes que lo consumen utilicen defaultValue.

Context.Provider

```
<MyContext.Provider value={/* algún valor */}>
```

Cada objeto Context viene con un componente Provider de React que permite que los componentes que lo consumen se suscriban a los cambios del contexto.

El componente Provider acepta una prop value que se pasará a los componentes consumidores que son descendientes de este Provider. Un Provider puede estar conectado a muchos consumidores.

Los Providers pueden estar anidados para sobreescribir los valores más profundos dentro del árbol.

Todos los consumidores que son descendientes de un Provider se vuelven a renderizar cada vez que cambia la prop value del Provider. La propagación del Provider a sus consumidores descendientes (incluyendo .contextType y useContext) no está sujeta al método shouldComponentUpdate, por lo que el consumidor se actualiza incluso cuando un componente padre evita la actualización.

Los cambios se determinan comparando los valores nuevos y antiguos utilizando el mismo algoritmo que Object.is.

Nota

La forma en que se determinan los cambios puede causar algunos problemas al pasar objetos como value: mira las Advertencias.

Class.contextType

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* realiza un efecto secundario en el montaje utilizando el
valor de MyContext */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* renderiza algo basado en el valor de MyContext */
  }
}
MyClass.contextType = MyContext;
```

A la propiedad contextType en una clase se le puede asignar un objeto Context creado por React.createContext(). Al usar esta propiedad puedes consumir el valor actual más cercano de ese Context utilizando this.context. Puedes hacer referencia a ella en cualquiera de los métodos del ciclo de vida, incluida la función de renderizado.

Nota:

Solo puedes suscribirte a un solo Context usando esta API. Si necesitas leer más de una, lee Consumir múltiples Context.

Si estás utilizando la sintaxis experimental de campos de clase pública, puedes usar un campo de clase static para inicializar tu contextType.

```
class MyClass extends React.Component {
  static contextType = MyContext;
```

```
  render() {
    let value = this.context;
    /* renderiza algo basado en el valor */
  }
}
```

Context.Consumer

```
<MyContext.Consumer>
  {value => /* renderiza algo basado en el valor de contexto */}
</MyContext.Consumer>
```

Un componente de React que se suscribe a cambios de contexto. Al usar este componente puedes suscribirte a un contexto dentro de un componente de función.
Requiere una función como hijo. La función recibe el valor de contexto actual y devuelve un nodo de React. El argumento value pasado a la función será igual a la prop value del Provider más cercano para este contexto en el árbol. Si no hay un Proveedor superior para este contexto, el argumento value será igual al defaultValue que se pasó a createContext().

Nota

Para más información sobre el patrón 'función como hijo', ver render props.
Context.displayName

El objeto Context acepta una propiedad de cadena de texto displayName. Las herramientas de desarrollo de React utilizan esta cadena de texto para determinar que mostrar para el Context.

Por ejemplo, el componente a continuación aparecerá como "NombreAMostrar" en las herramientas de desarrollo:

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'NombreAMostrar';
<MyContext.Provider> // "NombreAMostrar.Provider" en las
herramientas de desarrollo
<MyContext.Consumer> // "NombreAMostrar.Consumer" en las
herramientas de desarrollo
```

Ejemplos

Context dinámico

Un ejemplo más complejo con valores dinámicos para el tema:
theme-context.js

```
export const themes = {
  light: {
    foreground: '#000000',
    background: '#eeeeee',
  },
  dark: {
    foreground: '#ffffff',
    background: '#222222',
  },
};

export const ThemeContext = React.createContext(  themes.dark //
valor por defecto);
```

themed-button.js

```
import {ThemeContext} from './theme-context';

class ThemedButton extends React.Component {
  render() {
    let props = this.props;
    let theme = this.context;    return (
      <button
        {...props}
        style={{backgroundColor: theme.background}}
      />
    );
  }
}
ThemedButton.contextType = ThemeContext;
export default ThemedButton;
```

app.js

```
import {ThemeContext, themes} from './theme-context';
import ThemedButton from './themed-button';

// Un componente intermedio que utiliza ThemedButton.
function Toolbar(props) {
  return (
    <ThemedButton onClick={props.changeTheme}>
      Change Theme
    </ThemedButton>
  );
}

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      theme: themes.light,
    };

    this.toggleTheme = () => {
      this.setState(state => ({
        theme:
          state.theme === themes.dark
            ? themes.light
            : themes.dark,
      }));
    };
  }

  render() {
    // El botón ThemedButton dentro de ThemeProvider    // usa el
tema del estado mientras que el exterior usa    // el tema oscuro
predeterminado    return (
      <Page>
        <ThemeContext.Provider value={this.state.theme}>
<Toolbar changeTheme={this.toggleTheme} />
</ThemeContext.Provider>        <Section>
          <ThemedButton />        </Section>
      </Page>
    );
  }
```

```
}

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

Actualizando Context desde un componente anidado

A menudo es necesario actualizar el contexto desde un componente que está anidado en algún lugar del árbol de componentes. En este caso, puedes pasar una función a través del contexto para permitir a los consumidores actualizar el contexto:

theme-context.js

```
// Make sure the shape of the default value passed to
// createContext matches the shape that the consumers expect!
export const ThemeContext = React.createContext({
  theme: themes.dark,  toggleTheme: () => {},});
```

theme-toggler-button.js

```
import {ThemeContext} from './theme-context';

function ThemeTogglerButton() {
  // The Theme Toggler Button receives not only the theme  // but
also a toggleTheme function from the context  return (
    <ThemeContext.Consumer>
      {({theme, toggleTheme}) => (        <button
          onClick={toggleTheme}
          style={{backgroundColor: theme.background}}>
          Toggle Theme
        </button>
      )}
    </ThemeContext.Consumer>
  );
}

export default ThemeTogglerButton;
```

app.js

```
import {ThemeContext, themes} from './theme-context';
import ThemeTogglerButton from './theme-toggler-button';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.toggleTheme = () => {
      this.setState(state => ({
        theme:
          state.theme === themes.dark
            ? themes.light
            : themes.dark,
      }));
    };

    // State also contains the updater function so it will    //
be passed down into the context provider    this.state = {
      theme: themes.light,
      toggleTheme: this.toggleTheme,    };
  }

  render() {
    // The entire state is passed to the provider    return (
      <ThemeContext.Provider value={this.state}>        <Content
/>
      </ThemeContext.Provider>
    );
  }
}

function Content() {
  return (
    <div>
      <ThemeTogglerButton />
    </div>
  );
}

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

Consumir múltiples Context

Para mantener el renderizado de Context rápido, React necesita hacer que cada consumidor de contexto sea un nodo separado en el árbol.

```
// Theme context, default to light theme
const ThemeContext = React.createContext('light');

// Contexto de usuario registrado
const UserContext = React.createContext({
  name: 'Guest',
});

class App extends React.Component {
  render() {
    const {signedInUser, theme} = this.props;

    // Componente App que proporciona valores de contexto
iniciales
    return (
      <ThemeContext.Provider value={theme}>
<UserContext.Provider value={signedInUser}>          <Layout />
        </UserContext.Provider>      </ThemeContext.Provider>
);
  }
}

function Layout() {
  return (
    <div>
      <Sidebar />
      <Content />
    </div>
  );
}

// Un componente puede consumir múltiples contextos.
function Content() {
  return (
    <ThemeContext.Consumer>      {theme => (
<UserContext.Consumer>          {user => (            <ProfilePage
user={user} theme={theme} />          )}
</UserContext.Consumer>      )}    </ThemeContext.Consumer>  );
```

```
}
```

Si dos o más valores de contexto se usan a menudo juntos, es posible que desees considerar la creación de tu propio componente de procesamiento que proporcione ambos.

Advertencias

Debido a que Context usa la identidad por referencia para determinar cuándo se debe volver a renderizar, hay algunos errores que podrían provocar renderizados involuntarios en los consumidores cuando se vuelve a renderizar en el padre del proveedor. Por ejemplo, el código a continuación volverá a renderizar a todos los consumidores cada vez que el Proveedor se vuelva a renderizar porque siempre se crea un nuevo objeto para value:

```
class App extends React.Component {
  render() {
    return (
      <MyContext.Provider value={{something: 'something'}}>
<Toolbar />
      </MyContext.Provider>
    );
  }
}
```

Para evitar esto, levanta el valor al estado del padre:

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

  render() {
    return (
      <MyContext.Provider value={this.state.value}>
<Toolbar />
      </MyContext.Provider>
    );
  }
}
```

API antigua

Nota

React previamente había liberado el API experimental de Context. La antigua API será compatible con todas las versiones 16.x, pero las aplicaciones que la utilicen deberían migrar a la nueva versión. La API antigua se eliminará en una futura versión importante de React. Lee la documentación antigua de Context aquí.