Módulo 4 – Hooks

# ¿Qué son?

Slide 2

ReactJS es la librería más usada para el desarrollo de aplicaciones web. Y esto lo ha logrado con la adopción de patrones, tecnologías claves para que el desarrollo sea lo más eficiente y versátil. Alguno de esos conceptos son los Hooks.

Previamente ReactJS se manejaba solo con clases. Cada clase tenía dos principales propiedades “props” y “state”. Estas permiten recibir valores desde un componente padre y manejar el estado de valores ene l componente respectivamente.

Sin embargo, como hemos visto, con los componentes funciones no podemos mantener el estado de variables declarándolas con “let” ya que estas se reinicializan en cada render con el valor que escribimos en código. Es por esto que nacieron los Hooks, su traducción es gancho, y es porque permiten “enganchar” la funcionalidad del estado, y como lo veremos más adelante, administrar el Ciclo de vida del componente en Componentes funcionales y muchas más funcionalidades y características que podemos encapsular en Hooks. También podemos crear los nuestros para ahorrarnos cierto trabajo y reutilizarlo en diferentes lugares.

# ¿Cómo se realizaba antes?

Slide 3 - 4

En los componentes Clase la administración del ciclo de vida y el manejo de estado se obtenía por herencia de la clase Component de ReactJS. Estas nos daban funciones tales como componentDidUpdate, componentWillUnmount, setState y demás que permitían la modificación del estado y administración del ciclo de vida del componente.

Visualizar comparativa con las capturas del código

# Hooks principales

Visitar la página  
<https://react-hooks-cheatsheet.com>

* useState, administración de estados, pueden ser primitivos, JSON, clases, etc.
* useEffect, admintración del life cycle del componente, lo veremos más adelante del curso. Dato, este siempre se ejecuta después de que el componente de haya renderizado por algún cambio.
* useContext, usado para obtener el estado global cuando usamos React Context como Global Store
* useLayoutEffect, se invoca muy similar que el useEffect, con la diferencia de que se invoca antes del render. Puede afectar al rendimiento según su implementación.
* useReducer, usado principalmente como alternativa del useState, para simplificar la invocación al querer actualizar el estado.
* useCallback, usado principalmente para evitar que funciones se redefinan en cada render. Limitando su re-definicion según una lista de dependencias. Así logramos que por ejemplo componentes que reciben estas funciones o valores se vuelvan a renderizar a pesar de que los valores que usan no hayan cambiado. Este se usa en funciones no en Componentes.
* useMemo, muy similar a useCallback, pero esta enfocado a memorizar VALORES, estos también pueden ser Componentes. Por lo que se puede usar para que un componente solo sea re-renderizado según la lista de dependencias que se declare.
* useRef, usado para almacenar valores en toda la vida de un componente, puede parecerse mucho a useState, pero este no retorna una función para actualizar el estado. Su uso principal es para guardar referencias de Componentes para así invicar funciones de ellos de manera programáticas.

# Usando Hooks

Crear un nuevo URL “users/form/:id?” para crear un formulario donde vamos a usar los Hooks básicos de React. Crearemos el formulario con los cambios

* Name, String.
* Age, Number.
* City, String.
* Verified, boolean.

Usaremos los Hooks:

* useState, para adminitrar el type “Form” que tendrá los datos del formulario.
* useLayout, para precargar los datos cuando se tiene un ID en el URL.
* useMemo, para hacer que un componente custom cambie solo cuando verified cambia.
* useRef, para hacer un focus manual a un input al entrar
* useEffect, para invocar la función de focus del valor de hook ref solo cuando la página se monta por primera vez.

Cambios:

* use “$” para facilitar el uso de condicionales en tailwind y explicar porque un usar template en clases Tailwind.
* usar Zod para definir las clases y colocar todos los types en “types” incluyendo los Loaders.
* Crear los componentes de los inputs: Check, Input, Number, Verified.
* Crear la nueva vista “form” dentro de “pages/users” y usar los componentes. Usar los hooks necesarios dejar el guardado para el final.
* Crear una función en “api” para guardar usuario.
* Implementar DataSourse Repository. Explicar sus beneficios
  + Crear carpeta “domain” para las clases abstracttas, impl para implementacion.
  + En domain solo crear la carpeta “ds” para colocar el archivo “DataDS” que se implementará por LocalStorage
  + En impl crear “ds” donde estará la implementación de LocalStorage, en “repo” crear la clase que usara el DS recibido por el constructor.
  + Crear un archivo para exportar el DataSourceImpl.
  + Conectar todo con el nuevo datasoruce en los loaders.

# Práctica

Crear un botón en el componente “UserProfile” que se llame “Eliminar” de color rojo. Este deberá invocar una nueva función en nuestro DataSource “deleteUser” que recibe el index del usuario y lo elimina de la base de datos.

En la siguiente clase demostrar lo fácil que es intercambiar Datasource con este patron usando Firebase. Responder dudas de los módulos.

# Extras

¿Existen alternativas al patrón para cargar datos de React-router? Definitivamente sí. Es probable que muchas veces el enfoque de un Loader o fetch de recursos por página no sea muy flexible o se ajuste a las necesidades de nuestro proyecto. A veces necesitamos cargar recursos en base a la selección del usuario o una lógica más completa para realizar POSTs en formularios.

Para esos casos podemos usar una librería ampliamente usada que es “React-query”. Esta nos permite realizar operaciones de query (obtener datos) al montar un componente con el uso del hook “useQuery” y para realizar mutaciones o enviar de manera programática o manual podemos usar “useMutation”.

La ventaja de esta librería es la flexibidad de realiza queries y mutaciones de manera ordenaba y con un sistema de CACHING out of the box, es decir, ya configurado. Lo que puede acelerar la velocidad y flujos de nuestra apliación.