React Hooks: Context y useContext en la TODO Machine

By Bryan Garay

A medida que una aplicación crece empezamos a enviar información entre **componentes**, el conjunto de estos componentes unos padres otros hijos se denominan el **árbol de componentes.** Como habíamos trabajado la aplicación hasta este momento era mediante el paso de props entre componentes, normalmente para pasar **estados** de información o **estados derivados.** Una vez creado el estado y su lógica, los retornábamos en un componente, en otro lo recibíamos, hacíamos lógica y finalmente era necesario enviarlo como **props** al componente hijo.

El problema de usar props

Resulta que el uso de props no siempre es lo más conveniente, ya que a medida que escala nuestro código, aumentando de funcionalidades, componentes, carpetas o archivos, tendemos a anidar cada vez más estos componentes que pasan **props.** Llega cierto momento en el que usando **props** nos vemos en la necesidad de llamarlas en un componente y su único uso es recibirlas para enviarlas nuevamente a otro componente como por ejemplo en el componente **AppUI.js.**

```
import React from "react";
import { TodoCounter } from '../Components/TodoCounter/TodoCounter';
import { TodoSearch } from '../Components/TodoCounter/TodoSearch/TodoSearch';
import { TodoItem } from '../Components/TodoItem/TodoItem';
import { TodoList } from '../Components/TodoList/TodoList';
import { CreateTodoButton } from '../Components/CreateTodoButton/CreateTodoButton';
import { TodosLoading } from '../Components/TodosLoading';
import { TodosError } from '../Components/TodosError';
import { EmptyTodos } from '../Components/EmptyTodos';


function MainPage( {
        completedTodos,
        totalTodos,
        searchValue,
        setSearchValue,
        searchedTodos,
        checkTodo,
        deleteTodo,
        loading,
        error
    }) {
    return (
        // React.Fragment como contenedor invisible.
        <>
            <TodoCounter
                completed={completedTodos}
                total={totalTodos}
            />
```

Aquí recibimos el listado de props y luego al llamar componentes como **<TodoCounter>** simplemente las volvemos a enviar a otros componentes hijos que finalmente son los que usarán esa información o incluso la envíen dentro de otro componente más.

El problema de trabajar de esta manera es que estamos recibiendo y enviando estados de información desde todos los niveles del árbol de componentes, de forma un poco desordenada e incluso innecesaria como vimos en el caso anterior de las props del **TodoCounter**. A esta práctica de enviar y obtener datos entre varios niveles del árbol de componentes de nuestra aplicación se llama **prop drilling.**

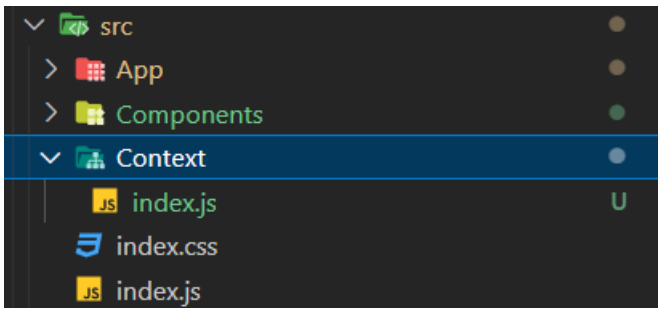
Solución: El uso de un contexto.

Dentro de todo este enviar y recibir información mediante **props** entre **componentes**, subyace la cuestión de que hay información que pasa por tantos niveles o capas de nuestra aplicación porque quizás tienen un uso global, razón por la cual cierta información debería estar disponible desde una especie de "**almacén global**" para que se pueda comunicar directamente con el componente que de verdad va a utilizar esa información y no solo "**reenviarla**".

Para esta situación resulta útil usar **Context** es la forma que permite **React** para proveer información a través del árbol de componentes de nuestra aplicación sin la necesidad de pasar props de componentes padres a hijos de forma manual y a través de cada nivel.

La API de React: createContext

React ofrece una forma de crear estos **contextos** de información mediante un objeto **Context.** Esto se lo puede crear preferiblemente desde su propia **carpeta de contextos** (puede haber varios dependiendo de la necesidad) y archivo **index.js** para organizar el código.



Usamos entonces la siguiente sintaxis:

```
const TodoContext = React.createContext();
```

Este contexto siempre contendrá dos elementos que son los que nos permitirán definir los elementos que permitan **proveer** o definir la información a enviar y los que permiten **consumir** o recibir la información, estos son:

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

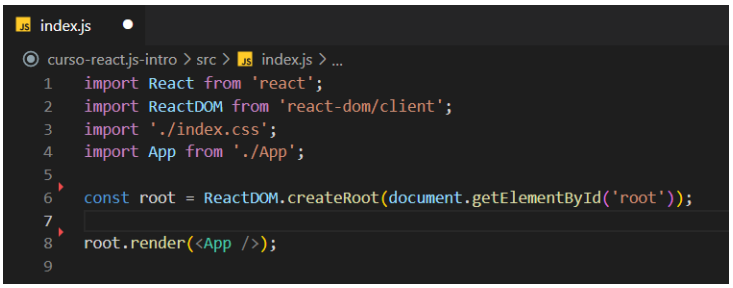
Y

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

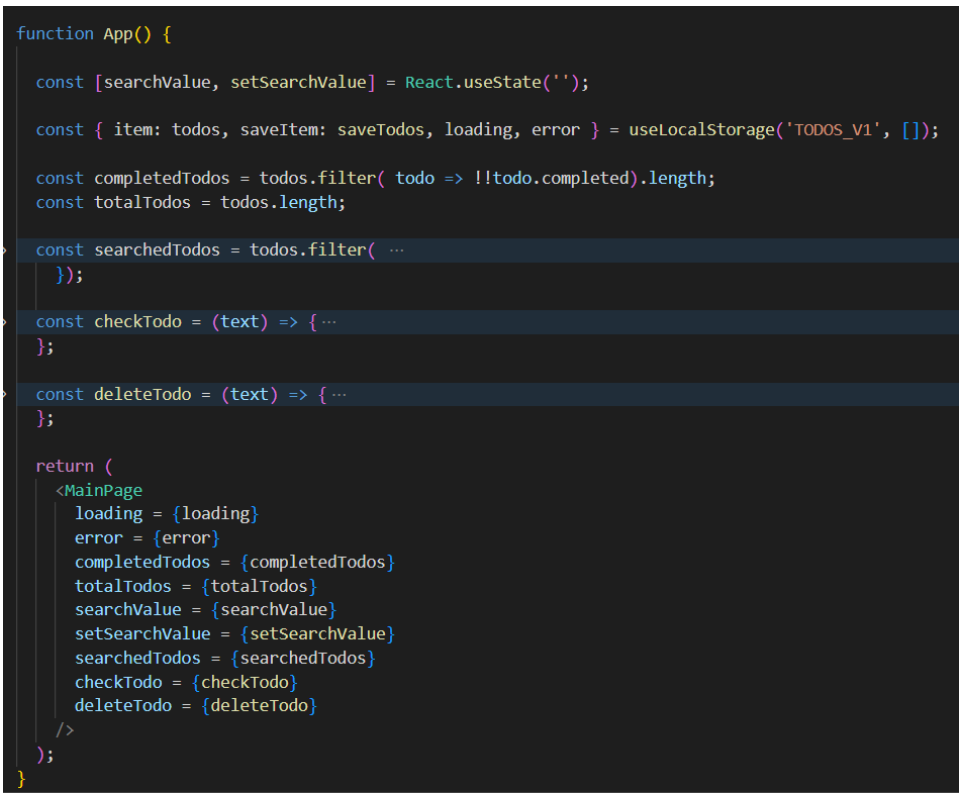
Con estas dos sintaxis podremos enviar y recibir la información entre componentes.

Ejemplo de uso del Context en la TODO machine

Vamos a empezar analizando uno de los archivos principales de la aplicación que es el **index.js** y su función **App.** Recordemos que este archivo es uno de los principales de nuestro proyecto ya que la función **App** es la que se **renderiza** en el **index.js de todo el proyecto**, por tanto, por motivos de mejorar la organización del código del proyecto, debería estar más limpio y contener solo lo necesario y ya vimos que podemos gestionar de mejor manera el enviar información usando **contextos** en vez de únicamente **props.**



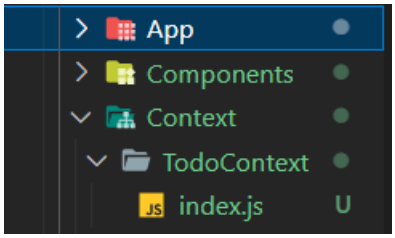
La forma en la que vamos a usar el **Context** es separando la lógica para **proveer** información que contiene nuestra función **App** desde la cual parte toda la información que usamos en nuestra aplicación, enviando como **props** los estados de carga y error, el total de "Todos", el valor buscado, el "Todo" completo o eliminado, etc.



Como vemos aparte del procesamiento de la información recibida del **useLocalStorage.js** tenemos lógica con la que se crea **estados derivados** y estos a su vez luego de renderizar el componente **MainPage**, son enviamos como props por dentro del componente.

Analizado esto vamos al proceso:

1. Empezaremos creando una carpeta de **Contextos,** creando su archivo **index.js**



2. En este archivo luego de importar React, vamos a crear nuestro nuevo contexto con la sintaxis aprendida: React.createContex(); Hacer una función para nuestro provider enviando como **props "children"** y en el **return** vamos a usar la sintaxis para el **context provider** y dentro encapsular también **children**:

```
<TodoContext.Provider value={{ }}>
        {children}
</TodoContext.Provider>
```

Con esta lógica nos aseguramos de que cualquier componente hijo del **TodoContext** reciba toda esa información de nuestro contexto global. Quedaría entonces así:

```jsx
import React from 'react';

const TodoContext = React.createContext();

function TodoProvider({ children }) {

    return (
      <TodoContext.Provider value={{ }}>
        {children}
      </TodoContext.Provider>
    );
  }

 export { TodoContext, TodoProvider };
```

3. Ahora que tenemos la base del context podemos empezar a transferir esa lógica de la función **App** a la función **TodoProvider** para que sea esta la que contenga esa información. Una vez ahí toda esa lógica, vamos a usar la propiedad **value** del **TodoContext.Provider** para enviar todos esos **estados** y **estados derivados** de información que necesita nuestra aplicación, de la misma manera como enviábamos **props.** Nuestro **TodoContext/index.js** quedaría así:

   Enviando todos esos estados de información como **value** del **TodoContext.Provider** y declarando el **children** cualquier componente que esté envuelto por el **TodoProvider** recibirá todas esas propiedades y estarán listas para ser usadas.

```
const TodoContext = React.createContext();

function TodoProvider({ children }) {

  const { item: todos, saveItem: saveTodos,
          loading, error } = useLocalStorage('TODOS_V1', []);
  const [searchValue, setSearchValue] = React.useState('');
  const completedTodos = todos.filter( todo => !!todo.completed).length;
  const totalTodos = todos.length;

  const searchedTodos = todos.filter( ···
    });

  const checkTodo = (text) => {···
  };

  const deleteTodo = (text) => {···
  };
  return (
    <TodoContext.Provider value={{
      loading,
      error,
      checkTodo,
      totalTodos,
      searchValue,
      setSearchValue,
      searchedTodos,
      completedTodos,
      deleteTodo, }}>

      {children}

    </TodoContext.Provider>
  );
```

4. Ahora, nos podemos dar cuenta hay un problema ya que al cortar nuestro código del **App/index.js** a este directorio dejamos sin comunicación a las funciones que usaban el custom hook del **useLocalStorage**, por lo que resulta necesario transferir ese archivo a la carpeta del **TodoContext** e importarlo.

   La importación del **useLocalStorage:**

5. Volviendo al **App/index.js** este ahora quedaría casi vacío y simplemente en el return renderizaría **<AppUI>**, pero como en el AppUI renderizamos todos los componentes hijos hay que empezar a transferir la información, por lo que vamos a encapsular el AppUI dentro del **TodoProvider**, convirtiéndose así en un **children** y recibiendo todas las propiedades, por lo que ya podemos eliminar ese listado de **props** que enviábamos.
   El archivo quedaría así:

```
import React from 'react';
import { AppUI } from './AppUI';
import { TodoProvider } from '../Context/TodoContext';

function App() {

  return (
    <TodoProvider>
      <AppUI/>
    </TodoProvider>
  );
}

export default App;
```

6. Ya tenemos el **contexto** y la interfaz en donde renderizamos los componentes está envuelta en nuestro **provider,** por lo que estamos listos para **consumir** nuestro contexto, es decir hacer uso de toda esta lógica para enviar la información el árbol de componentes. En este punto también tenemos que darnos cuenta de que al ya no usar **props** hay que actualizar la forma en la que recibe la información nuestros componentes:
   - AppUI
   - TodoCounter
   - TodoSearch
   - TodoList

Para esto tenemos 2 sintaxis o formas **consumir** el **contexto:** usando **TodoContext.Consumer** o **React.useContext.**

7. **Usando < TodoContext.Consumer>**

   En esta primera forma tenemos una sintaxis un poco más larga y menos legible, que podríamos encontrar en proyectos grandes en producción, usando **render functions**. Sin embargo, para nuestra TODO Machine quizás no sea tan necesaria esta sintaxis y la usaremos solo en el componente **AppUI** para mostrarlo.

   Primero vamos a importar el **TodoContext.** Después podemos eliminar todos los **props** que íbamos a recibir y enviar por ejemplo en **TodoCounter** o **TodoSearch.** En **TodoList** el caso es un poco diferente ya que ahí usamos directamente esas **props**, lo que no sucede con **TodoCounter** o **TodoSearch** en donde solo las reenviamos.

   Enfocándonos entonces en **TodoList** vamos a encapsular este código con la sintaxis del **TodoContext.Consumer** y enviarle una **render function** con el listado de propiedades a consumir:

```
import { TodoContext } from '../Context/TodoContext';

function AppUI() {
    return (
        <>
            <TodoCounter/>
            <TodoSearch/>

            <TodoContext.Consumer>
                {({
                    loading,
                    error,
                    totalTodos,
                    checkTodo,
                    deleteTodo,
                    searchedTodos
                }) => (
                    <TodoList>...
                    </TodoList>
                )}
            </TodoContext.Consumer>

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

   Con esta sintaxis ya estaría disponible la información para el **TodoList**, pero no aún para **TodoCounter** y **TodoSearch:**

8. **Usando useContext**

Como cualquier otro **Hook** de **React**, **useContext** nos ayuda a simplificar la sintaxis y hacer el código más legible. Su función es la misma que la del **Context.Consumer** pero con otra sintaxis.

Para el **useContext** la forma en la que consumiremos la información del **TodoContext** será primero importando ese contexto y luego declarando de la siguiente manera:

```
const { Propiedades a consumir } = React.useContext(TodoContext);
```

Es decir que en **const** definiremos lo que antes llamábamos como **props** pero que vienen directo del **contexto global** y no como **prop drilling** del árbol de componentes. Después de poner ese listado de propiedades usamos **React.useContext** y como parámetro enviamos el nombre del **contexto** creado.

Nuestros componentes quedarían así:

- **AppUI**

```
import { TodoContext } from '../Context/TodoContext';

function AppUI() {
    const {
        loading,
        error,
        totalTodos,
        checkTodo,
        deleteTodo,
        searchedTodos
      } = React.useContext(TodoContext);

    return (
        <>
            <TodoCounter/>
            <TodoSearch/>

            <TodoList>...
            </TodoList>

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

Debemos recordar que en el caso de **AppUI** solo estamos consumiendo las propiedades que necesita **TodoList**, las de **TodoCounter** y **TodoSearch**, se deben establecer de forma análoga en el componente en el que se usen las props.

- **TodoCounter**
  De forma análoga se deben eliminar los **props** y además como recibimos directamente las **propiedades** a **consumir** tendremos que adaptar las declaraciones usadas en el código, es decir que se debe reemplazar por **completedTodos** y **totalTodos**, donde antes teníamos **completed** y **total**:

```
import React from 'react';
import './TodoCounter.css';
import { TodoContext } from '../../Context/TodoContext';

function TodoCounter() {
    const {
      completedTodos,
      totalTodos
      } = React.useContext(TodoContext);

    return (
        <h1 className='TodoCounter'>
            Hola Cosmonauta 👩‍🚀 👨‍🚀 Has completado
            <span>{completedTodos}</span> de
            <span>{totalTodos}</span> TODO´s
        </h1>
    );
}
```

- **TodoSearch**
  Se realiza el mismo procedimiento que para **TodoCounter**

```
import React from 'react';
import './TodoSearch.css';
import { AiOutlineSearch } from 'react-icons/ai';
import { TodoContext } from '../../Context/TodoContext';

function TodoSearch() {
    const {
        searchValue,
        setSearchValue
        } = React.useContext(TodoContext);

    return (
        <span className='TodoSearch'>
            <input
                placeholder="Busca una tarea"
                className="TodoSearchInput"
                value = {searchValue}
                onChange={(event) =>
                    {
                        setSearchValue(event.target.value);
                    }}
            />
            <span className='IconSearch'>
                <AiOutlineSearch/>
            </span>
        </span>
    );
}
```