Documento de Estilos de Código

Guía de Estilo de Código en JavaScript

Algunas prácticas recomendadas para escribir código JavaScript claro y eficiente:

1. Convenciones de Nombres

- **Variables y funciones**: Usamos `camelCase` para nombres de variables y funciones.

```
const fetchedPedido = response.data.pedido;
function mostrarNotificacion() { ... }
```

- **Constantes**: Para valores constantes, usamos `SCREAMING_SNAKE_CASE`.

```
const MAX_ATTEMPTS = 5;
```

2. Declaraciones de Variables

- Usamos `let` o `const` en lugar de `var`.

```
const pedidoIndex = data.findIndex(p => p.id == idPedido);
let data = readData();
```

3. Espaciado y Sangría

- Usamos una sangría consistente de un tab, de 4 espacios de tamaño.

```
function example() {
    if (true) {
        console.log('Hola');
    }
}
```

4. Puntos y Comas

- Usamos punto y coma al final de cada linea. Aunque JavaScript tiene inserción automática de puntos y comas, es una buena práctica usarlos explícitamente para evitar errores.

```
const now = new Date();
```

5. Funciones de Flecha

- Usamos funciones de flecha (`=>`) para funciones anónimas cuando sea posible. Son más concisas y manejan el contexto `this` de manera predecible.

```
const readData = () => {
    // Código a ejecutar
};
```

6. Plantillas Literales

- Usamos plantillas literales (`` ` ``) en lugar de concatenación de cadenas para insertar variables.

```
const mensaje = `El pedido #ID${fetchedPedido.id} ya tiene un transportista
seleccionado.`;
```

7. Desestructuración

- Usamos la desestructuración de objetos y arrays para acceder a las propiedades de forma más limpia.

```
const { idPedido, metodo_pago, entrega_fecha, retiro_fecha, transportista,
precio } = req.body;
```

8. Comentarios

- Usamos comentarios cuando sea necesario para explicar lógica compleja. Para comentarios extensos usamos (`/* */`) y para comentarios en línea (`//`) para notas rápidas.

```
// Verifica si el pedido ya está confirmado
if (fetchedPedido.estado === 'Confirmada') {
        // Código a ejecutar
        return;
}
```

9. Operador Ternario

- Usamos operadores ternarios para condiciones simples, pero evita anidarlos excesivamente para mantener la legibilidad.

```
const imageSrc = nombreImagen === 'Success' ? '/Success.png' : '/Fail.png';
```

10. Uso de llaves

- La llave de apertura de un bloque de código se encuentra en la misma linea del encabezado del bloque, y la llave de cierre se encuentra en una nueva linea, al mismo nivel de independencia que el inicio del encabezado del bloque

```
if (year > currentYear) {
        return true;
}
```

11. Uso de iteradores

- Utilizamos funciones de ordenes superior para iterar sobre colecciones cuando es posible, en vez de realizar ciclos `for`.

```
listaPedidos.forEach((pedido) => {
        mostrarNotificacion(pedido.mensaje);
        handleEliminarPedido(pedido.id);
});
```

Guía de Estilo de Código en React

Algunas recomendaciones de estilo para React que hemos seguido:

1. Estructura de Componentes

- Usamos **Functional Components** y hooks siempre que sea posible, en lugar de componentes de clase, declarándolos mediante funciones flecha

```
const Chip = ({ texto }) => {
    return (
        <div>
            {texto}
        </div>
    );
};
```

2. Nombrado de Componentes

- Usamos **PascalCase** para el nombre de los componentes. Los archivos de componentes también coinciden con el nombre del componente.

```
// Componente: Modal.jsx
const Modal = () => { ... };
```

3. Props y Estado

- Usamos desestructuración para acceder a las props y el estado dentro de los componentes.

```
const Notificacion = ({ title, message }) => {
    return (
        <div>
              <div>{title}</div>
              <div>{message}</div>
        </div>
    );
};
```

4. Hooks

- Usamos hooks como `useState` y `useEffect` en lugar de los métodos de ciclo de vida de las clases en la parte superior del componente.

```
const [loading, setLoading] = React.useState(false);

useEffect(() => {
        const fetchCotizacion = async () => {
                //codigo a ejecutar...
        };
        fetchCotizacion();
}, []);
```

5. Condicionales en JSX

- Evitamos condicionales complejos dentro del JSX. Usa operadores ternarios o variables auxiliares para manejar la lógica de renderizado condicional.

```
  return (
        <Router>
          {isMobile ? (<Routes> ... </Routes>
          ) : ( <PhoneFrame> ... </PhoneFrame>
```

```
          )}
        </Router>
  );
```

6. Componentes Pequeños y Reutilizables

- Dividimos los componentes grandes en partes más pequeñas y reutilizables. Cada componente debe tener una única responsabilidad.

```
const Chip = () => { ... };
const Notificacion = () => { ... };
```

7. Teoría del "State Lifting"

- Mantemos el estado en el componente de nivel más alto que lo necesite y pasa el estado a componentes hijos a través de props.

```
const Transportista = () => {
        const [notificaciones, setNotificaciones] = useState([]);

        const handleClickNotificacion = (id) => {
                setNotificaciones((prev) => prev.filter((notif) => notif.id !== i
d));
        };

      return (
              <>
                      {notificaciones.map((notif) => (
                              <Notificacion
                                      title={"TANGO APP"}
                                      message={notif.mensaje}
                              />
                      ))}
              </>
      );
}
```