Guía de estilo de React JS

1. Organización de Archivos y Componentes

1.1. Estructura de Archivos

Mantener una estructura de archivos clara y organizada es esencial para la escalabilidad y el mantenimiento del proyecto. Algunas recomendaciones:

- Agrupa componentes relacionados en carpetas separadas.
- Sigue la convención de nomenclatura PascalCase para los nombres de archivos de componentes (`MiComponente.jsx`).
- Si un componente tiene archivos relacionados como estilos o pruebas,se deben agrupar en la misma carpeta del componente.

1.2. Separación de Responsabilidades

Seguir el principio de responsabilidad única. Cada componente debe hacer una sola cosa, y hacerlo bien. Los componentes que manejan lógica deben estar separados de aquellos que sólo renderizan contenido visual.

- **Componentes de Presentación**: se encargan de la UI y no manejan estado.
- **Componentes Funcionales**: manejan la lógica y el estado, y se encargan de pasar datos a los componentes de presentación.

2. Definición de componentes y exportación

2.1. Componentes Funcionales con Exportación por Defecto

En React, los componentes se pueden definir como funciones que retornan JSX, y una

práctica común es utilizar la exportación por defecto para hacer el componente disponible en otros archivos del proyecto.

Aporta simplicidad en la importación ya que no es necesario recordar el nombre exacto del componente exportado cuando se utiliza la exportación por defecto, solo se debe importar el componente con cualquier nombre deseado.

```
export default function MiComponente() {
  return (
    <div>
      <h1>
      Hola Mundo
      <h1/>
    </div>
  );
}
```

3. Reutilización de Componentes

3.1. Componentes Pequeños y Reutilizables

Crear componentes pequeños que puedan ser reutilizados en múltiples partes de la aplicación. Esto reduce la duplicación de código y hace que sea más fácil hacer cambios.

- Evitar hacer componentes demasiado grandes y con responsabilidades mixtas.
- Pensar en componentes modulares y reutilizables, como `Button`, `Input`, `Card`.
- Cada componente tiene su propia lógica y manejo de estado.

4. Manejo de Servicios

4.1. Separar la Lógica de las API en Servicios

Se recomienda crear un servicio separado para las llamadas a APIs en lugar de manejarlas directamente dentro de los componentes. Esto mejora la legibilidad y la capacidad de reutilización del código.

- Usar axios u otras librerías para gestionar las peticiones HTTP.
- Mantén la lógica de negocio relacionada con la comunicación con APIs separada del componente.

```
import axios from 'axios';

async function getAll() {
  const res = await axios.get('http://localhost:4000/nombres');
  return res.data;
}

export const nombreService = {
  getAll,
};
```

5. Manejo de errores

5.1. Captura de errores en Funciones Asíncronas

Cuando se trabaja con llamadas asincrónicas, es fundamental manejar los errores para mejorar la experiencia del usuario y evitar que la aplicación falle inesperadamente.
Es importante usar un bloque try/catch para manejar cualquier error que pueda ocurrir durante la llamada a servicios. Esto evita errores inesperados que bloqueen la ejecución del código y permite proporcionar mensajes de error útiles al usuario.

```
const fetchData = async () => {
    try {
        const data = await nombreService.getAll();
        setNombres(data);
    } catch (error) {
        console.error("Error al obtener nombres:", error);
    }
};
```

5.2. Manejo de errores en Formularios

Cuando se trabaja con formularios, es importante manejar errores de validación y errores de servidor de manera efectiva. Se deben proporcionar mensajes claros sobre qué salió mal y cómo el usuario puede corregirlo.

```
const handleSubmit = async (event) => {
    event.preventDefault();
    try {
        await formService.submitForm(formData);
        enqueueSnackbar("Formulario enviado con éxito", { variant: "success" });
    } catch (error) {
        if (error.response && error.response.status === 400) {
            // Error de validación
            setFormErrors(error.response.data.errors);
        } else {
            // Otros errores
            enqueueSnackbar("Error al enviar el formulario", { variant: "error" });
        }
    }
};
```

6. Estado y Ciclo de Vida

6.1. Evita el Estado Global No Necesario

Usar estado local en los componentes siempre que sea posible. Solo elevar el estado a un contexto global o a un estado en el nivel superior si múltiples componentes necesitan

acceso a esa información.

- Usar `useState` y `useReducer` para manejar estados locales simples.
- Usar `useContext` o herramientas como Redux para manejar estados globales sólo cuando sea realmente necesario.

6.2. Controla los Efectos con `useEffect`

El gancho `useEffect` debe usarse para manejar efectos secundarios como llamadas a APIs o manipulación del DOM.

- Limitar el uso de `useEffect` solo a lo necesario. Cargar demasiada lógica en `useEffect` puede hacer el código más difícil de entender.

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log('Actualizando cada segundo');
  }, 1000);

  return () => clearInterval(timer); // Limpieza
}, []);
```

7. Uso de Tailwind CSS

7.1. Uso de Tailwind CSS en React

Tailwind CSS es una librería de utilidades que permite aplicar estilos directamente en las clases de los elementos JSX, lo que hace que el desarrollo sea más rápido y los archivos CSS innecesarios para la mayoría de los casos. En lugar de escribir CSS tradicional, defines los estilos usando clases predefinidas.

En lugar de crear hojas de estilos separadas, aplicas clases de Tailwind directamente en el JSX de los componentes. Esto hace que los estilos sean más rápidos de aplicar y fáciles de modificar.

```
const Button = ({ label }) => {
  return (
    <button className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
      {label}
    </button>
  );
};
```

7.2. Tailwind CSS y Responsividad

Tailwind hace que el diseño responsivo sea sencillo al utilizar prefijos específicos para cada tamaño de pantalla.

```
  const ResponsiveButton = ({ label }) => {
    return (
      <button className="bg-blue-500 py-2 px-4 rounded sm:bg-green-500 md:bg-red-500">
        {label}
      </button>
    );
  };
```

8. Escritura de Código Limpio

8.1. Nombramiento Claro

Elegir nombres claros y significativos para los componentes, funciones y variables. Esto facilita la lectura y comprensión del código por parte de otros desarrolladores.

8.2. Comentarios y Documentación

Escribir comentarios donde sea necesario para explicar la lógica compleja, pero evitar comentar código obvio. La documentación del proyecto debe estar actualizada para facilitar

su mantenimiento y ampliación.

8.3. Indentación

Mantén una indentación consistente para mejorar la legibilidad del código.

Es importante que el código dentro de funciones, bloques de control y componentes esté correctamente indentado para reflejar la estructura jerárquica del código. Por ejemplo:

```
const MyComponent = () => {
    return (
        <div>
            <header>
                <h1>Bienvenido</h1>
            </header>
            <main>
                <p>Este es un componente de ejemplo.</p>
            </main>
        </div>
    );
};
```

8.2. Uso de punto y coma

Siempre usar punto y coma al final de las declaraciones.

9. Gestión de Dependencias

9.1. Evita el Uso de Librerías Innecesarias

Antes de agregar una nueva librería, se debe evaluar si realmente es necesaria. Instalar dependencias innecesarias puede inflar el tamaño de tu proyecto y hacer más compleja su actualización a futuro.

9.2. Versiones Fijas

Fijar las versiones de las dependencias para evitar problemas de compatibilidad cuando otras personas clonen el proyecto.