Prueba Práctica - Jhoan Daniel Rojas.

Sección 1: Teoría.

1. Primero que todo definamos que tanto **Context API** y **Redux** son soluciones o herramientas para el manejo y gestión de estados en **React.js,** Una de las principales diferencias es que **Redux** está basado en un **patrón de arquitectura llamado flux** lo cual le da una mayor versatilidad para el manejo de estados ya que centraliza toda la información del proyecto en un único **"Store"** el cual es el encargado de almacenar la información de los estados para posteriormente ser consultados por los componentes, mientras que **Context API** es una funcionalidad nativa de React que permite compartir estado entre componentes lo cual no tiene tanta escalabilidad gracias que no existe esa centralización de la información y se recomienda para información simple.

   Ahora bien, los casos de usos depende tanto del objetivo y tamaño del proyecto ya que **Context API** al ser nativo y poco escalable es más recomendable para proyectos pequeños o simples donde no hay muchos estados o lógica es un poco más simple, por el contrario **Redux** al ofrecer una mejor escalabilidad y un mejor manejo de diferentes estados gracias al **patrón Flux** es recomendable para usarse en proyectos grandes o complejos donde el proyecto requiera el manejo de muchos estados complejos, también es recomendable usar **Redux** si se desea trabajar con acciones asíncronas.

2. El ciclo de vida de un componente se puede definir en tres fases: **montaje**, **actualización** y **desmontaje (Mounting, Updating, Unmounting).**

   a. **Mounting** se refiere cuando el componente es renderizado por primera vez en el DOM.

   b. **Updating** se refiere cuando el componente tiene cambio de estado o de prop lo cual hace que se actualice y renderice nuevamente

   c. **Unmounting** se refiere a cuando el componente es eliminado del DOM.

   Los **efectos secundarios** son cualquier operación que interactúa con el mundo externo al componente que no dependen de las actualizaciones directas de estados del componente, como llamados a api, manipulación directa del DOM o suscribirse a eventos o configuraciones externas. Para manipular o controlar estos efectos de manera eficiente se usa **useEffect()** de forma que puedas usar estos cambios a favor

del ciclo vida del componente ejecutandolos cuando deseas, es decir poder realizar estas acciones en el montaje, en una actualización o en un desmontaje.

3. La reconciliación en react es el proceso en el cual se actualiza la interfaz de usuario o UI de manera eficiente para reflejar cambios de estado o props de los componentes. Este proceso funciona por medio del **“algoritmo de diferencias"** (**diffing algorithm**) el cual busca identificar y realizar únicamente los cambios necesario sin necesidad de renderizar nuevamente el **DOM** completamente, para lograrlo este algoritmo usa el **virtual DOM** el cual es una copia ligera del **DOM** actual el cual se usa para comparar los nodos e identificar únicamente los cambios necesarios sin necesidad de renderizar todos los componentes nuevamente.

Sección 2: Práctica.

1. **Componente Personalizado:** Link Repositorio Github.
2. **Gestión Compleja de Estado:** Link Repositorio Github.
3. **Hooks Personalizados:** Link Repositorio Github.

Sección 3: Resolución de Problemas.

1. **Optimización de Rendering**
   Para estos casos donde se busca mejorar la renderización de muchos elementos siempre es recomendable utilizar una estrategia o patrón que optimice la forma en la cual se muestran los componentes e información al usuario ya que muchas veces no es necesario mostrar o cargar toda información directamente. Por ello se recomiendan estrategias como el **scroll infinito,** el cual es una técnica o patrón de diseño que busca renderizar nuevo contenido únicamente cuando el usuario ya consultó o navegó el contenido actualmente renderizado o la **paginación** la cual es una estrategia que busca distribuir el contenido en partes más cortas por medio de páginas que permiten renderizar únicamente el contenido exclusivo de esa página. Además también se pueden utilizar otro tipo de estrategias o patrones que ayuden a optimizar parte de la carga de los datos como la **Carga Diferida o Lazy Loading** qué busca retrasar la carga de los datos más pesados de manera progresiva mejorando la velocidad de carga y la experiencia del usuario, ya que el usuario puede seguir navegando y consumiendo información el contenido renderizado actual mientras de fondo se van cargando los siguientes elementos o datos más pesados de la renderización.

2. **Diagnóstico de Errores.**
   El problema que pude encontrar es la forma de usar el **setCount()** dentro del intervalo ya que al estar dentro del contexto del **setInterval()** no se está actualizando correctamente el valor porque **count** dentro del contexto sigue siendo fijo el valor inicial, por lo cual se debe usar una forma que siempre use el valor de más reciente de count mediante el cambio en esta línea:

   setCount((prevCount) => prevCount + 1);

   Link del código corregido

Sección 4: Proyecto Corto.

Link del repositorio de Github.