PRUEBA TÉCNICA REACT – RESPUESTAS

Nombre: Santiago Rendón Múnera
Cédula: 1001578278
Fecha: 4/12/2025

1.1. Preguntas de Elección Múltiple

Pregunta 1: ¿Qué es JSX en React?
Respuesta: A

Pregunta 2: ¿Cuál es el props en React?
Respuesta: B

Pregunta 3: ¿Cómo se define un componente funcional en React?
Respuesta: A y C

Sección 2: Estado y Ciclo de Vida

Pregunta 1: Explica la diferencia entre state y props en React. ¿Cuándo deberías usar cada uno?
Respuesta:

El state en React representa valores internos que puede tener un componente y que pueden cambiar con el tiempo. Cuando esos valores cambian, actualizan la UI. Además, el state se suele usar para controlar estados relacionados con el fetching de datos, como loading, error o la data que se recibe.

Las props son las propiedades que recibe un componente desde su componente padre. Esas props son inmutables, es decir, el hijo no las modifica directamente. Si es necesario cambiar alguno de esos valores, ese cambio se debe hacer desde el componente padre, que actualiza su propio state y vuelve a pasar las props actualizadas al hijo.

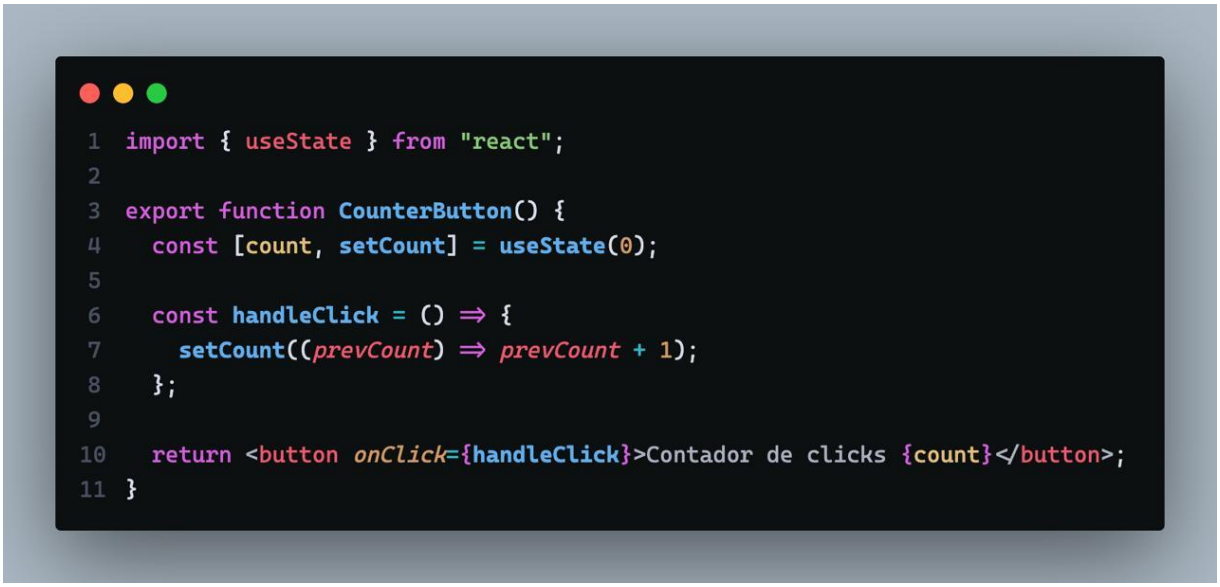
El state se usa para datos que el propio componente maneja y que cambian con el tiempo, y las props se usan para pasar información o funciones desde un componente

padre a un componente hijo, por ejemplo los handlers para manejar eventos y actualizar los estados del componente padre.

Pregunta 2: Describe cómo manejarías un evento de clic en un componente funcional de React.
Respuesta:

Lo manejaría definiendo un handler que se encargue de manejar el evento de click del elemento que dispara el evento.

Ese handler lo definiría dentro del componente o custom hook y luego se pasaría al elemento que a emitir el evento mediante la prop onClick. Dentro de ese handler podría ejecutar la lógica que necesite, como actualizar el state, llamar una función, o hacer alguna otra acción cuando el usuario haga click.



```
1  import { useState } from "react";
2
3  export function CounterButton() {
4    const [count, setCount] = useState(0);
5
6    const handleClick = () => {
7      setCount((prevCount) => prevCount + 1);
8    };
9
10   return <button onClick={handleClick}>Contador de clicks {count}</button>;
11 }
```

Pregunta 3: ¿Qué es el Hook useEffect en React? Proporciona un ejemplo de cómo se usa para realizar una petición de datos a una API.
Respuesta:

El Hook useEffect en React lo uso cuando necesito ejecutar lógica con efectos secundarios después de que el componente se ha renderizado. Sirve para sincronizar el componente con sistemas externos, como una API, el DOM del navegador o algún tipo de suscripción a eventos o timers.

En el caso de una petición a una API, lo que hago normalmente es utilizar la API de fetch dentro de un useEffect y usar un array de dependencias vacío para que esa petición se ejecute solo una vez cuando el componente se monta. Con la respuesta de la API actualizo el state y así React vuelve a renderizar la UI con los datos.

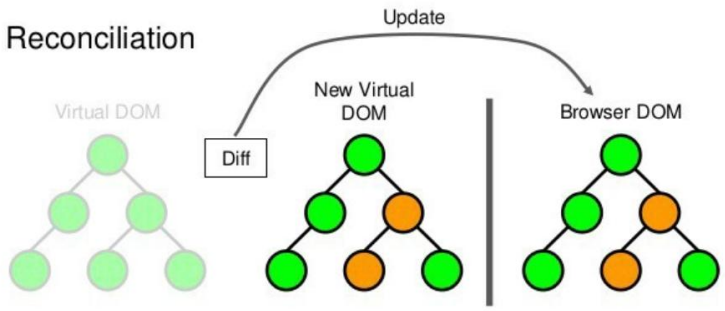
```tsx
export function UsersList() {
  const [users, setUsers] = useState<User[]>([]);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    async function fetchUsers() {
      try {
        setIsLoading(true);
        setError(null);

        const response = await fetch("https://api.example.com/users");

        if (!response.ok) {
          throw new Error("Error al obtener los usuarios");
        }

        const data = await response.json();
        setUsers(data);
      } catch (error) {
        console.error("Error al cargar usuarios", error);
        setError("Ocurrió un error al cargar los usuarios");
      } finally {
        setIsLoading(false);
      }
    }

    fetchUsers();
  }, []);

  if (isLoading) {
    return <p>Cargando usuarios ... </p>;
  }

  if (error) {
    return <p>{error}</p>;
  }

  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

Sección 3: Optimización y Buenas Prácticas (Senior)

Pregunta 1: Explica qué es la "reconciliación" en React y cómo React optimiza el proceso de actualización del DOM virtual.
Respuesta:

La reconciliación es el proceso interno que usa React para comparar el Virtual DOM anterior con el nuevo Virtual DOM, este se genera después de un cambio en el state o las props. A partir de esa comparación, React evalúa qué cambios mínimos tiene que aplicar para que la UI quede actualizada.

En lugar de actualizar todo el DOM real cada vez, React solo aplica al DOM real las diferencias que encuentra en el proceso de reconciliación, pasando a la fase de commit actualizando en el DOM real solo lo necesario. Eso hace que las actualizaciones sean más eficientes, porque se cambian solo los nodos necesarios y no toda la página.



Pregunta 2: ¿Cuáles son las mejores prácticas para manejar estados complejos en aplicaciones React grandes? Menciona algunas bibliotecas que se pueden usar.
Respuesta:

Para manejar estados complejos en aplicaciones grandes con React, primero intento mantener el estado lo más local posible. Si un estado solo lo necesita un componente o una parte pequeña de la interfaz, se puede manejar con hooks como useState o useReducer.

Algo importante es que, aunque existe useContext, yo no lo usaría como la solución principal para manejar estados complejos. Lo veo más para datos que casi no cambian,

como el tema de la aplicación, el idioma, o información global de usuario. Cuando el estado cambia muy seguido y se mete todo dentro de un Context, cada cambio puede hacer que se vuelvan a renderizar muchos componentes que consumen ese contexto y el código se vuelve más difícil de mantener.

Para estados globales más complejos prefiero usar librerías pensadas para eso, como Redux o Zustand, que permiten centralizar parte del estado de la aplicación y tener una forma más organizada de actualizarlo.

Y para el estado relacionado con peticiones a una API, normalmente utilizo TanStack Query, que se encarga de la parte de fetching, sincronización con el servidor e invalidations, en lugar de intentar manejar todo eso manualmente en el estado global.

Pregunta 3: Considera que estás trabajando en una aplicación React que tiene problemas de rendimiento. ¿Qué herramientas y técnicas utilizarías para identificar y solucionar los problemas de rendimiento?
Respuesta:

Lo primero que haría es medir antes de tocar el código. Usaría las herramientas de desarrollo del navegador para revisar la pestaña de Performance y ver qué tan rápido se renderiza la página y si hay algo que esté bloqueando, y la pestaña de Network para ver si hay peticiones muy lentas o muchas peticiones seguidas. También usaría React DevTools, sobre todo el Profiler, para ver qué componentes se están renderizando muchas veces o cuáles tardan más de la cuenta.

Con esa información empezaría a ajustar cosas. Por ejemplo, revisaría si tengo estados muy arriba en el árbol de componentes que están causando renders en muchos componentes cuando en realidad solo debería actualizarse una parte pequeña. En esos casos intento mover el estado más cerca del componente que lo necesita o sacar la lógica a custom hooks.

Y ya cuando veo que un componente se está renderizando muchas veces sin que haga falta, ahí sí reviso si vale la pena usar memo, useMemo o useCallback, pero solo en los casos donde de verdad haya un problema de rendimiento que haya visto en las herramientas.

Además, hoy en día React ya ayuda bastante con el nuevo compilador, que aplica varias optimizaciones de forma automática y reduce muchos renders innecesarios, pero aun

así es importante identificar la causa raíz el problema y conocer estas técnicas de optimización.

Sección 4: Avanzado – Preguntas Abiertas

Pregunta 1: Explica cómo implementarías la carga de componentes de forma diferida (lazy loading) en una aplicación React. Proporciona un ejemplo de código.
Respuesta:

Usaría lazy junto con Suspense. La idea es que, en lugar de importar el componente de forma normal, lo defino como un componente que se carga de forma diferida, y luego lo envuelvo en un Suspense para mostrar un fallback mientras se carga.

```jsx
import { useState, lazy, Suspense } from "react";

const UserDetails = lazy(() => import("./UserDetails"));

export function ProfilePage() {
  const [showDetails, setShowDetails] = useState(false);

  const handleClick = () => {
    setShowDetails(true);
  };

  return (
    <div>
      <h1>Perfil de usuario</h1>

      <button onClick={handleClick}>Ver detalles</button>

      {showDetails && (
        <Suspense fallback={<p>Cargando detalles ... </p>}>
          <UserDetails />
        </Suspense>
      )}
    </div>
  );
}
```

De esta forma, el componente UserDetails no se carga en el bundle inicial, sino solo cuando el usuario lo necesita, y mientras tanto React muestra el fallback que le paso al Suspense.

Pregunta 2: Describe cómo manejarías la autenticación y autorización en una aplicación React. ¿Qué herramientas o bibliotecas usarías y por qué?
Respuesta:

Para la autenticación, tendría un formulario de login en React que envía el usuario y la contraseña a una API. Si las credenciales son correctas, el backend devuelve un token, por ejemplo, un JWT, y la información básica del usuario.

En el frontend guardo ese estado de sesión en un sitio centralizado, por ejemplo, en un store con Zustand o en un contexto con un custom hook tipo useAuth, donde expongo valores como user, isAuthenticated, login y logout.

Para las peticiones a la API usaría una capa de servicios, donde se añada el token en el header Authorization y, si el backend responde con un 401, desde ahí mismo puedo hacer logout o redirigir al usuario a la pantalla de login.

Para la autorización, usaría el router para proteger rutas. Por ejemplo, con React Router crearía un componente tipo ProtectedRoute o RequireAuth que revise si isAuthenticated es true. Si no lo es, redirige a login, si sí lo es, deja pasar al usuario.

Adicionalmente, con la información de roles o permisos que viene del backend puedo crear algo como RequireRole para que ciertas rutas o secciones solo sean accesibles para, por ejemplo, administradores. Dentro de los propios componentes también se pueden ocultar botones o secciones según el rol del usuario.

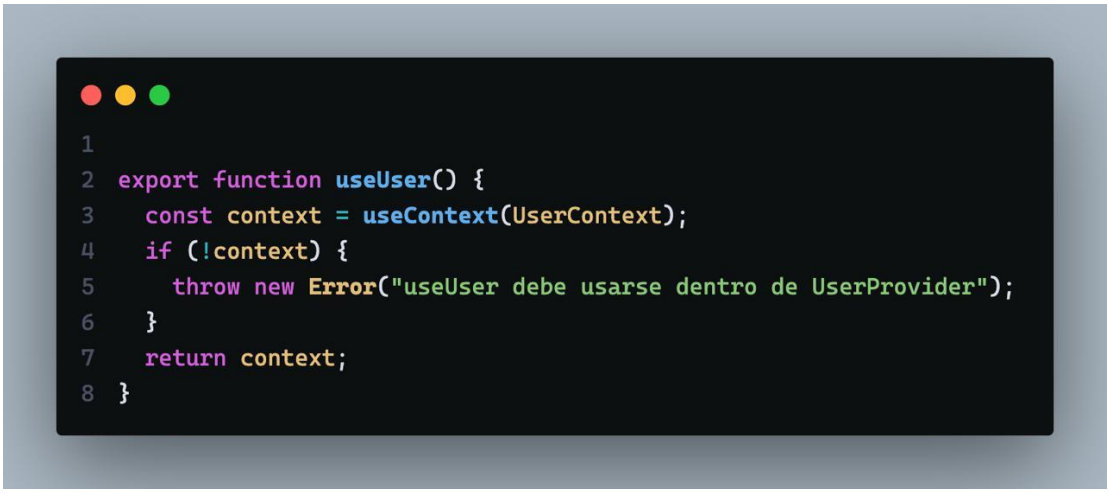
Pregunta 3: ¿Cómo implementarías un contexto global en React para manejar datos compartidos entre múltiples componentes? Proporciona un ejemplo de código usando el Hook useContext.
Respuesta:

Un contexto global en React lo usaría cuando tengo datos que se comparten entre varios componentes y no quiero estar pasando props por muchas capas. Esto ayuda a evitar el problema del prop drilling.

La idea es crear un contexto con createContext, un Provider que envuelva la parte de la aplicación que necesita esos datos, y luego en los componentes hijos usar useContext para leer ese valor.

```tsx
import { createContext, useContext, useState } from "react";

type User = {
  name: string;
  email: string;
};

type UserContextValue = {
  user: User | null;
  login: (newUser: User) => void;
  logout: () => void;
};

const UserContext = createContext<UserContextValue | null>(null);

export function UserProvider({ children }: { children: React.ReactNode }) {
  const [user, setUser] = useState<User | null>(null);

  const login = (newUser: User) => setUser(newUser);
  const logout = () => setUser(null);

  return (
    <UserContext.Provider value={{ user, login, logout }}>
      {children}
    </UserContext.Provider>
  );
}
```

Luego, crearía un custom hook que consulte el contexto y lanza una excepción si se usa fuera del provider.



```tsx

export function useUser() {
  const context = useContext(UserContext);
  if (!context) {
    throw new Error("useUser debe usarse dentro de UserProvider");
  }
  return context;
}
```

En main.tsx envuelvo la aplicación o los componentes que necesiten consultar el valor, con el UserProvider para que el contexto esté disponible en sus hijos.

```tsx
import { createRoot } from "react-dom/client";
import { App } from "./App";
import { UserProvider } from "./UserContext";
import { StrictMode } from "react";

createRoot(document.getElementById("root")!).render(
  <StrictMode>
    <UserProvider>
      <App />
    </UserProvider>
  </StrictMode>
);
```

Luego en el componente que requiera el valor, consulto el context a través del custom hook y accedo al valor del usuario para mostrar su información.

```tsx
import { useUser } from "./UserContext";

export function Profile() {
  const { user } = useUser();

  if (!user) {
    return <p>No hay usuario autenticado</p>;
  }

  return <p>Hola, {user.name}</p>;
}
```

Sección 5: Proyecto Práctico

Descripción
Para esta prueba técnica desarrollé una aplicación de gestión de usuarios con React 19 y TypeScript organizada por módulos de funcionalidad, donde todo lo relacionado con usuarios (componentes, hooks, servicios y tipos) vive en su propio módulo. La app consume la API de DummyJSON, pero solo trae los campos que realmente necesito para evitar problemas de performance al realizar el fetching de datos. Además, guarda los filtros en la URL, así que se pueden compartir por enlace y sobreviven a recargas.

Elegí React 19 con el nuevo compilador para aprovechar las optimizaciones pero me estaban dando problemas las versiones con module-federation, entonces opté por trabajar con React 18 y TypeScript para mejorar la experiencia de desarrollo, Vite como empaquetador de aplicaciones, TailwindCSS v4 para los estilos, TanStack Query para manejar todo el estado de datos de las API's (fetching, caché, loading, error), y React Router para la parte de rutas. Para la calidad usé tests unitarios con Vitest y E2E con Playwright, y todo esto corre en GitHub Actions para que los tests se ejecuten automáticamente en cada push y se despliegue en Vercel.

Uno de los aspectos más retadores de la prueba fue que no conocía la librería de module-federation, así que primero tuve que revisar la documentación y entender cómo integrarla correctamente. También tuve que trabajar con temas de compatibilidad entre librerías y ajustar la configuración cuando algo no era compatible. Además, fue especialmente desafiante el manejo de estilos con Tailwind CSS para conseguir que los estilos se compartieran y aplicaran bien entre el proyecto padre y los microfrontends hijos, por lo que tuve que investigar bastante hasta encontrar una solución adecuada.

Era la primera vez que construía microfrontends desde cero y que los desplegaba como proyectos separados en Vercel, así que, aunque fue exigente, terminó siendo un ejercicio muy interesante y de bastante aprendizaje.

Repositorio GitHub:
https://github.com/rendonnm/domina-react-test

Demo:
https://domina-react-test.vercel.app/

Demo microfrontend:
https://domina-react-test-remote.vercel.app/