Consumo de APIs con hook useEffect

 ¡Hola! Te damos la bienvenida a esta nueva guía de ejercicios.

¿En qué consiste esta guía?

En la siguiente guía podrás trabajar los siguientes aprendizajes:

- Utilizar el hook useEffect para controlar efectos secundarios en aplicaciones react.
- Aplicar useEffect para el consumo de información proveniente de una API.
- Conocer el uso del hook useRef para referenciar y modificar elementos del DOM.

En este material de estudio, abordaremos el manejo de efectos secundarios que pueden generarse en una aplicación de React. Generalmente, estos estarán enfocados en el consumo y carga de información proveniente de una API, para lograrlo, necesitamos primero aprender a utilizar el hook useEffect.

¡Vamos con todo!

Tabla de contenidos

¡Comencemos!

¿Por qué tenemos que aprender sobre useEffect?

useEffect es una función que permite que un componente pueda ejecutar alguna acción después de haber sido renderizado, por ejemplo:

- Actualizar el DOM, cambiando el título del sitio a partir de ciertos cambios o acciones que se ejecuten.
- Leer alguna base de datos, conectarse a una API y extraer la información de ella.
- Escribir en una base de datos, conectarse a una API para hacer POST de información.
- Todas aquellas acciones que puedan cambiar el resultado de nuestros componentes.

Este tipo de acciones, que no se hacen durante el rendereo del componente, reciben el nombre de **Side Effects** o efectos secundarios. Es por eso que la función se llama useEffect, y al igual que useState, que es un hook.

Tenemos que aprender a utilizar useEffect para aprender a trabajar con APIs.

Introducción a useEffect

El ejemplo inicial de la documentación oficial nos da una idea de cómo funciona useEffect

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

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Imagen 1: Introducción a useEffect.
Fuente: React Js

En la imagen vemos un componente llamado `Example()`, el cual tiene un estado llamado `count` que inicia su valor en cero. Además, existe un evento `onClick` que al hacer clic sobre el botón aumenta el contador en 1.

Lo interesante en este caso es que por cada cambio que se registre en el estado a través del evento onClick, se ejecutará por medio del `useEffect` un cambio en el título del documento HTML. Entonces, el efecto secundario que se realiza a partir de la interacción del clic en el botón es que se actualice el título del documento.

Actividad 1: Crear el proyecto del ejemplo

Realiza los siguientes pasos, utilizando el ejemplo de la imagen anterior:

1. Crea un proyecto desde cero usando Vite llamado **ejemplo-efecto**.
2. Elimina el código del componente App.jsx por el código que muestra la documentación. Puedes copiar directamente el código:

```
// App.jsx
import { useState, useEffect } from 'react'
import './App.css'

function App() {
  const [count, setCount] = useState(0)

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`
  })

  return (
    <>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </>
  )
}

export default App
```

3. Abre el proyecto con el navegador.
4. Observa el cambio del título del tab cuando se hace clic en el documento.
5. Abre el inspector de elementos en el tab componente.

a. Hace clic en App.
b. Observa con la herramienta react-developer-tools los cambios en el state y en el effect.

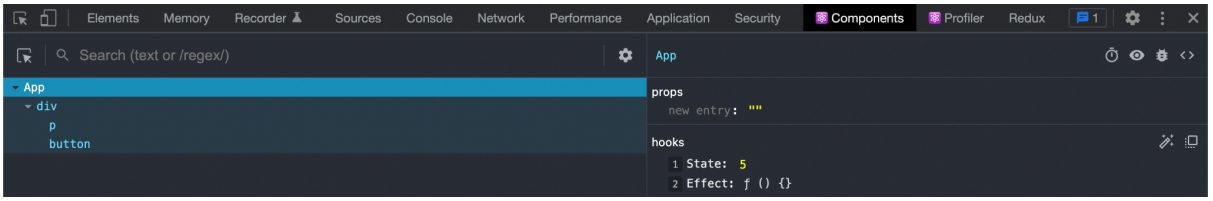

Imagen 2: Observando los hooks en el navegador
Fuente: Desafío Latam.

useEffect y el ciclo de vida

useEffect se ejecuta en las siguientes fases que forman parte del ciclo de vida de un componente:

1. **Montaje**: El componente es cargado la primera vez y mostrado en el DOM.
2. **Actualización:** El componente recibe alguna actualización de su estado inicial.
3. **Desmontaje:** Etapa donde el componente se remueve del DOM.

Lo importante que tenemos que aprender es, que debido a que useEffect se ejecuta cada vez que se actualiza el componente, podemos afectar negativamente el rendimiento de la aplicación sin darnos cuenta.

Para demostrar esto, puedes cambiar el código del archivo App.jsx, para así tener un input para escribir un nombre, que guardaremos en un estado, y utilizaremos useEffect para mostrar un mensaje en la consola. Observaremos que este mensaje aparecerá cada vez que cambiemos algo en el input.

```
// App.jsx
import { useState, useEffect } from 'react'

function App() {
 const [nombre, setNombre] = useState("");

 useEffect(() => {
   console.log("hola")
 })

 return (
   <div className="App">
     <h1>Manejo de efectos en react</h1>
     <form>
```

```
        <input
          type="text"
          placeholder="nombre"
          name={nombre}
          value={nombre}
          onChange={(e) => setNombre(e.target.value)}
        />
      </form>
    </div>
  );
}

export default App
```

Al abrir nuestra aplicación en el navegador, luego la consola y escribir en el input, observaremos que aparece un mensaje con un "hola" cada vez que agregamos o borramos una letra en el input.

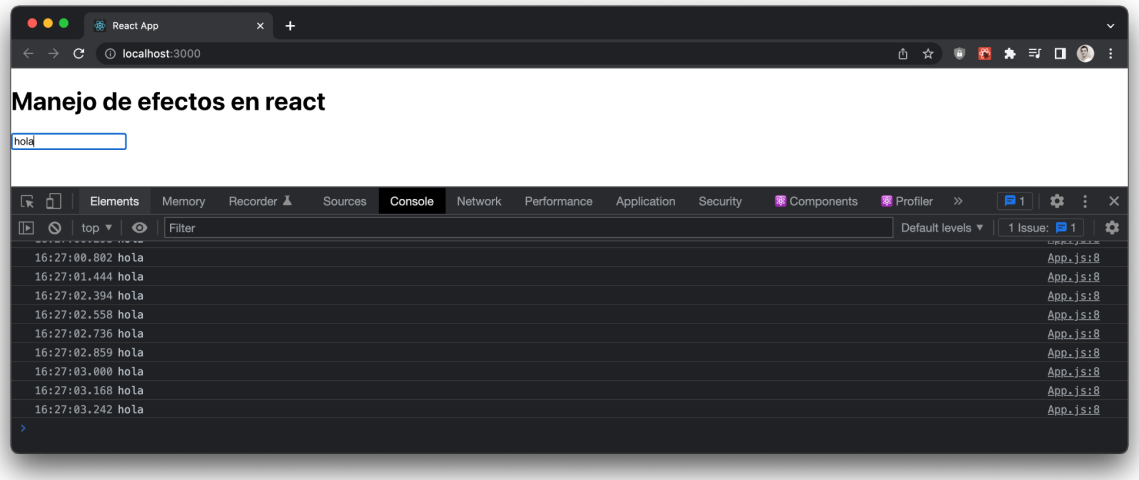

Imagen 3: Llamado al callback en la actualización de la componente
Fuente: Desafío Latam.

¿Por qué es un problema que useEffect se ejecute después de cada actualización del componente?

1. Afecta negativamente el rendimiento.
2. Podemos estar llamando varias veces a una API sin darnos cuenta.
    a. Hay APIs que tienen límite de uso por tiempo, por ejemplo, solo 1 llamado por cliente por cada segundo, entonces si se ejecuta múltiples veces se bloquea.

Para resolver el problema, haremos que useEffect se ejecute únicamente en la fase de montaje. Lograr esto es muy sencillo, simplemente agregaremos un argumento más a useEffect, el cual será un arreglo vacío. Más adelante explicaremos el motivo de esto.

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

Al recargar la página veremos que ahora el mensaje aparece 2 veces y luego, al escribir en el input no vuelve a aparecer.

¿Por qué aparece 2 veces? Esto se debe a que React está ejecutándose en modo estricto y, por lo tanto, en desarrollo (no en producción) el código se renderiza dos veces para detectar fallos.

¿Qué significa el arreglo vacío que estamos pasando a useEffect? Este puede recibir opcionalmente un arreglo de dependencias, si hay dependencias especificadas useEffect se llama solo cuando haya un cambio en alguna de ellas, si el arreglo está vacío se está especificando que no depende de nada y, por lo tanto, solo se llamará a la función al momento del montaje

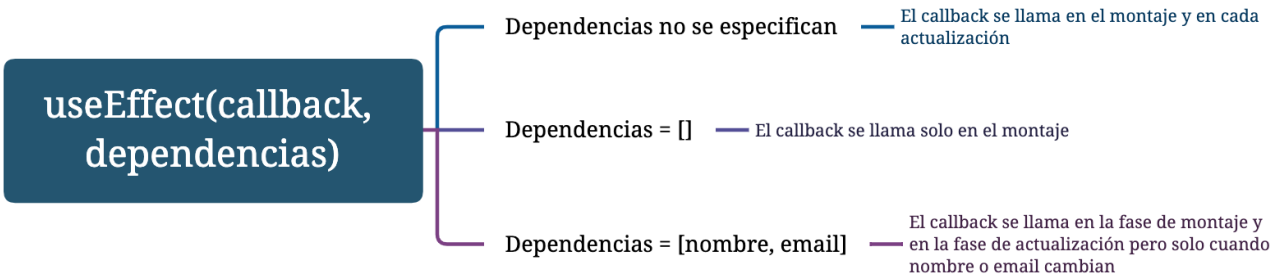

Imagen 4: Diagrama explicando alternativas posibles al momento de usar useEffect
Fuente: Desafío Latam.

Nota:
Cuando trabajemos con APIs haremos únicamente el llamado al momento del montaje.

Consumiendo una API con useEffect()

A continuación, haciendo uso del hook useEffect, mostraremos una frase famosa aleatoria de personajes de la serie *Game Of Thrones*, para esto utilizaremos la siguiente API.
Es decir, aprenderemos a crear una aplicación en React que consuma una API con useEffect.

Para crear una componente que muestre datos consumidos de una API necesitamos:

1. Una función que nos devuelva los resultados de la API.
2. Llamar a la función creada desde useEffect.
3. Un estado, donde guardaremos los valores que nos devuelve la API.
4. Utilizar el estado en el return de la página para mostrar los datos.

```
// App.jsx
import { useState, useEffect } from "react";
import "./App.css";

function App() {
  // 3 - info guardará los valores traídos desde la API
  const [info, setInfo] = useState([]);

  // 2 - Llamamos a la función consultarApi al momento de montar el
componente
  useEffect(() => {
    consultarApi();
  }, []);

  // 1 - Función que consulta la API
  const consultarApi = async () => {
    const url = "https://api.gameofthronesquotes.xyz/v1/random";
    const response = await fetch(url);
    const data = await response.json();
    setInfo(`${data.sentence} - ${data.character.name}`); // Con setInfo
actualizamos el estado
  };

  return (
    <>
      {/* Mostramos la info */}
      {info}
```

```
    </>
  );
}

export default App;
```

Al cargar la aplicación veremos una frase de la serie de Game Of Thrones:

A Lannister always pays his debts. - Tyrion Lannister

Imagen 5: Screenshot de un componente que consume una API con React funcionando.
Fuente: Desafío Latam.

Nota: Podrás acceder al repositorio de este ejercicio en el LMS con el nombre **Repositorio Ejercicio - GOT API**

Encontrarás otro ejercicio relacionado a **useEffect** dentro del documento "Material Complementario - Manejo de efectos en React".

Conozcamos el hook useRef

Otro de los hooks más conocidos de React es useRef. Este permitirá almacenar una referencia a algún elemento del DOM y acceder a sus métodos.
En otras palabras, permitirá acceder y modificar elementos del DOM directamente, almacenar valores mutables a lo largo del ciclo de vida del componente y evitar renderizaciones innecesarias.

Para comprender su uso veamos un ejemplo:

- Creamos un nuevo proyecto llamado **ejemplo-ref**.
- Se debe limpiar el proyecto eliminando los archivos innecesarios y eliminando las líneas de código que no utilizaremos en el archivo App.jsx

En este ejemplo crearemos un formulario de ejemplo con dos inputs, pero el primero de ellos se enfocará automáticamente al cargar la página, para esto haremos uso de los hooks useEffect y useRef.

```
// App.jsx

// Importamos useRef de igual forma que hemos importado los otros hooks
import { useRef} from 'react'
import './App.css'

function App() {

  return (
    <>
      <h1>Ejemplo useRef - Focus automático</h1>
      <input type="text" placeholder="Nombre" />
      <input type="text" placeholder="Correo electrónico" />
      <input type="text" placeholder="Teléfono" />
    </>
  )
}
```

Si vemos nuestra página, vemos que para poder ingresar datos en el formulario, debemos hacer clic en uno de los campos.

- Luego de esto, crearemos una referencia llamada inputRef y la asignamos al campo de entrada.
- Utilizaremos useEffect para ejecutar la función una vez que el componente se haya montado en el DOM.
- Dentro de useEffect ejecutaremos el método focus() para enfocar el campo del nombre.
- En el campo Nombre, agregamos el atributo ref para asignar la referencia que creamos.

El código completo quedaría de la siguiente manera:

```
// App.jsx

import { useRef} from 'react'
import './App.css'

function App() {
  // Creamos una referencia utilizando useRef() y la asignamos a la
constante inputRef, esta referencia permitirá acceder al elemento del DOM
correspondiente al campo de entrada
  const inputRef = useRef()

  useEffect(() => {
```

```
    // Dentro de esta función, llamamos a la referencia inputRef.current
para tener acceso directo al elemento actual al que se está haciendo
referencia e invocamos al método focus()
    inputRef.current.focus();
  }, []);

  return (
    <>
      <h1>Ejemplo useRef - Focus automático</h1>

      // El atributo ref={inputRef} asigna la referencia al campo de
entrada, esto permitirá acceder a él y aplicar el enfoque.
      <input type="text" placeholder="Nombre" ref={inputRef}/>
      <input type="text" placeholder="Correo electrónico" />
      <input type="text" placeholder="Teléfono" />
    </>
  )
}
```

Ahora, si recargamos la página, el campo del nombre se enfoca automáticamente.

Si bien este ejercicio se pudo realizar solo con useState en lugar de useRef, si usamos useState este activaría una nueva renderización del componente, en cambio, useRef no provoca nuevas renderizaciones cuando se modifica el valor de la referencia, lo que mejora el rendimiento de la página.

DOM virtual

El DOM es la abreviatura de Document Object Model y corresponde a una interfaz de programación que nos permite agregar, actualizar y remover elementos de una página web.

Cada vez que modificamos la página web utilizando Javascript estamos manipulando el DOM real. Sin embargo, a diferencia de cuando manipulamos elementos directamente con Javascript, en React se utiliza un DOM virtual, el cual es un objeto que lleva registro de cada cambio a implementar y con base en este registro se realizan los cambios de forma ordenada.

En React todo esto funciona de forma automática sin necesidad de que tengamos que hacer algo en especial.

¿Por qué se utiliza un DOM virtual en lugar de un DOM real?

Porque manipular el DOM real múltiples veces es costoso en términos de recursos, utilizando un DOM virtual es posible implementar los cambios en la página web de forma ordenada, sincronizada y generando un impacto mucho menor que haciendo los cambios directamente.

Actividad 2: Ejemplo de buscador en react

A continuación, realizaremos un ejercicio donde veremos un ejemplo de buscador en React. Para lograrlo, implementaremos las siguientes características:

1. Consumo de API, extrayendo datos de usuarios de jsonplaceholder.
2. Uso del hook useState para guardar los datos de los usuarios y controlar el input de la barra de búsqueda.
3. Uso del hook useEffect, para controlar los efectos de la aplicación y realizar la consulta a la API antes mencionada.

- **Paso 1:** Creamos una aplicación con vite y le asignamos el nombre ejemplo-buscador.

- **Paso 2:** Limpiamos la aplicación y eliminamos los archivos App.css e index.csss. En el componente principal, también ejecutamos la limpieza y dejamos el siguiente código:

```
import Search from "./components/Search";

function App() {
  return (
    <>
      <h1>Buscador</h1>
      <Search />
    </>
  );
}

export default App;
```

- **Paso 3:** Creamos la carpeta components y creamos un componente llamado Search.jsx.

En el Search.jsx es donde definiremos todo el código para realizar la consulta a la API y el conjunto de funciones que realizarán las acciones filtrado.

- **Paso 4:** Ahora, importamos los hooks useState y useEffect y escribimos el código para obtener los datos de la API, para esto utilizaremos fetch.

```
//Estado
  const [users, setUsers] = useState([]);
  const [search, setSearch] = useState("");

  //Consumo de API
  const url = "https://jsonplaceholder.typicode.com/users";

  const getData = async () => {
    const response = await fetch(url);
    const data = await response.json();
    setUsers(data);
  };

  useEffect(() => {
    getData();
  }, []);
```

- **Paso 5:** Seguidamente, escribimos las funciones que ejecutarán el filtro a los datos que se reciban de la API y de este modo hacer funcionar la barra de búsqueda.

```
//Búsqueda de datos
  const handleSearch = (e) => {
    setSearch(e.target.value);
    // console.log(e.target.value);
  };

  //Filtrado de datos
  let results = [];
  if (!search) {
    results = users;
  } else {
    results = users.filter((user) =>
      user.name.toLowerCase().includes(search.toLowerCase())
    );
  }
```

Entendamos el código

1. La utilidad de tener la función handleSearch, es que podamos conectarla al input del formulario para capturar los datos ingresados mediante texto.
2. Seguidamente, creamos una variable let results como arreglo vacío, en la cual aplicamos una validación:
    a. Sí, el search es distinto a un string vacío, entonces el arreglo de results será igual la variable de estado donde se encuentran los usuarios (users).
    b. En caso contrario, a results le asignamos la variable de estado users, en la cual aplicamos un filtro.
    c. Este filtro convierte los valores en minúscula con el método .toLowerCase()
    d. Al cumplirse esta condición, el filtro retornará los datos que hayan sido ingresados mediante el input.

- **Paso 6:** Ahora, escribimos el código para el render del componente Search.jsx

```
//Render a la vista
  return (
    <div>
      <input
        type="text"
        placeholder="search"
        className="form-control"
        value={search}
        onChange={handleSearch}
      />

      <table className="table table-striped table-hover my-4 shadow-lg">
        <thead>
          <tr>
            <th>Name</th>
            <th>User name</th>
          </tr>
        </thead>

        <tbody>
          {results.map((user) => (
            <tr key={user.id}>
              <td>{user.name}</td>
              <td>{user.username}</td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
```

Entendamos el código

1. En primera instancia, tenemos un input que será la barra de búsqueda. Este input está conectado la función handleSearch, en la cual capturamos lo ingresado mediante texto. Además, al value del input le asignamos el valor de la variable de estado search. Esta, en su estado inicial, se encuentra como un string vacío.
2. Seguidamente, pintamos una tabla haciendo uso de clases de bootstrap.
3. Esta tabla muestra en la cabecera el nombre del usuario y username.
4. Luego, el cuerpo de la tabla, es rellenado de manera dinámica haciendo un recorrido por el arreglo de results, en este caso, se muestra el nombre y el username de casa usuario.

Si ejecutas la aplicación hasta este punto, verás que el filtro de búsqueda se ejecuta correctamente. En la plataforma tendrás el código terminado con el nombre ***"Repositorio Ejercicio - Ejemplo buscador React"***

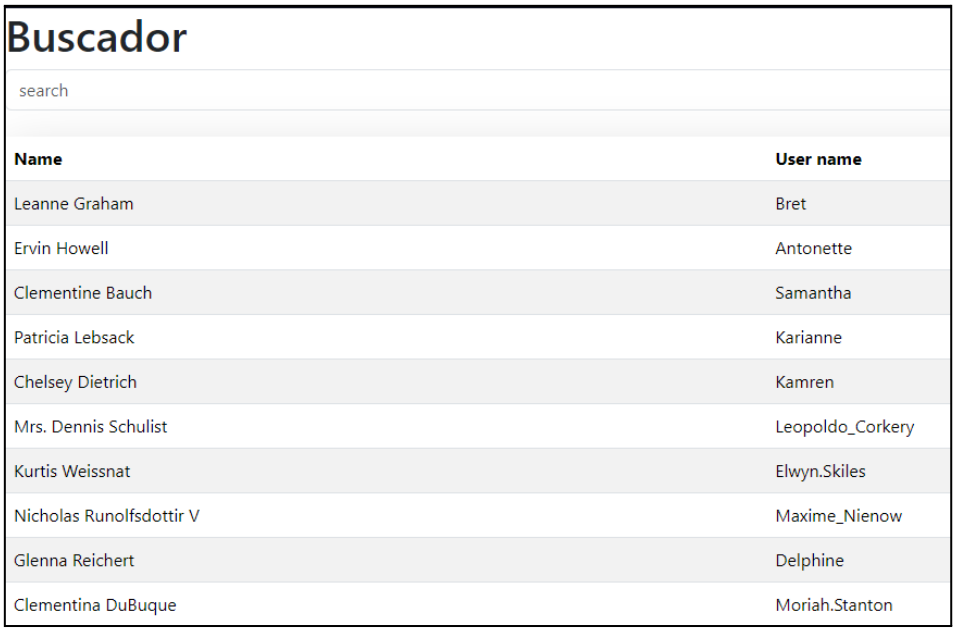

Imagen 5. Datos sin filtrar
Fuente: Desafío Latam

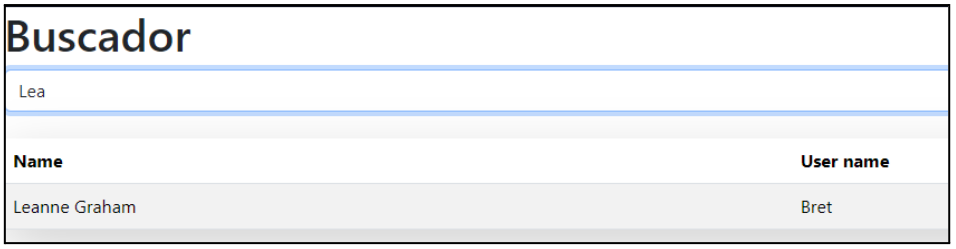

Imagen 6. Datos filtrados
Fuente: Desafío Latam

Próximos pasos

Durante este curso hemos trabajado con componentes que manejan el estado de manera local, o sea, dentro del mismo componente. Cuando las aplicaciones van creciendo, es muy posible que distintos componentes intervengan sobre el mismo estado, y esto, agrega complejidades a resolver que son muy interesantes.

Preguntas de React para una entrevista laboral

A modo de actividad revisa que seas capaz de contestar todas estas preguntas

Unidad 1 - Introducción a React

1. ¿Cuál es la diferencia entre un componente de clase y un componente funcional?
2. ¿Pueden los exploradores leer JSX?
3. ¿Qué atributo tenemos que utilizar para agregar una clase en una componente de React?
4. ¿Cuál es el problema con el siguiente código dentro de una componente?

```
return(
      <li> 1 </li>
      <li> 2 </li>
      <li> 3 </li>
)
```

5. ¿Cómo podemos pasar información entre un componente padre y un componente hijo?
6. ¿Qué tipo de expresión es la siguiente? ¿Cómo se lee?

```
condición ? expr1 : expr2
```

Unidad 2 - Estados de los componentes y eventos

1. ¿Para qué sirven los props?
2. ¿Cuál es la diferencia entre un estado y un props?
3. ¿Para qué sirve la función setter entregada por el hook useState?

Unidad 3 - Renderización dinámica de componentes

1. ¿Qué problema podemos encontrar con los objetos mutables, como los arreglos, cuando cambiamos estados?

2. ¿Qué es un efecto secundario?

3. ¿Qué usos tiene el hook useState?

4. ¿Qué función cumplen las keys (o claves) en React?

Unidad 4 - Consumo de APIs con React

1. ¿Cuáles son las etapas (o fases) del ciclo de vida de un componente?

2. ¿Qué sucede si a useEffect le pasamos un arreglo vacío?

3. ¿Por qué en modo de desarrollo, en la fase de montaje, el código de una aplicación en React se ejecuta 2 veces?

4. ¿Qué es lo que permite realizar el hook useRef?

5. ¿Qué diferencias hay entre montaje y renderizado de un componente?

6. ¿Cuál es la diferencia entre el DOM virtual y el DOM real?