Hoja de estilo de código - TP6 ICS - Grupo 3

1. **Indentación y espaciado**

   Usamos la librería Prettier que nos permite tener un formato consistente automático en cada archivo, generando indentación uniforme, límite de longitud en líneas y formateo automático.

2. **Nombres de variables, funciones y clases**

   Utilizamos camelCase para nombres de funciones y variables en el frontend como en el backend, dando mayor consistencia:

```
const procesarPago = async (pagoData: PagoData) => {
  setLoading(true);
  setError(null);
```

   Para los componentes React utilizamos PascalCase:

```
const InterfazTransportista: React.FC = () => {
  const { user, logout } = useAuth();
```

   Y para constantes globales optamos por usar SNAKE_CASE:

```
const API_BASE_URL = "http://localhost:3000/api";
```

3. **Estructura de funciones**

   Para la gran mayoría de las funciones utilizamos arrow functions, dando una responsabilidad clara y única para cada una:

```
const cargarCotizacionesPrevias = () => {
  const cotizacionesPrevias = localStorage.getItem('cotizacionesConfirmadas');
  return cotizacionesPrevias ? JSON.parse(cotizacionesPrevias) : [];
};
```

   También usamos funciones asíncronas para el manejo del fetch a la API, manejando los errores y la data obtenida con try-catch

```
const fetchTarjetas = useCallback(async () => {
  setLoading(true);
  setError(null);
  try {
    const response = await getTarjetasByDador(dador_id);
    setTarjetas(response);
  } catch (err: any) {
    setError(err.message || "Error al obtener las tarjetas");
  } finally {
    setLoading(false);
  }
}, [dador_id]);
```

4. **Control de errores**

En el backend devolvemos los errores en formato JSON en caso de encontrarlos y mostrarlos.

```
const getMisCotizacionesConfirmadas = (req, res) => {
  const { id } = req.params;
  const query = `SELECT c.* FROM cotizaciones c INNER JOIN transportistas t ON t.id = c.transportista_id WHERE t.id = ? AND c.estado = 'Confirmado'`;
  connection.query(query, [id], (err, results) => {
    if (err) {
      return res
        .status(500)
        .json({
          error:
            "Error al obtener las cotizaciones confirmadas del transportista logueado",
        });
    }
    res.json(results);
  });
};
```

En el frontend manejamos los errores obteniéndose y asignadolos a un hook de React llamado useState, luego si es necesario los mostramos por consola o los mostramos detallados en la interfaz.

```
const [error, setError] = useState<string>("");
```

5. **Comentarios y documentación**

   Documentamos y comentamos funciones complejas o lógicas no evidentes, y posibles mejoras con //

```
//ver en el momento que cuenta nos da de prueba https://ethereal.email/create
//para ver los correos simulados deben entrar a messages
const transporter = nodemailer.createTransport({
  host: "smtp.ethereal.email",
  port: 587,
  auth: {
    user: "maximillian.watsica90@ethereal.email",
    pass: "5wpE9kSqyy1nSuaz9U",
  },
});
```

6. **Estructura de archivos**

   En el backend usamos una estructura MVC, organizando controladores, rutas y configuraciones en carpetas separadas

   En el frontend usamos la estructura Modular Architecture, en el cual organizamos el código según funcionalidades o características específicas de la aplicación.

7. **Comillas y sintaxis**

   Usamos comillas simples en lugar de dobles para definir strings en TypeScript y JavaScript.

   Finalizamos todas las líneas con punto y coma.

   Uso de plantillas de string cuando necesites interpolar valores en un string;

```
Swal.fire({
  title: 'Nueva Cotización Confirmada',
  text: `Tienes ${nuevasCotizaciones.length} nueva(s) cotización(es) confirmada(s).`,
  icon: 'info',
  confirmButtonText: 'OK',
  timer: 2000,
  timerProgressBar: true,
});
```

8. **Uso de constantes y variables**

   Usamos const para valores que no cambian y let cuando una variable puede cambiar:

```
let pagoData: any = {
  cotizacion_id: cotizacion.id,
  forma_pago_id: metodosPago.find((metodo) => metodo.descripcion === formaPagoSeleccionada)?.id,
  importe_a_pagar: importeCotizacion,
  dador_id: dador_id,
};
```

   En este caso usamos let en pagoData ya que si el dador selecciona la forma de pago luego, se le agregan más propiedades a pagoData, cosa que con const no se podría realizar.

9. **Manejo de estilos**

   Usamos Tailwind CSS para el diseño y estructura responsive de los componentes de React, con la paleta de colores asignada configurada en nuestro archivo tailwind.config.js:

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: ["./src/**/*.{ts,tsx}"],
  theme: {
    extend: {
      colors: {
        'jade': {
          '50': '#CAF0F8',
          '100': '#CAF0F8',
          '200': '#90E0EF',
          '300': '#90E0EF',
          '400': '#00B4D8',
          '500': '#00B4D8',
          '600': '#0077B6',
          '700': '#0077B6',
          '800': '#03045E',
          '900': '#03045E',
          '950': '#03045E',
        },
      },
      screens: {
        'xs': '350px',
        'sm': '480px',
        'md': '768px',
        'lg': '1024px',
        'xl': '1366px',
        '2xl': '1600px',
        '3xl': '1920px',
      }
    },
  },
  plugins: [],
};
```

10. **Buenas prácticas**

Uso de Hooks personalizados en el frontend encapsulando la lógica y evitando duplicación de código manteniendo el código modular.
Validaciones de entrada en las funciones, asegurando que los datos sean correctos antes de procesar el pago.
Funcionalidad en el backend separada en controladores, siguiendo el patrón MVC. Mantenimiento lógica organizada y modular.