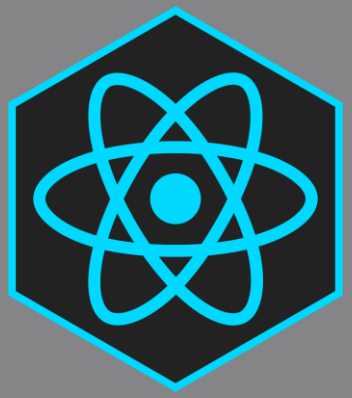


Parte 3: Comunicación entre componentes de React

UD4: Fundamentos de React

Después de este documento…

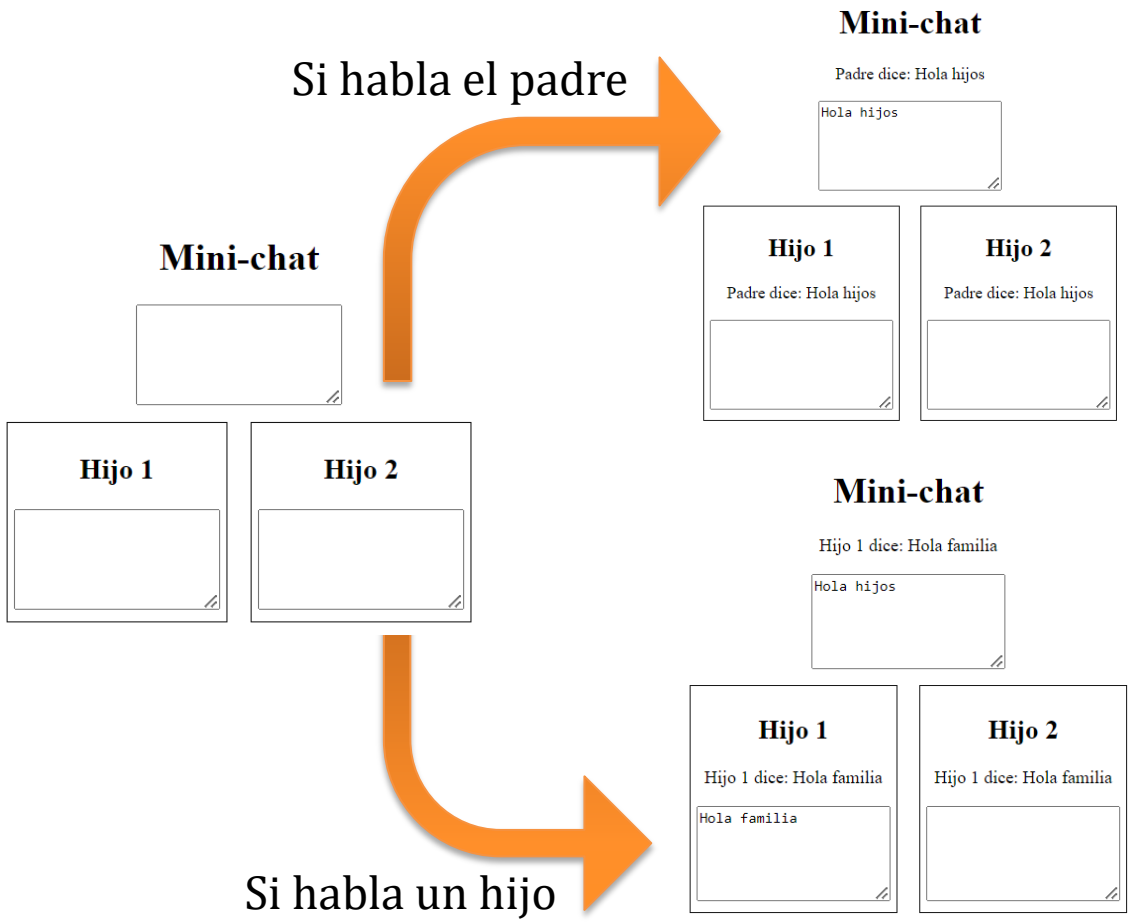
- Veremos aspectos relacionados con la **comunicación entre componentes**:
  - Cómo especificar **qué recibe** un componente (tipo de datos, etc.)
  - Los hijos también necesitan pasar información a los padres (**lifting, children, contextos**)
- Profundizaremos en aspectos de React que habíamos "pospuesto"
  - Cómo trabajar con estados que son de tipo **array**
  - Cómo trabajar con **formularios**
  - Como programar "efectos colaterales" con **useEffect**

Lifting (“alzamiento”)

- Necesidad de que una reacción en un hijo, provoque una reacción en un ancestro
  - Se define una función en el padre y se le pasa al hijo como prop
  - El hijo ejecuta la función cuando desea causar la reacción
  - Si el estado del padre cambia, se renderiza el padre y el hijo

EJERCICIO PROPUESTO I (1/2)

- **Mini-chat**
  - Función manejadora de onChange "**actualiza estado**" común para los tres componentes



Propiedad children

- Igual que es común anidar etiquetas html de esta forma:

```
<div>
  <img />
</div>
```

- También podemos anidar componentes

```
<Card>
  <Avatar />
</Card>
```

- El componente padre recibe el contenido a través de una prop llamada **children**

Ejemplo de children

- Usar este patrón permite que los estilos aplicados al componente padre se comparten con los hijos

```
function Layout(props) {
  return <div className="container">{props.children}</div>;
  }
function IndexPage() {
  return ( <Layout>
            <Header />
            <Hero />
            <Footer />
            </Layout> );
  }
function AboutPage() {
  return (<Layout>
          <About />
          <Footer />
          </Layout> );
  }
```

EJERCICIO PROPUESTO II

- Haz una nueva versión del ejercicio anterior donde apliques la propiedad **children**
  - Necesitarás buscar información sobre la función **cloneElement**:

```
return cloneElement(child,{
    mensaje: mensaje,
    componente: componente,
    actualizarMensaje: actualizarMensaje,
     });
```

- El programa principal será algo así:

```
<Padre>
 <Hijo1/>
 <Hijo2/>
</Padre>
```

useContext

- Hook que permite comunicar componentes funcionales a través del **contexto** en React
  - Evita tener que pasar propiedades
  - Si cambia el contexto, el componente se vuelve a renderizar
- ¿Cuál es el PERO?
  - Utilizar **Context** puede limitar la reutilización de componentes
- En general lo usaremos para:
  - Comunicar una gran cantidad de componentes
  - Comunicar componentes hermanos
  - Si existen componentes intermedios que no requieren hacer uso de las propiedades

Sintaxis del uso de contextos: Proveedor

- Importar **createContext**:

```
import { createContext } from 'react'
```

- Crear el objeto contexto:

```
export const Contexto = createContext();
```

- En el return crear el **provider**, estableciendo su valor en el atributo value:

```
  <Contexto.Provider value={{
            propiedad: valor,
            otrapropiedad: otrovalor
          }}>
…
  </Contexto.Provider>
```

Sintaxis del uso de contextos: Cliente

- Importar **useContext**:

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

- Importar el contexto – usando el nombre que se le haya dado en el proveedor:

```
import { Contexto } from './Proveedor';
```

- En el hilo principal del componente llamar a **useContext**:
  - Recibe como parámetro el contexto que hemos importado

```
const contexto = useContext(Contexto);
```

  - Se puede usar desestructuración para obtener directamente los valores guardados en el contexto

```
const {propiedad, otrapropiedad} = useContext(Contexto);
```

Ejemplo

```
const temas = {
 light: {
   foreground: "#000000",
   background: "#eeeeee"
   },
 dark: {
   foreground: "#ffffff",
   background: "#222222"
   }
  };
```

```
const TemasContext = createContext(temas.light);

function App() {
    return (
      <TemasContext.Provider value={temas.dark}>
        <BotonEstilo />
      </TemasContext.Provider>);

function BotonEstilo() {
  const tema = useContext(TemasContext);
  return (
    <button style={{ background: tema.background,
                     color: tema.foreground }}>
        Tengo el estilo del tema del contexto
    </button>);
  }
```

EJERCICIO PROPUESTO III

- **Mini-chat:** Realiza el mismo ejercicio, pero usando contextos de la siguiente forma:
  - En el padre…
    - Crea el contexto con **createContext** y expórtalo para que esté disponible en los hijos
    - Encapsula el JSX en una etiqueta **Context.Provider**
    - **A los hijos no se les pasará ninguna propiedad**
  - En los hijos…
    - Haz una llamada a **useContext** para obtener el mensaje, el componente, y la función manejadora
    - **RETO**: Busca información sobre `<Contexto.Consumer>` y pruébalo

Prop types

- Si a un componente no se le pasan el número y tipo de parámetros que necesita puede no mostrarse adecuadamente:

- **Prop types** permite ayudar al desarrollador con el tipo de datos que permiten los componentes

Uso de Prop types

- En las últimas versiones de React "se ha sacado" **propTypes** y ahora es una dependencia:
  - Instalar: `npm install prop-types --save`
  - Usar en el componente: `import PropTypes from 'prop-types';`

{} package.json > ...

```
1  {
2    "name": "holamundo",
3    "version": "0.1.0",
4    "private": true,
5    "dependencies": {
6      "@testing-library/jest-dom": "^5.16.5",
7      "@testing-library/react": "^13.4.0",
8      "@testing-library/user-event": "^13.5.0",
9      "prop-types": "^15.8.1",
```

```
Warning: Failed prop type: Invalid prop `text` of type `object` supplied to `Button`, expected `string`.
    at Button (http://localhost:3000/static/js/bundle.js:226:5)
```
react-jsx-dev-runtime.development.js:87

Ejemplo propTypes

- Sintaxis para usar **propTypes** en del componente:
- Permite definir:
  - Tipo de datos
  - Valores por defecto
  - Si es requerido
  - Número de hijos que puede tener…
  - Documentación

```
function Componente(props) {
  // código renderización
}
Componente.propTypes = {
  // definiciones propTypes
}
```

```
import PropTypes from 'prop-types';

function Saludo () {
    return (
        <h1>Hello,{this.props.nombre}</h1>
      );
  }
Saludo.propTypes = {
    nombre: PropTypes.string
  };
```

EJERCICIO PROPUESTO I (2/2)

- Define con **propTypes** que es obligatorio que los hijos reciban una función como parámetro

Estados con arrays

- Es necesario pasar un nuevo array a la función que actualiza el estado

Operación	NO (muta el array)	Sí (devuelve un nuevo array)
añadir	push, unshift	concat, [...arr]
eliminar	pop, shift, splice	filter, slice
reemplazar	splice, arr[i] = ... asigna	map
ordenar	reverse, sort	copia el array primero

Ejemplos con arrays

Escultores inspiradores:

Añadir

```
const [artists, setArtists] = useState([]);


    <button onClick={() => {
      setArtists([
        ...artists,
        { id: nextId++, name: name }
      ]);
    }}>Añadir</button>
```

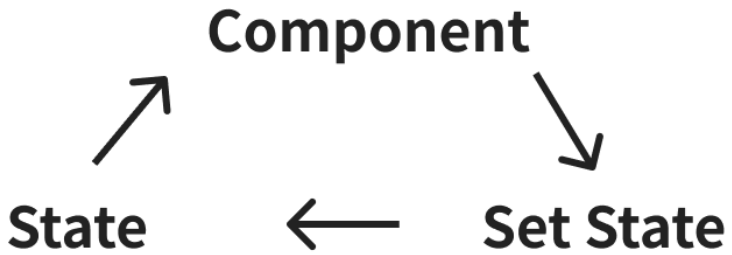
Escultores inspiradores:

- Marta Colvin Andrade Eliminar
- Lamidi Olonade Fakeye Eliminar
- Louise Nevelson Eliminar

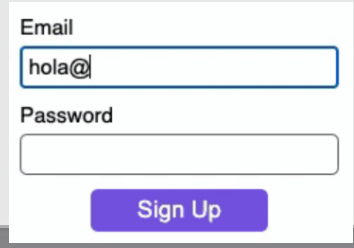
```
 let initialArtists = [
   { id: 0, name: 'Marta Colvin Andrade' },
   { id: 1, name: 'Lamidi Olonade Fakeye'},
   { id: 2, name: 'Louise Nevelson'},
   ];

const [artists, setArtists] = useState(
      initialArtists
    );
    <button onClick={() => {
      setArtists(
        artists.filter(a =>
          a.id !== artist.id
        )
      );
    }}>
      Eliminar
    </button>
```

Formularios en React

- Los elementos de formularios como <input>, <textarea> y <select> normalmente mantienen sus propios estados y los actualizan de acuerdo a la interacción del usuario (**componente controlado**)
  - El estado es la "única fuente de la verdad"
    - De acuerdo a la documentación oficial de React



Ejemplo de formulario




```
function Form() {
  const [values, setValues]=useState({
    email: "",
    password: "",
    });

  function handleSubmit(e) {
    e.preventDefault();
    // Aquí puedes usar values
    // para enviar la información
    }

  function handleChange(e) {
    const newValues = {
     ...values,
     [e.target.name]: e.target.value,
    };
    // Sincroniza el estado de nuevo
    setValues(newValues);
    }
```

```
return (
  <form onSubmit={handleSubmit}>
    <label htmlFor="email">Email</label>
    <input
      id="email"
      name="email"
      type="email"
      value={values.email}
      onChange={handleChange}
    />
    <label htmlFor="password">Password</label>
    <input
      id="password"
      name="password"
      type="password"
      value={values.password}
      onChange={handleChange}
    />
    <button type="submit">Sign Up</button>
  </form>
 );}
```

useEffect

- Una **función que se ejecutará cada vez que haya un cambio en el componente** (cambio de estado, recibe props nuevas, se está creando…)
- Es el segundo Hook más usado
  - Para llamar a una API
  - Para usar LocalStorage
  - …

Usando useEffect

- Es necesario importarlo:

```
import { useEffect } from 'react';
```

- En la llamada, este **hook** recibe como parámetros:
  - [OPCIONAL] Un array con las **dependencias** (elementos "escuchados" – los que queremos que al cambiar provoquen la ejecución de la función de useEffect)
    - Si queremos que sólo se ejecute una vez al crear el componente, se le pasa como segundo parámetro un array vacío:

```
useEffect(()=>{
  console.log("render");
},[])
```

Ejercicio ejemplo

```
import { useState, useEffect } from 'react';

function Example() {
   const [count, setCount] = useState(0);
   useEffect(() => {
      document.title = `Hiciste clic ${count} veces`;
      });
    return ( <div>
                <p>Hiciste clic{count} veces</p>
                <button onClick={() => setCount(count + 1)}>
                      Pínchame</button> </div> ); }
```

- El título se actualiza
  - Al ingresar en la página (se ejecuta **useEffect**)
  - Cada vez que hacemos clic en el componente (cambia el state, dispara renderizado, y se ejecuta **useEffect**)