¡Les damos la bienvenida!

¿Comenzamos?

Esta clase va a ser

- grabada

Encuesta
After Class

Por encuestas de Zoom

¡Coordinamos nuestro Segundo After Class!

CODERHOUSE

Clase 08. REACT JS

Workshop: Hooks, Children y Patrones

CODERHOUSE

Temario

07

Consumiendo API's

- ✓ Paradigmas de intercambio de información
- ✓ REQUESTS VIA HTTP/S
- ✓ REQUESTS EN EL BROWSER

08

Workshop: Hooks, Children y Patrones

- ✓ Custom Hooks
- ✓ Patrones

09

Routing y navegación

- ✓ Organicemos nuestra app
- ✓ React router

CODERHOUSE

Objetivos de la clase

- **Profundizar** los conocimientos sobre hooks y children.
- **Conocer** distintos patrones de reutilización.

CLASE N°7

Glosario

Componentes: básicamente, las aplicaciones en React se construyen mediante los mismos. permiten separar la interfaz de usuario en piezas independientes, reutilizables y pensar en cada pieza de forma aislada

Propiedades: son la forma que tiene React para pasar parámetros de un componente superior a sus hijos.

Estados: se utilizan para representar la información que puede cambiar durante la vida útil del componente y afectar su representación en la interfaz de usuario

Hooks

Hooks

Como vimos en la clase anterior, un hook es una función especial que permite a los desarrolladores utilizar el estado y otras características de React en componentes de función.

Los hooks son una adición relativamente nueva a la librería de React y fueron introducidos en la versión 16.8.

Hay varios hooks integrados en React, como **useState**, **useEffect**, **useContext**, **useRef** y otros.

Cada uno de ellos tiene una función específica y puede ser utilizado para diferentes propósitos en el desarrollo de aplicaciones web.

Reglas al utilizar hooks

Lo más importante es que los hooks deben ser **llamados únicamente en el nivel superior del componente funcional**, nunca dentro de bucles, condiciones o funciones anidadas.

Esto asegura que los hooks siempre sean llamados en el **mismo orden** en cada renderizado y mantener la relación con cada una de las fibras de React.

También es importante tener en cuenta que los hooks **no deben ser llamados desde funciones regulares**. Si se llama a un hook desde una función regular, se producirá un error en tiempo de ejecución.

React detecta que **una función es un componente** gracias a la convención **PascalCase** utilizada para nombrarlos.

Custom
Hooks

Custom Hooks

Ahora, pensemos...

¿De qué forma React detecta que una función es un hook y aplica las validaciones correspondientes para verificar que se cumplan las reglas?

Para eso, **React verifica el nombre de la función** y busca el **prefijo “use”**, con eso React sabe que **es un hook** y debe tratarlo como tal.

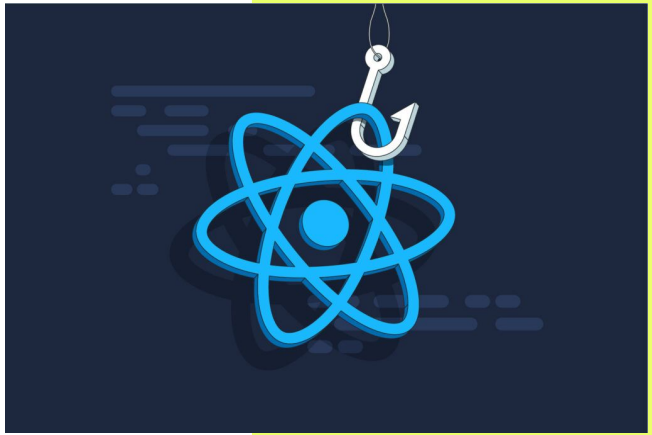
Custom Hooks

Entonces, ¿si yo creo una función con el prefijo “use” estaría creando mi propio hook?

La respuesta es **sí**. A esa función la llamamos **Custom Hook**.

¿Y de qué me sirve esto?

Si React aplica las validaciones de hooks a esta función, React permitirá escribir otros hooks dentro y así crear lógica de componentes reutilizable.



Custom Hooks

Entonces los hooks pueden ser escritos dentro de Componentes Funcionales y de Custom Hooks.

```
import { useState } from 'react'  4.1k (gzipped: 1.8k)

export const useCount = (initial = 0, min, max) => {
    if(initial < min || initial > max) initial = min

    const [count, setCount] = useState(initial)

    const decrement = () => {
        if(count > min) setCount(prev => prev - 1)
    }

    const increment = () => {
        if(count < max) setCount(prev => prev + 1)
    }

    const reset = () => {
        setCount(initial)
    }

    return { count, decrement, increment, reset }
}
```

```
import { useCount } from "../hooks/useCount"

const Counter = () => {
    const { count, decrement, increment, reset} = useCount(1, 0, 10)

    return (
        <div>
            <div>{count}</div>
            <div>
                <button onClick={decrement}>decrement</button>
                <button onClick={increment}>increment</button>
                <button onClick={reset}>reset</button>
            </div>
        </div>
    )
}

export default Counter
```

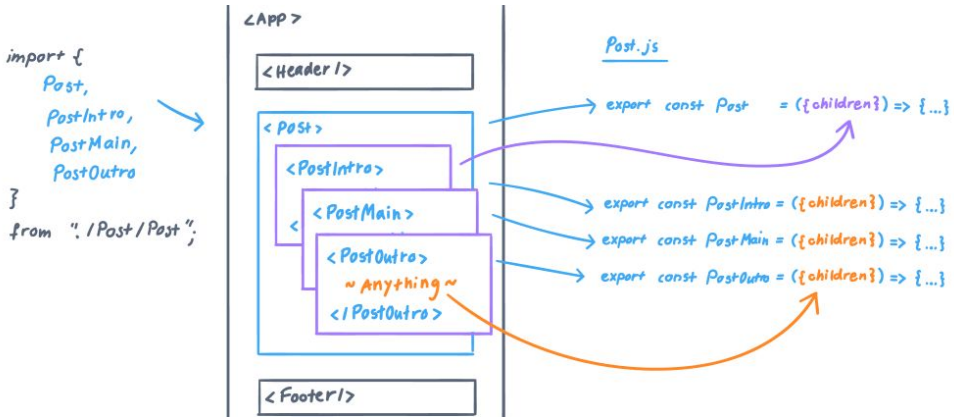
CODERHOUSE

Children

Children

El concepto de children es la forma en que React nos permite incluir un componente dentro de otro sin que al padre le importe exactamente qué componente o componentes hijos están siendo incluidos.

Los elementos hijos son aquellos que se colocan dentro de la etiqueta de apertura y cierre del componente y la prop “children” se utiliza para acceder y manipular estos elementos hijos desde dentro del componente padre.

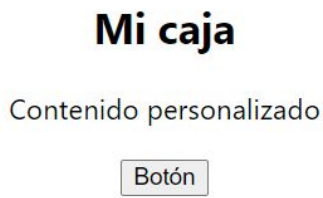


Fuente imagen: DEV

CODERHOUSE

Children

De esta manera, podemos crear componentes más flexibles y reutilizables que acepten contenido personalizado y que puedan renderizarlo de manera dinámica.




```jsx
const Box = (props) => {
    return (
        <div className="box">
          <h2>{props.title}</h2>
          <div className="content">
            {props.children}
          </div>
        </div>
      );
}

const Container = () => {
    return (
        <Box title="Mi caja">
            <p>Contenido personalizado</p>
            <button>Botón</button>
        </Box>
    )
}
```

CODERHOUSE

Patrones

Patrones

Hay formas de crear componentes que nos ayudan a reutilizar y mantener código.

Dos de ellas son **High Order Components** (HOC) y **Render Props**. Ambas son técnicas avanzadas.

Veamos de qué se trata cada una. 🤓

High Order Components

Este patrón permite reutilizar una funcionalidad común entre componentes. En pocas palabras, es una función que toma un componente como argumento y devuelve un nuevo componente con una funcionalidad adicional.

Un ejemplo simple de High Order Component podría ser uno que agregue la funcionalidad de validación de formularios a cualquier formulario que lo necesite.

High Order Components

Primero, creamos la función que va a retornar el componente con la funcionalidad de validación:

```
// Definimos un HOC que agregará la funcionalidad de validación de formulario a cualquier formulario que lo necesite
const withFormValidation = (WrappedComponent) => {
  const WithFormValidation = (props) => {
    const [errors, setErrors] = useState({});

    function validateForm() {
      // Aquí agregamos la lógica de validación del formulario
      // Por ejemplo, si el campo 'nombre' está vacío, agregamos un error correspondiente al estado 'errors'
      const newErrors = {};
      if (!props.formData.nombre) {
        newErrors.nombre = 'El nombre es requerido';
      }
      if (!props.formData.email) {
        newErrors.email = 'El email es requerido';
      }
      setErrors(newErrors);
    }

    return (
      <WrappedComponent
        {...props}
        errors={errors}
        validateForm={validateForm}
      />
    );
  };

  return WithFormValidation
}
```

High Order Components

Segundo, creamos el formulario que necesitaremos validar con todas las props necesarias para su funcionamiento:

```jsx
// Creamos un componente de formulario genérico que recibe formData, errors y validateForm como props
const Form = ({ formData, errors, validateForm, onChange }) => {
  function handleSubmit(event) {
    event.preventDefault();
    validateForm && validateForm();
  }

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Nombre:</label>
        <input type="text" name="nombre" value={formData.nombre} onChange={(e) => onChange(e)}/>
        {errors && errors.nombre && <div>{errors.nombre}</div>}
      </div>
      <div>
        <label>Email:</label>
        <input type="text" name="email" value={formData.email} onChange={(e) => onChange(e)}/>
        {errors && errors.email && <div>{errors.email}</div>}
      </div>
      <button type="submit">Enviar</button>
    </form>
  );
}
```

High Order Components

Tercero, creamos el componente que tendrá la validación y lo utilizamos:

```
// Usamos el HOC para crear un nuevo componente con la funcionalidad de validación de formulario
const FormWithValidation = withFormValidation(Form);

// Usamos el nuevo componente en nuestro código
const Container = () => {
  const [formData, setFormData] = useState({
    nombre: '',
    email: '',
  });

  function handleChange(event) {
    setFormData({
      ...formData,
      [event.target.name]: event.target.value,
    });
  }

  return (
    <div>
      <FormWithValidation formData={formData} onChange={handleChange} />
    </div>
  );
}

export default Container
```

Ejemplo en vivo

Revisemos este código en profundidad.

Componentes

Crear un componente genérico WithTitle

Duración: **15 minutos**

ACTIVIDAD EN CLASE

Children

Crear un componente **WithTitle** que reciba el parametro children para replicar la funcionalidad del componente HOC visto en clase.

Luego en el componente App, llamarlo con diferentes componentes hijos. Ejemplo:

- Button
- Input
- Imagen
- Lista

Break

¡10 minutos y volvemos!

Render Props

Este patrón implica pasar una función como una prop al componente hijo con el propósito de permitir al componente hijo renderizar su contenido a través de la función.

En otras palabras, un componente padre puede pasar una función como prop al componente hijo y el componente hijo puede llamar a esa función en su renderizado para obtener información o funcionalidad específica que necesita.

Render Props

El patrón de diseño Render Props puede ser útil en situaciones donde quieres **crear un componente reutilizable que puede ser personalizado con diferentes funciones o comportamientos** según las necesidades del usuario.

También es una alternativa al patrón de diseño HOC (Higher Order Component), que puede ser más complicado de entender y utilizar.

Por ejemplo, imagina un componente que muestra una lista de tareas y un componente de filtro que se utiliza para filtrar las tareas según su estado.

Podrías pasar una función como prop del componente de filtro, que devuelve la lista filtrada de tareas, y el componente de filtro llamaría a esta función cuando se produce un cambio en su estado para actualizar la lista de tareas que se muestra.

Render Props

Primero, creamos el componente que se encarga de mostrar el listado de tareas:

```
function TaskList({ tasks }) {
  return (
    <ul>
      {tasks.map((task) => (
        <li key={task.id}>{task.title}</li>
      ))}
    </ul>
  );
}
```

Render Props

Segundo, creamos el componente que se encarga de filtrar tareas:

```
function Filter({ children }) {
  const [filterState, setFilterState] = useState("all");

  const handleFilterChange = (event) => {
    setFilterState(event.target.value);
  };

  // llamamos a la función "children" pasando el estado actual de filtrado
  return children(filterState, handleFilterChange);
}
```

Render Props

Tercero, utilizamos ambos componentes y sumamos toda la funcionabilidad de selección de filtro dentro de la función:

```
function Container() {
    const tasks = [
      { id: 1, title: "Comprar leche", completed: true },
      { id: 2, title: "Ir al gimnasio", completed: false },
      { id: 3, title: "Cocinar cena", completed: false },
    ];

    // pasamos una función como "children" del componente "Filter"
    return (
      <div>
        <Filter>
          {(filterState, handleFilterChange) => (
            <div>
              <label htmlFor="filter">Filtrar por estado:</label>
              <select id="filter" value={filterState} onChange={handleFilterChange}>
                <option value="all">Todos</option>
                <option value="completed">Completados</option>
                <option value="uncompleted">Sin completar</option>
              </select>
              <TaskList
                tasks={
                  filterState === "all"
                    ? tasks
                    : tasks.filter((task) =>
                        filterState === "completed" ? task.completed : !task.completed
                      )
                }
              />
            </div>
          )}
        </Filter>
      </div>
    );
  }
```

Ejemplo en vivo

Revisemos este código en profundidad.

Diferencia entre Render Props y HOC

Tanto los Render Props como los High Order Components (HOC) son técnicas avanzadas en React que nos permiten reutilizar lógica y abstraer comportamientos comunes en nuestros componentes.

La **principal diferencia** radica en que Render Props se basa en la idea de pasar una función de renderizado como una prop a un componente para que pueda renderizar información, mientras que los HOC son componentes de orden superior que envuelven otro componente y le agregan alguna funcionalidad.

Conclusión

En pocas palabras, los **Render Props** nos permiten compartir la lógica de un componente a través de una función de renderizado, mientras que los **HOC** nos permiten compartir la lógica de un componente a través de la composición de componentes.

En cuanto a cuándo usar uno u otro, depende del caso de uso específico. Tanto los Render Props como los HOC son técnicas poderosas que nos permiten escribir componentes más reutilizables y mantenibles, por lo que es importante tenerlos en cuenta en nuestra caja de herramientas de React.

Segunda pre-entrega

En la clase que viene se presentará la segunda parte del Proyecto Final, que nuclea temas vistos entre las clases 5 y 7.
Recuerda que tendrás 7 días para subirla en la plataforma a partir del momento en que se habilite la entrega.

Resumen
de la clase hoy

- ✓ Custom Hooks
- ✓ Children
- ✓ High Order Components
- ✓ Render Props

Opina y valora esta clase

Muchas gracias.

#DemocratizandoLaEducación