DigitalHouse>

Certified Tech Developer
The Ultimate Degree

Front End III

Glosario de repaso

1) Hook: "Un Hook es una función especial que permite 'conectarse' a características de React." Los Hooks son básicamente funciones. Permiten a componentes funcionales incorporar características de React, previamente reservadas solo a componentes de clase. Entre otras utilidades, dotan a los componentes funcionales de estado interno y ciclo de vida. Además —y no menos relevante— resuelven problemas que React ha enfrentado por años. Los Hooks son un modo simplificado de insertar características de React —como por ejemplo: estado, ciclo de vida, contexto, referencias, en componentes funcionales—; sin alterar el funcionamiento y las bases de React como son la importancia de los componentes, el flujo de datos y las props. Además, permiten una mejor y más simple reutilización, composición y testeo del código. Lo brillante de Hooks es que nos permiten extraer lógica de un componente para ser reutilizada y compartida.

2) Problemas de los componentes de clase:

- La lógica de estado no es fácil de reutilizar.
- Los componentes complejos son difíciles de comprender.
- La implementación de clases puede ser confusa y molesta.

3) Surgimiento de Hooks: en el mes de octubre del año 2018 —en la React Conf celebrada en Henderson, Nevada, EEUU—, Sophie Alpert y Dan Abramov, miembros del equipo de Facebook encargado de desarrollar React (por ese entonces), presentaron una nueva característica revolucionaria para el mundo front end: React Hooks. Más tarde, en febrero de 2019, esta propuesta se hizo realidad y React liberó su versión 16.8 incluyendo la API de Hooks.

4) Beneficios hooks:

- Menor cantidad de código.
- Código más organizado.
- Utilización de funciones reutilizables.
- Mayor facilidad para testear.
- No llama a super() en un constructor de clases.
- No trabaja con this y bind en el uso de manejadores.
- Simplifica la vida y el ciclo de vida.
- El estado local está dentro del alcance de los handlers y las funciones de efectos secundarios.
- Componentes más reducidos que facilitan el trabajo de React.

5) Convivencia entre componentes de Clase y Hooks: en React conviven ambos tipos de programación. Es decir, que en una misma aplicación pueden convivir componentes de ambos tipos, para dar tiempo a un cambio de mentalidad gradual. Como ejemplo de esta decisión, Facebook conserva una enorme cantidad de código escrito en clases. Finalmente, debemos aclarar que no existe la libertad absoluta entre ambos tipos de componentes, ya que no podemos usar Hooks dentro de un componente de clase.

6) Hooks nativos

useState	Declara variables de estado y las actualiza.
useEffect	Permite incluir y manejar efectos secundarios.
useContext	Permite crear un proveedor de datos globales para que puedan ser consumidos por los componentes envueltos por el proveedor.
useReducer	Es una variación de useState, es conveniente utilizarlo cuando nos encontramos ante un gran número de piezas de estado.
useCallback	Devuelve un callback que es memorizado para evitar renderizados innecesarios.
useMemo	Devuelve un valor memorizado. También es útil para evitar renderizados innecesarios. Optimiza el rendimiento.
useRef	Permite utilizar programación imperativa. Devuelve un

	objeto mutable cuyos cambios de valor no volverán a renderizar el componente. Por el contrario, se mantendrá persistente durante la vida completa del componente.
useImperativeHandle	Personaliza el valor de instancia que se expone a los componentes padres cuando se usa ref.
useLayoutEffect	Es similar a useEffect, pero se dispara de forma síncrona después de todas las mutaciones de DOM.
useDebugValue	Permite aplicar una etiqueta a los Hooks personalizados visible mediante React DevTools.

7) useState: desde el lanzamiento de Hooks, los componentes funcionales pueden declarar y actualizar su estado interno mediante el hook useState. Este es una función que crea una variable que permite almacenar una pieza de estado interno y, a la vez, actualizar dicho valor en un componente funcional.

8) Estructura de useState: este recibe un argumento y devuelve un array con 2 elementos. El primero será el valor de la variable y el segundo la función para modificarla.

9) Funcionamiento de useState: cuando React renderiza o rerenderiza nuestro componente, este solicita a React el último valor de estado con el fin de dibujar la interfaz de usuario y también la función para actualizar el valor de estado. Si se ejecuta la función para modificar el valor, React rerenderizará la interfaz. React comparará la interfaz de usuario generada con la versión almacenada y actualizará la pantalla del modo más eficiente.

10) Ciclo useState:

1. React atraviesa el árbol de componentes montándolos para dibujar la UI. En cada caso, React pasará las props correspondientes.
2. Cada componente invoca useState por primera vez, solicitando el valor inicial a useState que a su vez se lo requiere a React.
3. React retorna tanto el valor actual como la función de actualización.
4. El componente configura los controladores de eventos de usuario.
5. El componente utiliza el valor de estado obtenido para declarar la UI.
6. React actualiza el DOM con las modificaciones requeridas.

7. Cuando el controlador de eventos invoca a la función de actualización. Se activa un evento y se ejecuta el controlador. A través del handler, se invoca a la función de actualización para modificar el valor del estado.
8. React modifica el valor del estado con el valor pasado por la función de actualización.
9. React llama al componente.
10. El componente vuelve a llamar a useState, pero sin tener en cuenta el argumento pasado inicialmente.
11. React vuelve a retornar el valor de estado actual y la función de actualización.
12. El componente crea una nueva versión del manejador de eventos y utiliza el nuevo valor de estado.
13. El componente devuelve su IU con el nuevo valor de estado.
14. React actualiza el DOM con las modificaciones requeridas.

11) useEffect: es el Hook que se encarga de configurar y gestionar los efectos secundarios en el ciclo de vida de un componente funcional. Los hace predecibles, controlables. Es el equivalente, sintetizado en una sola función, de lo que podemos hacer con los métodos componentDidMount, componentDidUpdate y componentWillUnmount en un componente de clase.

12) Características de useEffect:

- Tiene acceso a las variables dentro del componente puesto que comparten el mismo ámbito.
- Se ejecuta después del primer render y también después de cada render posterior, a no ser que se lo configure en sentido contrario.
- Nos permite pensar de otro modo al ciclo de vida, en términos de sincronización de piezas lógicas.

13) Estructura de useEffect: la función tiene 2 argumentos: effect y deps. El primer argumento es una función callback. Es en donde el efecto secundario sucede, donde el código debe ser escrito. El segundo argumento nos brinda las opciones de sincronización.

14) Opciones de sincronización de useEffect:

- Se ejecuta en cada render.

- Solo se ejecuta tras el primer render.
- Corre en el primer render y luego solo si varía la variable incluida como dependencia dentro del segundo argumento.

15) Casos de uso de useEffect:

- Establecer el título de una página de forma imperativa.
- Trabajar con temporizadores: setInterval o setTimeout.
- Leer ancho, alto o posición de elementos del DOM.
- Registrar mensajes.
- Establecer u obtener valores de almacenamiento local.
- Realizar peticiones a servicios.

16) Función de limpieza de useEffect: cuando implementamos componentes que incluyen efectos secundarios es posible —en algunos casos— que el usuario al navegar por la aplicación desmonte componentes antes que tales efectos secundarios finalicen. Ya sea, por ejemplo, el retorno de datos de una petición a una API externa o una función temporizadora. Esto producirá errores debido a que la aplicación buscará actualizar valores de estado de un componente que se ha eliminado. useEffect incluye un mecanismo de limpieza, el cual es una función que se ejecutará cuando se desmonte el componente. Siempre que el componente se vuelva a renderizar, React llamará a la función de limpieza antes de ejecutar useEffect si el efecto se ejecuta de nuevo. Si hay varios efectos que necesitan ejecutarse de nuevo, React llamará a todas las funciones de limpieza para esos efectos. Una vez finalizada la limpieza, React volverá a ejecutar las funciones de efectos según sea necesario.