



Clase 10. React JS

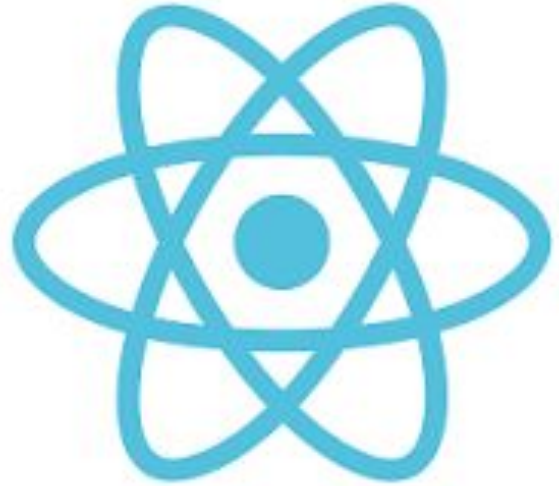
EVENTOS APLICATIVOS



OBJETIVOS DE LA CLASE

- Organizar los eventos aplicativos de nuestros proyectos
- Crear interacciones persistentes entre componentes
- Desarrollar implementaciones custom de context

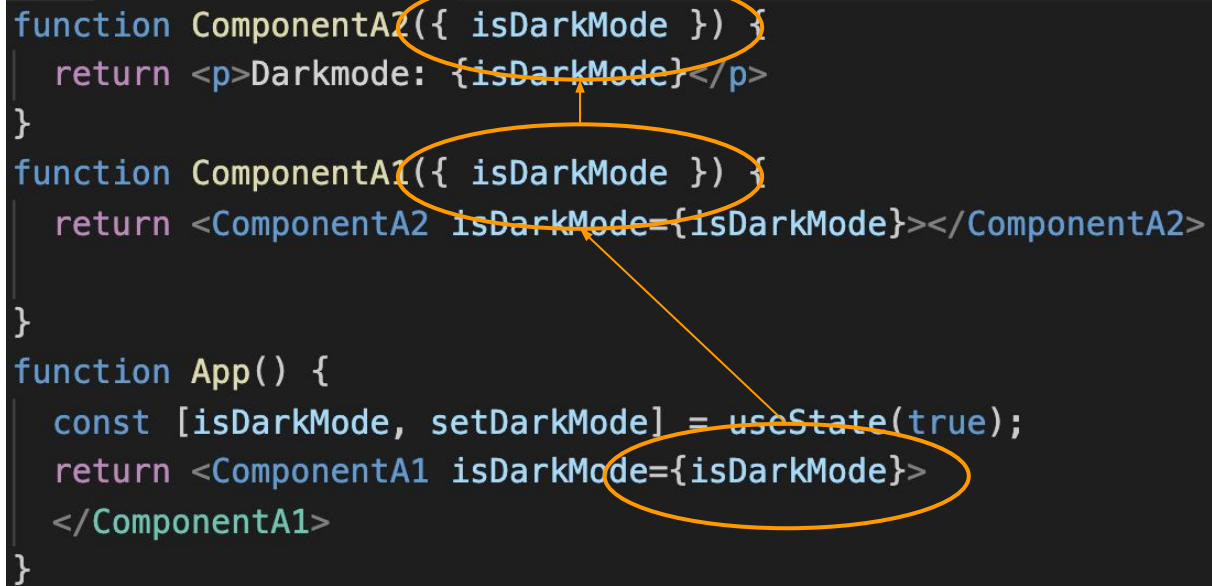
Dado que React funciona con un flujo de datos unidireccional, la única manera de transmitir datos es vía **props**



CONTEXTO

Si quisiera llevar una variable desde el punto más alto a mi punto de uso, me vería en algo parecido a lo siguiente

```
function ComponentA2({ isDarkMode }) {  
  return <p>Darkmode: {isDarkMode}</p>  
}  
function ComponentA1({ isDarkMode }) {  
  return <ComponentA2 isDarkMode={isDarkMode}></ComponentA2>  
}  
function App() {  
  const [isDarkMode, setDarkMode] = useState(true);  
  return <ComponentA1 isDarkMode={isDarkMode}>  
    </ComponentA1>  
}
```

A diagram illustrating prop drilling. Three orange ovals highlight the 'isDarkMode' prop in the function signatures and JSX of ComponentA2, ComponentA1, and App. An arrow points from the 'isDarkMode' prop in the App component's return statement to the 'isDarkMode' prop in the ComponentA1 component's return statement. Another arrow points from the 'isDarkMode' prop in the ComponentA1 component's return statement to the 'isDarkMode' prop in the ComponentA2 component's return statement.

Requeriría pasar la prop 'isDarkMode' por cada componente hasta llegar al requerido, y este caso son solo tres nestings (anidaciones).

Declarando un **contexto**, podemos sacar todos los pasamanos intermedios.

Esto a sabiendas del tipo de variable, que debería ser global

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

function ComponentA2() {
  const isDarkMode = useContext(ThemeContext)
  return <p>Darkmode: {isDarkMode}</p>
}

function ComponentA1() {
  return <ComponentA2></ComponentA2>
}

function App() {
  const [isDarkMode, setDarkMode] = useState(true);
  return <ThemeContext.Provider value={isDarkMode}>
    | <ComponentA1 />
  </ThemeContext.Provider>
}
```

***CREANDO UN
CONTEXT***

La declaración es muy simple

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

Y puedo darle un default value

```
const ThemeContext = React.createContext(false);
```

En los próximos pasos entenderemos para qué sirve este 'default'

CONTEXT PROVIDER
(proveedor)

Tengo que **envolver** el
nodo de React al que
quiero que este provider
propague hacia sus
children

```
}  
function App() {  
  const [isDarkMode, setDarkMode] = useState(true);  
  return <ThemeContext.Provider value={isDarkMode}>  
    <ComponentA1 />  
  </ThemeContext.Provider>  
}
```

*Cuidado: Si no seteamos **value**, usará el valor que se le dió en la creación*

CONSUMIENDO UN CONTEXTO

De esta manera

useContext (ThemeContext)

Nos devolverá el value del

<x.Provider value={}>

```
import React, { useContext } from 'react'

const ThemeContext = React.createContext();

function ComponentA2() {
  const isDarkMode = useContext(ThemeContext)
  return <p>Darkmode: {isDarkMode}</p>
}
```

*Probablemente también importemos el contexto desde el archivo donde lo hayamos creado, por ejemplo **src/context/themeContext.js***

DECLARANDO UN CONSUMER (modo alternativo)

Utilizando un consumer podemos lograr un efecto similar y si el value cambia React hará el re-render cuando cambie el value del provider

```
import React, { useContext } from 'react'

const ThemeContext = React.createContext(false);

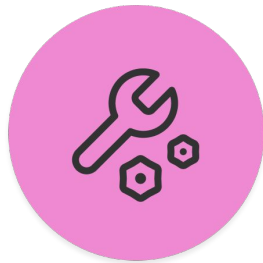
function ComponentA2() {
  return <ThemeContext.Consumer>
    |   {(isDarkMode) => (<p>Darkmode: {isDarkMode}</p>)}
    </ThemeContext.Consumer>
}
```

*Esto consumers son cómodos cuando no necesitamos el estado en el componente consumidor (**ComponentA2**) para lograr otro efecto secundario*

VAMOS AL CÓDIGO



CODER HOUSE



Crea tu contexto

En tu proyecto en `src/context/` crea un Contexto llamado **cartContext.js** cuyo valor default sea `[]`, e importalo como provider de tu app con value `[]`

Pista: Los pasos son los mismos que en las slides, pero deberás exportar tu context creado para poder usarlo en App.js ;)

Tiempo: 15 minutos

CODER HOUSE

¿PREGUNTAS?



BREAK



CODER HOUSE

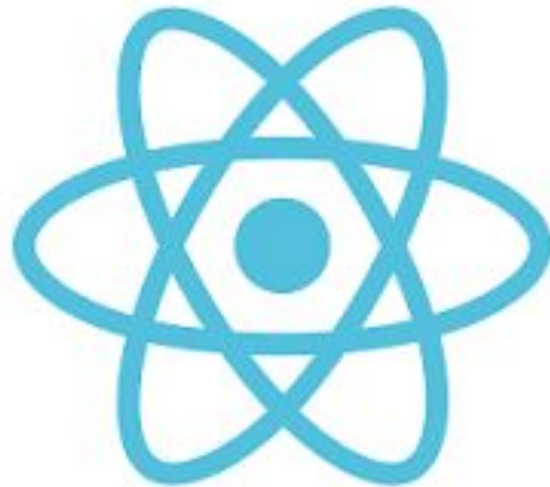


Recap context

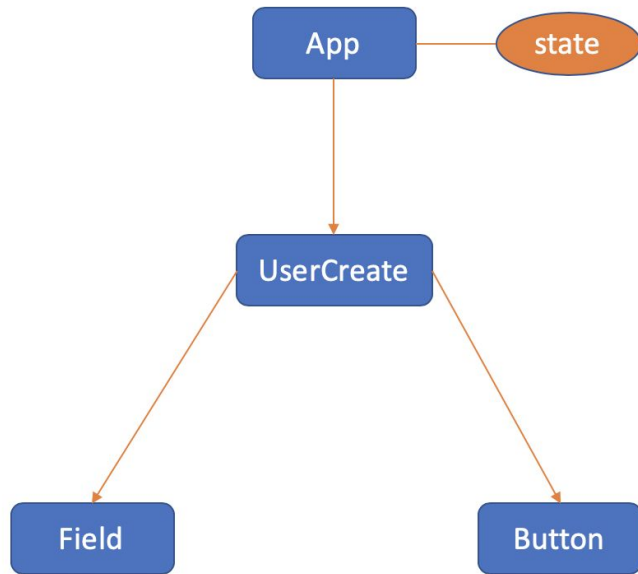
- Permiten compartir un valor único cross-app
- Reducen el wrapper-hell (infierno de nesting)
- No solo pueden llevar values, sino cualquier tipo de **fn**, **obj** o referencia
- Toman el valor del provider mas cercano o el definido durante su declaración

CONTEXTO DINÁMICO

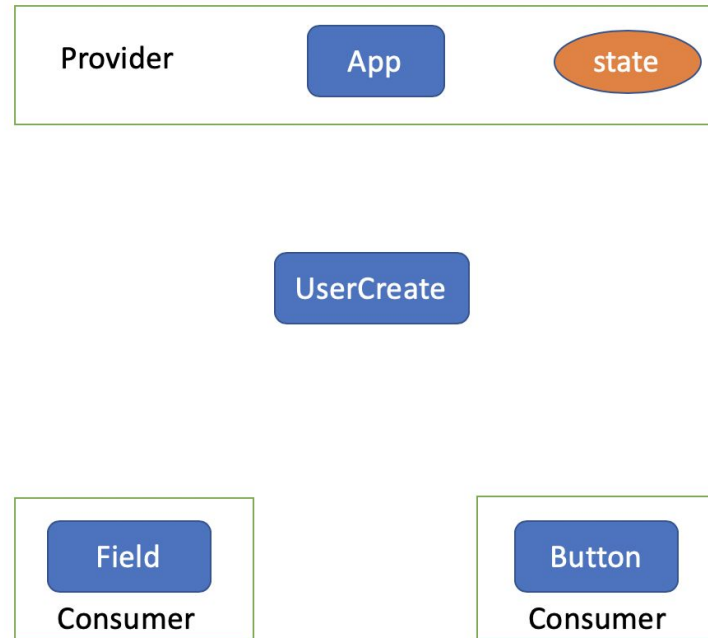
Los **contextos** también pueden ser alterados en **tiempo de ejecución**, y sus efectos **propagados** al resto de los consumidores

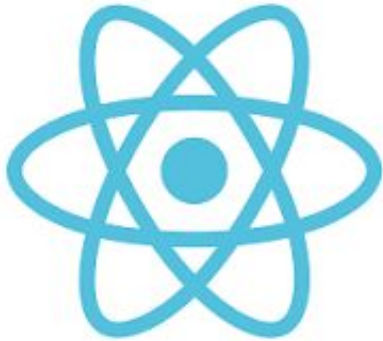


Props Data Flow



Context Data Flow





Flexibilidad de context

+

State

+

Hooks

+

Higher Order Components

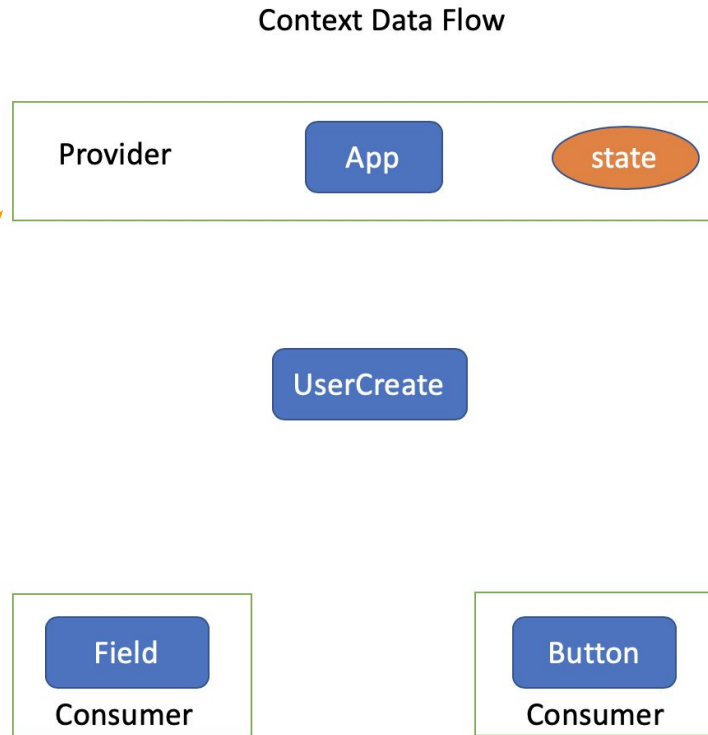
=



CONFIGURANDO UN NODO PROVEEDOR

Lo importante al configurar esta estrategia será:

1. Saber **elegir** cuál es el **punto estratégico** de mi aplicación donde iniciaré el estado de ese context
2. **Combinarlo** estratégicamente con un **useState** para poder **mutarlo** y que el useState me ayude a hacer trigger de renderings en consumers





CODER TIPS:- CONTEXT

- Puedo tener múltiples contextos del mismo corriendo en una aplicación
- El valor provisto por el hook de contexto será el del parent provider más próximo del árbol a mi componente
- Más que con redux, ¡una gran cantidad de casos de uso podemos lograrlos sólo conociendo bien react, context, hooks y patrones que aprendimos en este curso, no te dejes llevar!

CREANDO UN CUSTOM PROVIDER

Contexto dinámico

Si bien podríamos declarar un provider simplemente haciendo lo siguiente:

```
<CacheContext.Provider value={{ cache: []}}>  
  <CompA />  
  <CompB />  
</CacheContext.Provider>
```

Contexto dinámico: Custom Provider

Podemos dar mucho más valor agregado si lo transformamos en un proveedor con capacidades customizadas y su estado linkeado

```
export default function CacheProvider({ defaultValue = [], children }) {  
  const [ cache, setCache ] = useState(defaultValue);  
  
  function getFromCache(id) {  
    return cache.find(obj => obj.id === id)  
  }  
  
  function isInCache(id) {  
    return id === undefined ? undefined : getFrom !== undefined  
  }  
  
  function addToCache(obj) {  
    if (isInCache(obj && obj.id)) {  
      console.log('Won-t add existing obj to caché');  
      return;  
    }  
    setCart([...cache, obj]);  
  }  
  
  return <CartContext.Provider value={{ cache, addToCache, isInCache, cacheSize: cache.length }}>  
    {children}  
  </CartContext.Provider>  
}
```

Contexto dinámico: Custom Provider

Creamos un componente virtual de fachada

Al que podemos agregar helpers

Y hacer wrapping de cualquier nodo que quiera transformar en provider

```
export default function CacheProvider({ defaultValue = [], children }) {  
  const [ cache, setCache ] = useState(defaultValue);  
  
  function getFromCache(id) {  
    return cache.find(obj => obj.id === id)  
  }  
  
  function isInCache(id) {  
    return id === undefined ? undefined : getFrom !== undefined  
  }  
  
  function addToCache(obj) {  
    if (isInCache(obj && obj.id)) {  
      console.log('Won-t add existing obj to caché');  
      return;  
    }  
    setCart([...cache, obj]);  
  }  
  
  return <CartContext.Provider value={{ cache, addToCache, isInCache, cacheSize: cache.length }}>  
    {children}  
  </CartContext.Provider>  
}
```

Consumir el custom provider

Entonces simplemente envolvemos los componentes que queramos

El **custom provider** dará estado y hará **sync con sus consumers** de manera **automática** en updates

```
export default function App() {  
  return (  
    <div>  
      <h1>Custom context app</h1>  
      <CacheProvider> { /* Sólo A y B pueden acceder*/ }  
        <ComponentA />  
        <ComponentB />  
      </CacheProvider>  
      <ComponentC />  
    </div>  
  );  
}
```

Coder tip: Observa como hemos excluido a **C** del Provider

VAMOS AL CÓDIGO



CODER HOUSE

¿PREGUNTAS?





CartContext

Al clickear en el botón de **comprar** en ItemDetail.js, se debe guardar el producto y cantidad comprada en el nuevo **CartContext**. Al navegar a Cart.js se debe poder ver el detalle del producto agregado, y en el CartIcon.js deberíamos ver la longitud de ítems agregados.

Formato de entrega: GIF mostrando la sync compra+icon+cart

Tiempo: 30 minutos



¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Eventos aplicativos persistentes
 - Context
 - CustomProviders
- 



OPINA Y VALORA ESTA CLASE