



**¡Les damos la
bienvenida!**

¿Comenzamos?

Esta clase va a ser

- grabada

Class 11. REACT JS

Context

Temario

10

Eventos

- ✓ Eventos
- ✓ Componentes basados en eventos

11

Context

- ✓ [Contexto](#)
- ✓ [Contexto dinámico](#)
- ✓ [Nodo proveedor](#)
- ✓ [Custom Provider](#)

12

Técnicas de rendering

- ✓ Principios básicos de React
- ✓ Rendering condicional
- ✓ Rendering optimization

Objetivos de la clase

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

CLASE N°10

Glosario

Evento: es un estímulo programático, que puede ser provocado de manera automática, o ser el resultado de una interacción del usuario con la UI.

Event Listener: es un patrón de diseño que sirve, como su nombre lo indica, para escuchar cuando un algo ocurre en algún elemento, librería o API, y poder realizar una acción en consecuencia.consectetur adipiscing elit.



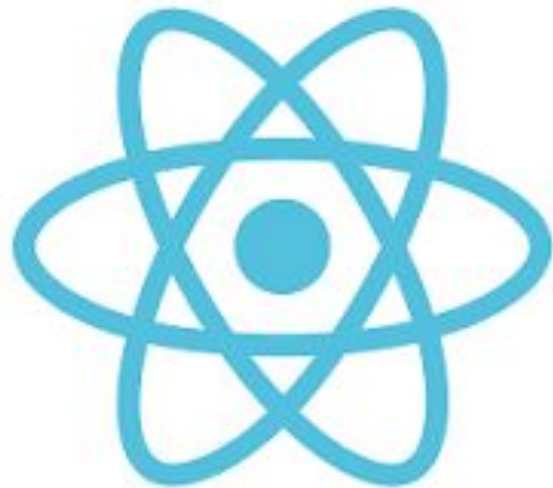
MAPA DE CONCEPTOS



Contexto

Contexto

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

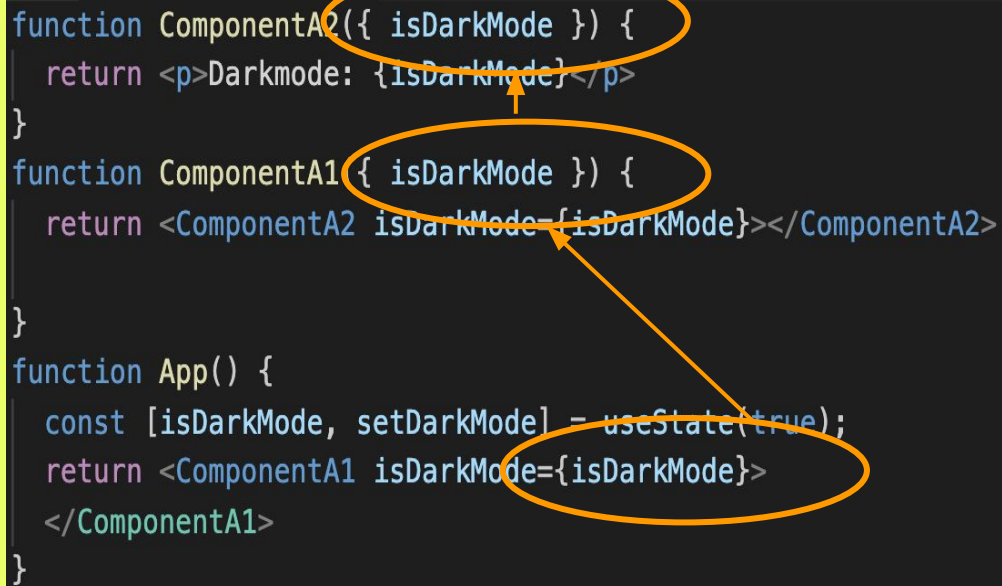


Contexto

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

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

```
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>  
}
```



Contexto

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 contexto

Creando un contexto

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)

Context Provider

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

Cuidado: si no seteamos **value**, usará el valor que se le dio en la creación.

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

Consumiendo un contexto

Consumiendo un contexto

De esta manera

useContext(ThemeContext)

nos devolverá el value
del <x.Provider value={} >

Probablemente, también
importemos el contexto
desde el archivo donde
lo hayamos creado, por
ejemplo

src/context/themeContext.js

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

const ThemeContext = React.createContext();

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

Declarando un Consumer (Modo Alternativo)

Declarando un Consumer

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

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

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

const ThemeContext = React.createContext(false);

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



Ejemplo en vivo

Vamos al código



Crea tu contexto

En tu proyecto en `src/context/`, crea un Contexto llamado `cartContext.js`

Duración: **15 minutos**



ACTIVIDAD EN CLASE

Título de la Actividad

Descripción de la actividad.

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`.

Cuentas con 15 minutos para realizar la actividad.



Break

¡10 minutos y volvemos!

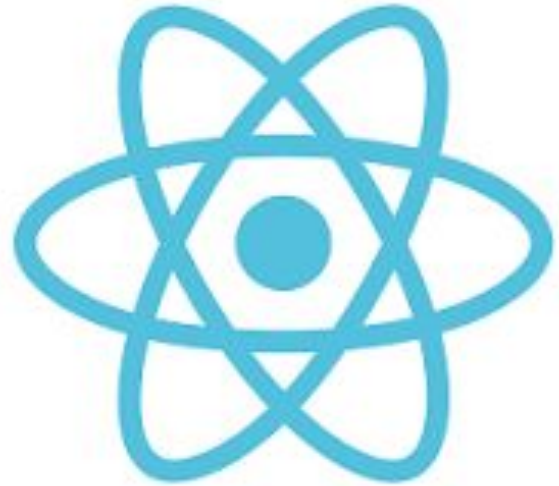
Recapitulación Context

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

Contexto Dinámico

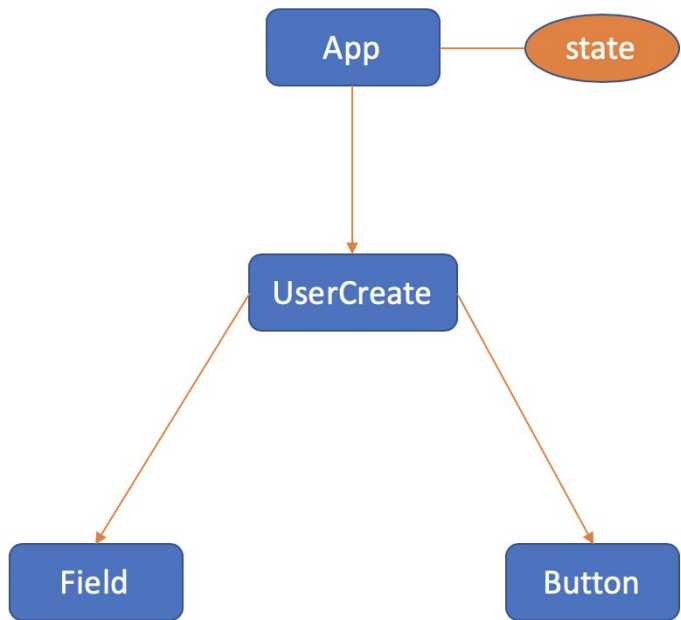
Contexto Dinámico

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

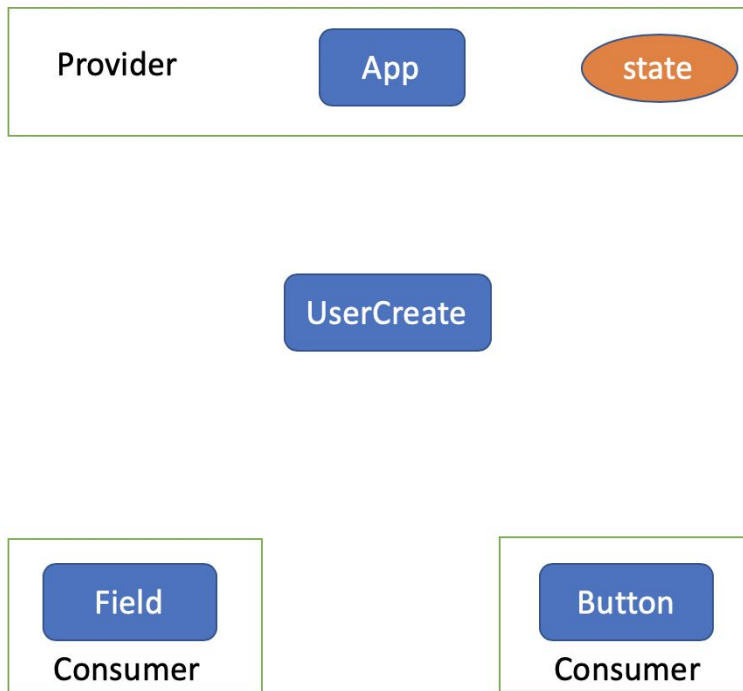


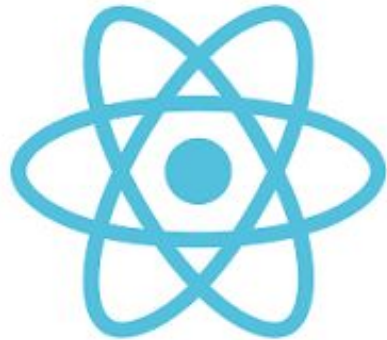
Contexto Dinámico

Props Data Flow



Context Data Flow





Contexto Dinámico

Flexibilidad de context

+

State

+

Hooks

+

Higher Order Components

=



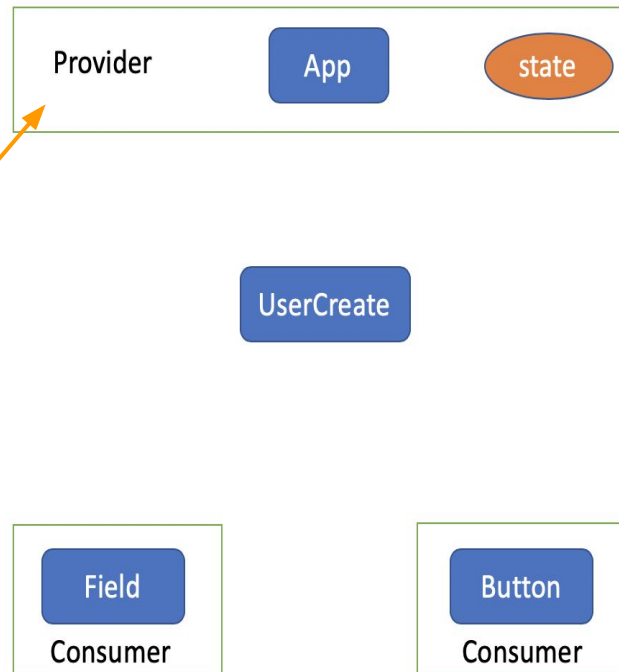
Configurando un Nodo Proveedor

Configuración

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

Context Data Flow



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.

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

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



Ejemplo en vivo

Vamos al código



#Coderalert

Ingresa al manual de prácticas y realiza la quinta actividad “Cart Context”. Ten en cuenta que el desarrollo de la misma será importante para la resolución del Proyecto Final.



Cart Context

Descripción de la actividad.

- ✓ Implementa React Context para mantener el estado de compra del user

Recomendaciones

- ✓ Al clicar comprar en ItemDetail se debe guardar en el CartContext el producto y su cantidad en forma de objeto { name, price, quantity, etc. } dentro del array de productos agregados
- ✓ Detalle importante: CartContext debe tener la lógica incorporada de no aceptar duplicados y mantener su consistencia.
- ✓ Métodos recomendados:
 - ✓ addItem(item, quantity) // agregar cierta cantidad de un ítem al carrito
 - ✓ removeItem(itemId) // Remover un ítem del cart por usando su id
 - ✓ clear() // Remover todos los items
 - ✓ isInCart: (id) => true|false

¿Preguntas?

Resumen de la clase hoy

- ✓ Eventos aplicativos persistentes.
- ✓ Context.
- ✓ CustomProviders.

Opina y valora
esta clase

Muchas gracias.

#DemocratizandoLaEducación