Front End II

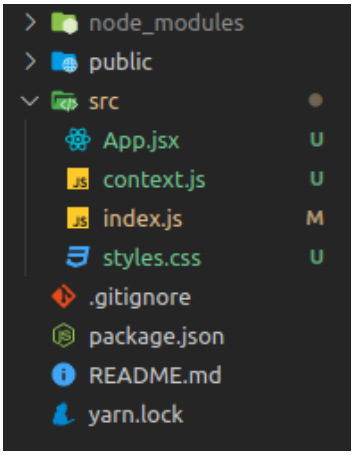
Ejemplo de un theme context

Vamos a realizar el paso a paso para crear un switch de un dark mode y light mode que hoy en día la mayoría de los sitios web utilizan.

Paso a paso

Como base, vamos a realizar un **create-react-app** para inicializar un proyecto listo para utilizar.

Luego vamos a limpiar los archivos para que solamente tengamos los siguientes:



Nuestro App.jsx debería verse así de simple. Recordemos limpiar dentro de index y app lo que no se necesita.

```
import React from 'react'
import "./styles.css"
const App = () => {
 return (
```

```
    <div>
      CONTEXT TUTORIAL
    </div>
  )
}
export default App
```

Vamos a crear un archivo donde crearemos nuestro theme context con el light y dark mode. El archivo se llamará **'context.js'**.

Este cuenta con el objeto con nuestros colores y el createContext para utilizar en nuestro proyecto.

Con el createContext podemos hacer uso del componente Provider y el Hook de useContext, que nos va a permitir consumir los datos que se pasen por el provider.

```
import React, {createContext} from 'react';
export const themes = {
 light: {
   font: 'black',
   background: 'white'
 },
 dark: {
   font: 'white',
   background: 'black'
 }
};
const ThemeContext =.createContext(themes.light);
export default ThemeContext;
```

➔ Exportamos un objeto con nuestros themes.
➔ Exportamos nuestro context con el valor inicial de light.

Configurando nuestro ThemeContext

Volvamos a nuestra app.jsx. La primera parte son los imports:

```
import React, { useState } from 'react'
import "./styles.css"
import ThemeContext, { themes } from './context'
```

Traemos el componente **ThemeContext** que viene con Key para completar, como por ejemplo, el Provider que vamos a usar.

Luego, dentro de nuestro componente App, vamos a agregar un state dejando el themes.light como defaultValue, y creamos una función que se encargue de cambiar el theme.

En el return utilizamos nuestro componente **themeContext.Provider** y le damos los valores theme (font y background) y la función para cambiar el color.

Nuestro código se verá así:

```
const App = () => {
 const [theme, setTheme] = useState(themes.light);
 const handleChangeTheme = () => {
   if (theme === themes.dark) setTheme(themes.light)
   if (theme === themes.light) setTheme(themes.dark)
 }
 return (
   <ThemeContext.Provider value={{ theme, handleChangeTheme }}>
     <div>
```

```
      <h1>CONTEXT TUTORIAL</h1>
    </div>
  </ThemeContext.Provider>
 )
}
export default App
```

Context API + useState

En este caso, en vez de almacenar un dato estático (como solamente un título) en nuestro Contexto, nos apoyamos en la API de estado para poder implementar un dato dinámico con un incluso una función seteadora que nos permita actualizarlo.

```
//...
const [theme, setTheme] = useState(themes.light);
const handleChangeTheme = () => {
    //...
}
return (
  <ThemeContext.Provider value={{ theme, handleChangeTheme }}>
    //...
  </ThemeContext.Provider>
)}
```

Mejoras de performance

Cómo React renderiza nuevamente todo el componente cada vez que haya un cambio de estado, podemos caer en el riesgo de tener bajas de optimización mediante esta

implementación. Por lo que una buena práctica sería su utilización en conjunto con el hook **useMemo** para memorizar el valor del estado y de su función seteadora.

```
 //...
 const [theme, setTheme] = useState(themes.light);
 const handleChangeTheme = () => {
     //...
 }
 const providerValue =  useMemo(()=>({theme,
handleChangeTheme}),[theme,handleChangeTheme])

 return (
   <ThemeContext.Provider value={providerValue}>
     //...
   </ThemeContext.Provider>
```

Agregando nuevos componentes

Vamos a agregar tres componentes para darle más contenido a nuestro proyecto. Crearemos una carpeta llamada components donde crearemos archivos llamados **Navbar.jsx**, **Body.jsx** y **Layout.jsx**.

Nuestro import en App debería verse así:

```
 return (
   <ThemeContext.Provider value={providerValue}>
     <Layout>
       <Navbar />
       <Body />
     </Layout>
   </ThemeContext.Provider>
 )
```

```
}
export default App
```

Veamos cómo quedarían los códigos a nuestros archivos:

Navbar.jsx

```
import React from 'react'
import "../styles.css"

const Navbar = () => {
   return (
       <div className="navbar">
           <p>Inicio</p>
           <button>THEMED BUTTON</button>
       </div>
   )
}

export default Navbar
```

Body.jsx

```
import React from 'react'

const Body = () => {
   return (
       <div>
            <h1>CONTEXT TUTORIAL</h1>
```

```
            <p>Lorem ipsum dolor sit amet consectetur adipisicing elit.
Ratione, quis assumenda culpa distinctio necessitatibus provident
accusamus quas suscipit atque et officia modi veritatis. Adipisci nulla
temporibus eum. Beatae, quis esse!</p>
        </div>
    )
}

export default Body
```

Layout.jsx

```
import React, {useContext} from 'react'
import ThemeContext from '../context'

const Layout = ({children}) => {
    const {theme} = useContext(ThemeContext);

    return (
        <div style={{background: theme.background, color:theme.font}}>
            {children}
        </div>
    )
}

export default Layout
```

El layout es un simple div que envuelve a nuestro proyecto donde utilizaremos nuestro context.

Trabajando en el button

Nos queda darle el evento al botón de nuestra navbar para que haga el cambio:

```
import React, {useContext} from 'react'
import "../styles.css"
import ThemeContext from '../context'
const Navbar = () => {
   const {theme, handleChangeTheme} = useContext(ThemeContext)
   return (
       <div className="navbar">
           <p>Inicio</p>
           <button
           onClick={handleChangeTheme}
           style={{background: theme.background, color:theme.font}}>
               Change Theme
               </button>
       </div>
   )}
export default Navbar
```

Final feliz :)

¡Y con esto terminamos! Tenemos un theme context para utilizar en toda nuestra App sin la necesidad de tener que pasar por props.