¡Les damos la bienvenida!

¿Comenzamos?

Esta clase va a ser

- grabada

Clase 05. REACT JS

Componentes II

Temario

04

Componentes I

- ✓ Componentes I: Introducción
- ✓ Patrones

05

Componentes II

- ✓ Componentes II: Introducción
- ✓ Efectos

06

Promises, asincronía y MAP

- ✓ Promise
- ✓ MAP

CODERHOUSE

Objetivos de la clase

- **Analizar** cambios de estados y efectos secundarios.
- **Comprender** cómo aplicar props, children, estados y sincronización de componentes.

CLASE N°04

Glosario

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

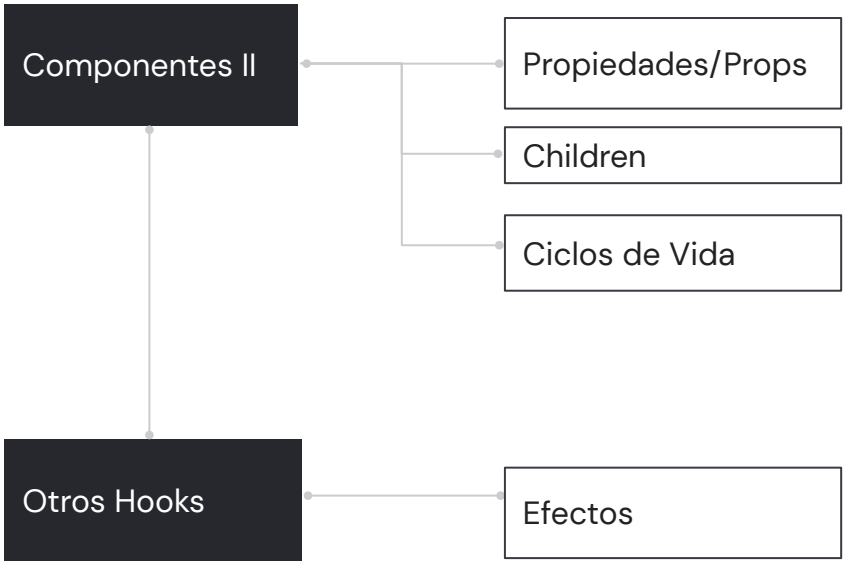
Propiedades: son la forma que tiene React para pasar parámetros de un componente superior a sus hijos.

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

Componentes contenedores: tienen como propósito encapsular a otros componentes y proporcionarles las propiedades que necesitan. Además se encargan de modificar el estado de la aplicación para que el usuario vea el cambio en los datos (por eso son también llamados state components).

Componentes de presentación: son aquellos que simplemente se limitan a mostrar datos y tienen poca o nula lógica asociada a manipulación del estado (por eso son también llamados stateless components).

MAPA DE CONCEPTOS



Componentes II:
Introducción

Propiedades/Props

Props: ¿un espacio multipropósito?

No están limitadas a ser valores fijos como: **1 / “Alexis” / true**

Pueden ser lo que sea:

- ✓ Valores comunes:
  - num, bool, array, obj
- ✓ Funciones
- ✓ Componentes. Si los componentes son funciones, ¡entonces puedo pasar componentes!
- ✓ Children
- ✓ Valores inyectados por librerías:
  - location, rutas, traducciones

Children

Children

Children es una manera que tiene react de permitirnos proyectar/transcluir uno o más componentes dentro otro.

Es ideal cuando:

- ✓ Necesitamos que un elemento quede dentro de otro, sin que sepan el uno del otro.
- ✓ Necesitamos implementar patrones más complejos.

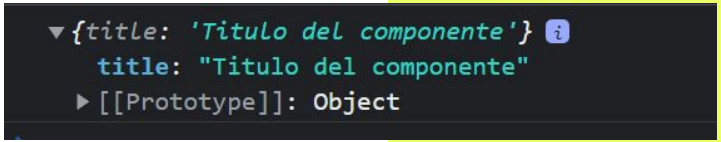
Relación entre Children y props

React inyecta automáticamente **children** en las props, sólo si encuentra alguno.

En este ejemplo **<Layout>** no tiene children.

```jsx
const Layout = (props) => {
  console.log(props)
  return (
    <div>
      <h1>{props.title}</h1>
    </div>
  );
}


const App = () => {
  return (
    <div>
      <Layout title={'Titulo del componente'} />
    </div>
  );
}
```



CODERHOUSE

Relación entre Children y props

Si le agregamos children en el JSX...

Los inyecta **como objeto** si es único o **como array** si son muchos.

Tener cuidado para evitar errores del tipo **children[0]** o cuando estoy utilizando un método de array**.**

Profundizaremos esto en el Workshop sobre Hooks, Custom Hooks y Children.

```jsx
const Layout = (props) => {
  console.log(props)
  return (
    <div>
      <h1>{props.title}</h1>
      {props.children}
    </div>
  );
}


const App = () => {
  return (
    <div>
      <Layout title={'Titulo del componente'}>
        <p>Este párrafo se mostrará dentro del componente Ejemplo</p>
      </Layout>
    </div>
  );
}
```

CODERHOUSE

Ejemplo en vivo

Vamos al código

Ciclos de vida

Ciclos de vida

El ciclo de vida no es más que una serie de estados por los cuales pasa todo componente a lo largo de su existencia.

1

El montaje

Se produce la primera vez que un componente va a generarse, incluyéndose en el DOM.

2

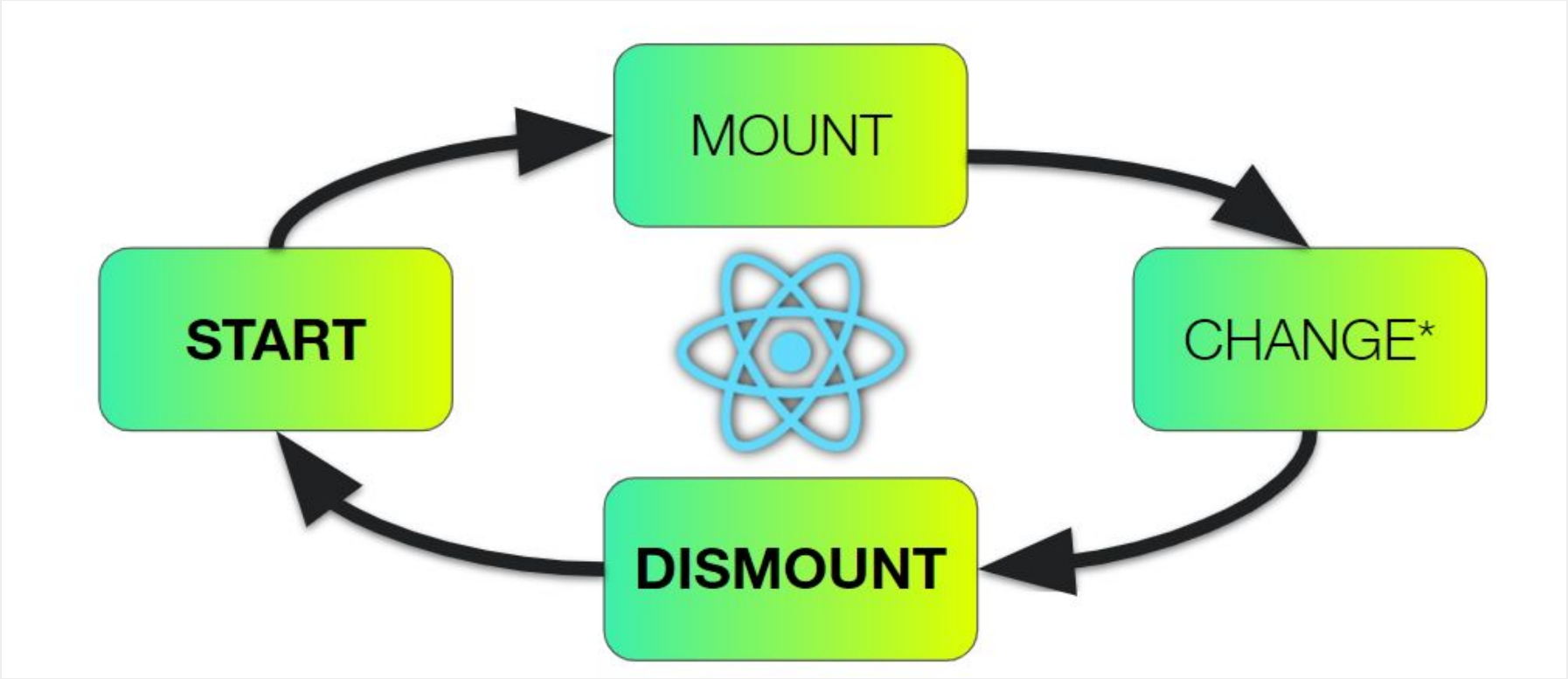
La actualización

Se produce cuando el componente ya generado se está actualizando.

3

El desmontaje

Se produce cuando el componente se elimina del DOM.



El hijo tendrá la posibilidad de cambiar todas las veces que quiera hasta que el componente que lo generó lo destruya.

Otros Hooks

Hooks ¿cuáles y cuántos son?

Como vimos anteriormente los hooks son funciones que le dan funcionabilidad adicional a nuestros componentes funcionales.

- ✓ **Hooks básicos:**
  - useState: para crear un estado
  - useEffect: para sincronizar componentes con sistemas externos
  - useContext: para usar un contexto

Además también existen:

- ✓ **Hooks adicionales:** useReducer, useCallback, useMemo, useRef, useImperativeHandle, useLayoutEffect, useDebugValue, useDeferredValue, useTransition, useId.
- ✓ **Hooks de librería:** useSyncExternalStore, useInsertionEffect.

Poco a poco aprenderemos a usar los más importantes...

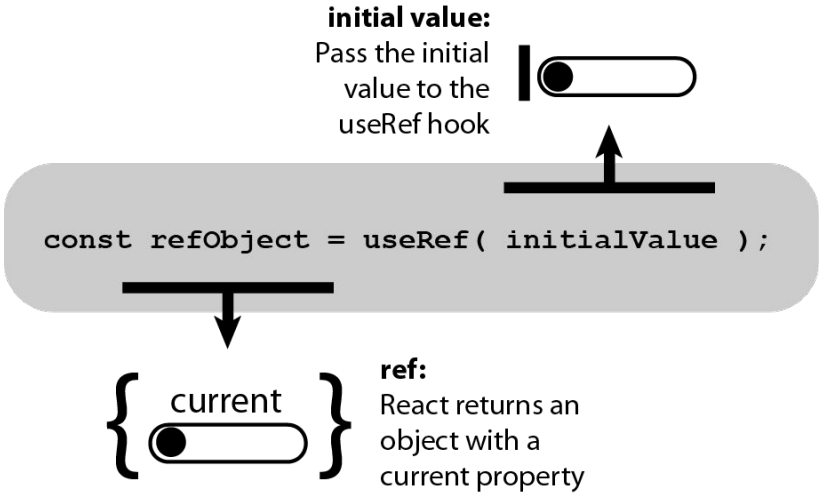
useRef

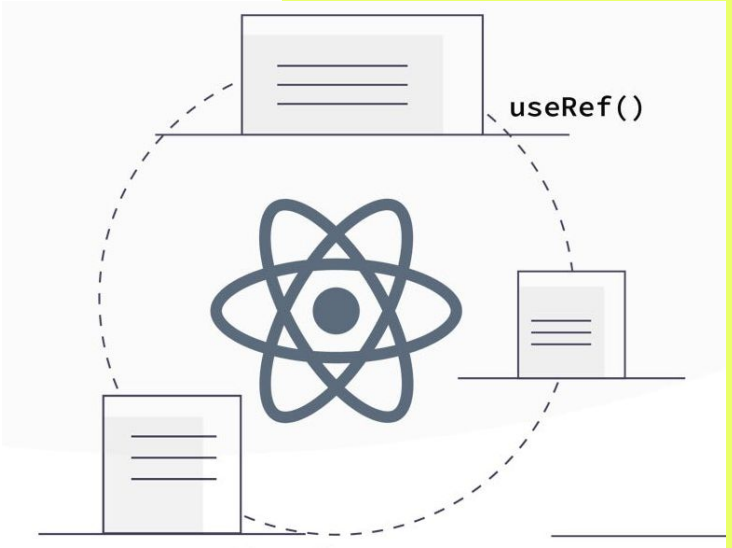
CODERHOUSE

useRef

El hook useRef se utiliza para crear una referencia mutable.

Este hook retorna un objeto con una propiedad current la cual apuntará a un valor y podremos mutar.





CODERHOUSE

useRef

useRef se utiliza a menudo para acceder a elementos del DOM y modificarlos directamente.

Esto puede ser especialmente útil cuando se trabaja con librerías de terceros que requieren acceso directo a elementos del DOM.

Es recomendable, siempre que sea posible, dejarle a React la responsabilidad de manejar el DOM.

```jsx
const App = () => {
  const divRef = useRef(null);

  const handleClick = () => {
    divRef.current.innerHTML = 'Nuevo contenido';
  };

  return (
    <div>
      <div ref={divRef}>Contenido original</div>
      <button onClick={handleClick}>Cambiar contenido</button>
    </div>
  );
}
```

CODERHOUSE

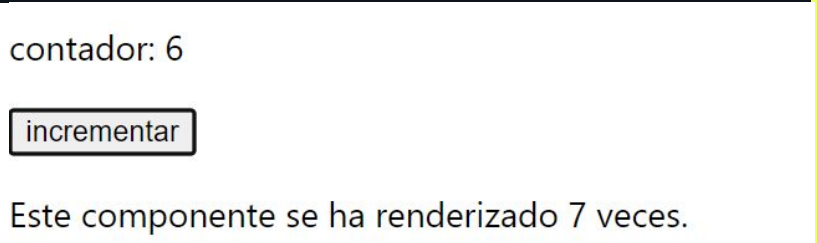
useRef

Además, useRef también puede ser utilizado para mantener valores persistentes en el componente, lo que puede ser útil para realizar cálculos o para almacenar datos temporales que no deben ser expuestos al estado del componente.

```jsx
const App = () => {
  const [count, setCount] = useState(0)
  const renderCount = useRef(0);

  renderCount.current++;

  return (
    <div>
      <p>contador: {count}</p>
      <button onClick={() => setCount(count => count + 1)}>incrementar</button>
      <p>Este componente se ha renderizado {renderCount.current} veces.</p>
    </div>
  );
}
```



Render y efectos

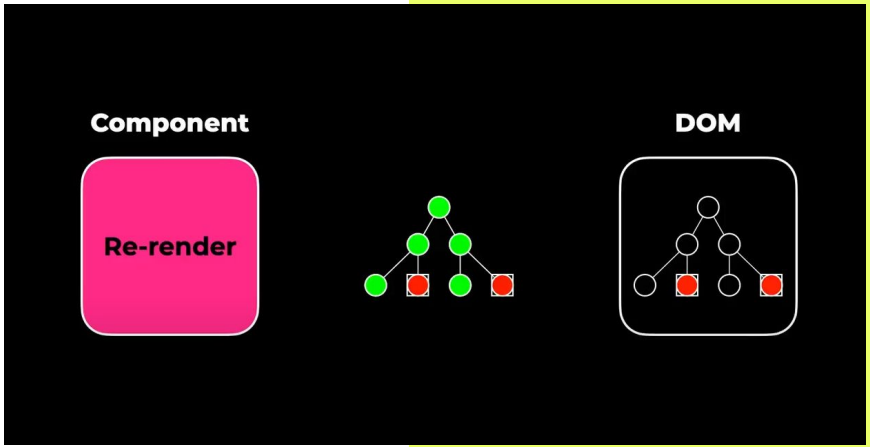
CODERHOUSE

Render y efectos

El funcionamiento de React está estrechamente ligado al cambio de estados.

Cuando se produce un cambio de estado, React ejecuta un nuevo proceso de renderizado para ese componente y, de manera recursiva, para todos sus componentes hijos.

Durante este proceso React ejecuta de nuevo las funciones de los componentes para generar el nuevo árbol de elementos de la interfaz de usuario.



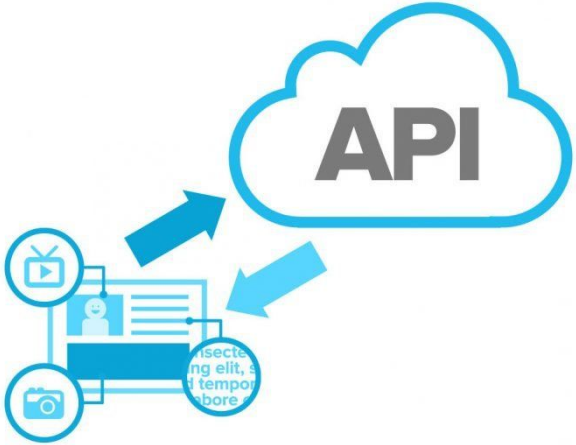
CODERHOUSE

Render y efectos

Dentro de los componentes podemos tener funciones para realizar **diferentes tareas**, como, por ejemplo, una llamada a una api para obtener datos.

Si en cada render se ejecuta esta llamada a la API, quiere decir que **un cambio de estado está provocando efectos secundarios** y no controlarlos puede traernos problemas.

Entonces, ¿cómo los controlamos?



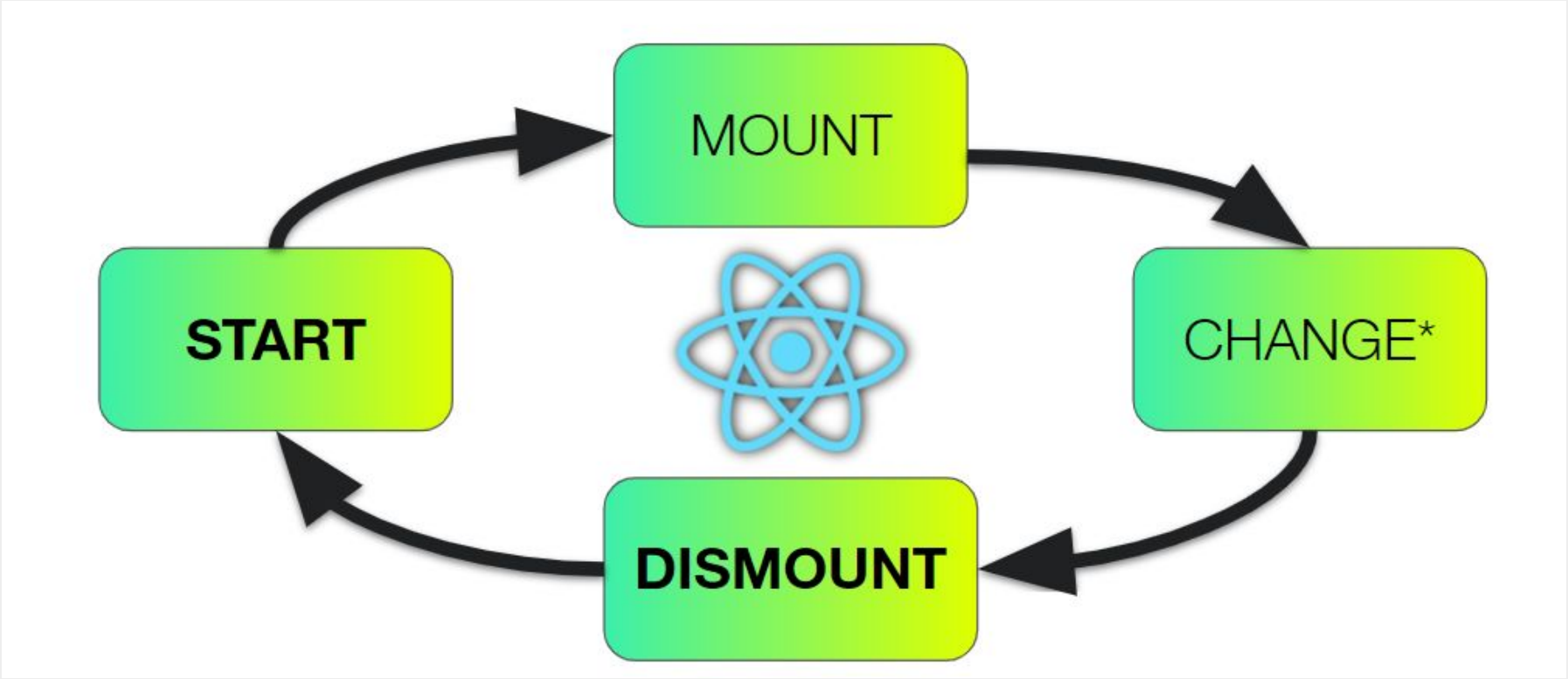
***Trabajaremos con promesas y APIs en las próximas clases**

CODERHOUSE

Break

¡10 minutos y volvemos!

useEffect



El hijo tendrá la posibilidad de cambiar todas las veces que quiera hasta que el componente que lo generó lo destruya.

useEffect

useEffect va a ser el hook que nos permita controlar efectos secundarios provocados por cambios de estados. Utilizado normalmente para sincronizar un componente con un sistema externo.

Este hook recibe dos argumentos:

- El primero es una **función de callback** con acciones a ejecutar.
- El segundo es un **array de dependencias**, donde se indicará qué estados (pueden venir por props) deben cambiar para que se vuelva a ejecutar la función del primer argumento.

useEffect

Para completar este array de dependencias la pregunta que debo hacerme es:
¿Con que estados debo sincronizar este efecto?

useEffect(fn) // Con todos los estados

useEffect(fn, []) // Con ningún estado

useEffect(fn, [esos, estados])

Hay que tener en cuenta que useEffect se ejecuta siempre después del renderizado y por lo menos una vez al montar el componente.

```
const App = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log(`El valor del contador es ${count}`);
  }, [count]);

  const increment = () => {
    setCount(prev => prev + 1);
  }

  return (
    <div>
      <p>contador: {count}</p>
      <button onClick={increment}>Incrementar count</button>
    </div>
  )
}
```

CODERHOUSE

useEffect

En la función de callback escribiremos el bloque de código que deseamos sincronizar en el cuerpo y retornaremos una función, llamada función de limpieza, que se ejecutará tantas veces como se haya ejecutado la función que la retorna.

Es decir, que **el proceso es simétrico.**

```
const App = () => {
  const [laps, setLaps] = useState(0)
  const [timeInSeconds, setTimeInSeconds] = useState(0);

  useEffect(() => {
    setTimeInSeconds(0)

    const intervalId = setInterval(() => {
      setTimeInSeconds(timeInSeconds => timeInSeconds + 1);
    }, 1000);

    return () => {
      clearInterval(intervalId);
    };
  }, [laps]);

  return (
    <div>
      <p>Vueltas: {laps}</p>
      <p>Tiempo en segundos: {timeInSeconds}</p>
      <button onClick={() => setLaps(laps => laps + 1)}>Finalizar Vuelta</button>
    </div>
  );
}
```

CODERHOUSE

useEffect

Dado que la función se ejecutará por lo menos una vez al montar el componente, la función que se retorna se ejecutará una vez al desmontar el componente.

En caso de tener dependencias de estados, la limpieza se ejecutará antes de volver a ejecutar el efecto frente a un cambio en una de sus dependencias.

Acción => Limpieza => Acción => Limpieza

*Actualmente el modo estricto de React ejecuta dos veces la función del primer argumento. Por eso puede ser que veas dos veces un mismo console.log que se ejecutó dentro de esta función.

useEffect

Como podemos utilizar todos los useEffect que deseemos, es conveniente separar responsabilidades.

Cada useEffect encargándose de un solo efecto y limpieza.

```
const ChatFeed = () => {

  useEffect(() => {
    //subcripcion al feed
    return () => {
      //desubscripcion del feed
    }
  })

  useEffect(() => {
    //configurar el titulo del documento
    return () => {
      //restaurar el titulo del documento
    }
  })

  useEffect(() => {
    //subscribirse a la geo localizacion
    return () => {
      //desuscribirse de la geo localizacion
    }
  })

  return <div>{ /* UI de Chat App */ }</div>
}
```

#Coderalert

Ingresa al manual de prácticas y realiza la primera actividad “Contador con botón”. Ten en cuenta que el desarrollo de la misma será importante para la resolución del Proyecto Final.

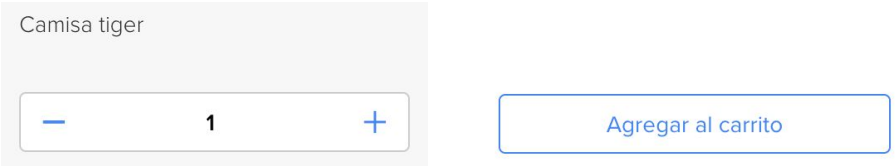
Actividad Nº 1

Segunda pre-entrega PF

Contador con botón

Descripción de la actividad.

- ✓ Crea un componente ItemCount.js, que debe estar compuesto de un botón y controles, para incrementar y decrementar la cantidad requerida de ítems



No es necesario usar este estilo, sirve a modo de orientación

CODERHOUSE

Actividad Nº 1

Contador con botón

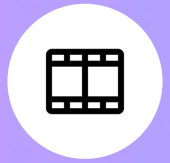
Recomendaciones.

- ✓ El número contador nunca puede superar el stock disponible.
- ✓ De no haber stock el click no debe tener efecto y por ende no ejecutar el callback onAdd.
- ✓ Si hay stock al clickear el botón se debe ejecutar **onAdd** con un número que debe ser la cantidad seleccionada por el usuario.

Tener en cuenta.

- ✓ Como sabes, todavía no tenemos nuestro detalle de ítem y este desarrollo es parte de él, así que por el momento puedes probar e importar este componente dentro del **ItemListContainer**, solo a propósitos de prueba. Después lo sacaremos de aquí y lo incluiremos en el detalle del ítem.

¿Preguntas?



¿Quieres saber más?
Te dejamos material ampliado de la clase

CODERHOUSE

MATERIAL AMPLIADO

Recursos

Artículos

✓ https://beta.reactjs.org/reference/react/useEffect | **React useEffect**

Disponible en nuestro repositorio.

Resumen
de la clase hoy

- ✓ Props, children y estados.
- ✓ Efectos

Opina y valora esta clase

Muchas gracias.

#DemocratizandoLaEducación