Universidad Tecnológica Nacional
Facultad Regional Córdoba

Ingeniería y calidad de software

ESTILOS DE CÓDIGO

Integrantes:

Beluzzo Gino, Federico	79233
Carrasco, Francisco	87930
De Dios, Guillermo	86635
Escudero, Octavio	89866
Galiano, Milagros	89513
Marchiori, Nicolás	85335
Pizarro, Emilio	87471
Vega, Candela	87345

Docentes:

María Cecilia Massano
Georgina Gisela González
Laura Inés Covaro

Grupo 2

4K4 - 2024

Índice

Introducción

El propósito de este documento es proporcionar un conjunto de normas para la escritura de código en nuestro equipo, con el fin de mejorar la colaboración y asegurar la calidad del desarrollo. Mantener un estilo de codificación uniforme facilita no solo la lectura del código, sino también su mantenimiento y escalabilidad a medida que los proyectos crecen.

Estas pautas están diseñadas para garantizar que todo miembro del equipo pueda trabajar de manera coherente, siguiendo prácticas recomendadas y minimizando errores.

En las siguientes secciones, se detallan las reglas que deben aplicarse al escribir código utilizando TypeScript.

En este documento, se incluyen la nombres de variables, estructura del código, uso adecuado de espacios en blanco, entre otros aspectos fundamentales. Siguiendo estas recomendaciones, lograremos un código más limpio, fácil de entender y de modificar en futuras iteraciones.

Guía de Estilo de Código

1. Convenciones de Nombres y nombres de variables

- **Componentes**: Usar `PascalCase` para los nombres de los archivos y componentes.
    - Ejemplo: `Body.tsx`, `Footer.tsx`

El nombre de las variables debe reflejar de forma clara su propósito. Esto facilita la comprensión del código a los demás desarrolladores. Algunas reglas son:

- Utilizar **camelCase** para nombrar variables y funciones. Los nombres de las variables deben ser lo más descriptivos posible.
    - ❌ **Incorrecto**: `const tc = 10;`
    - ✅ **Correcto**: `const tipoCarga= 25;`
- Para variables booleanas, utiliza prefijos que denoten una condición o estado, como is, es, etc.
    - Ejemplo: `const esPendienteRetiro = true;`

2. Estructura de Componentes

En React, es recomendable utilizar **componentes funcionales** en lugar de componentes de clase, ya que son más sencillos y favorecen el uso de hooks.

Los componentes funcionales son funciones de JavaScript que reciben props como argumento y retornan JSX.

- Los componentes deben ser pequeños y tener una única responsabilidad. Si un componente maneja múltiples tareas o se vuelve demasiado grande, deberías dividirlo en componentes más pequeños.
- Los componentes deben aceptar **props** como parámetros y evitar efectos secundarios dentro de las funciones de renderizado.

    - Ejemplo:

```
type FooterProps = {
  title: string;
};
```

```
const Footer: React.FC<FooterProps> = ({ title }) => {
  return <footer>{title}</footer>;
};
```

- **Hooks**: Usar hooks de React (`useState`, `useEffect`, etc.) en lugar de métodos de ciclo de vida de clase. Los **Hooks** en React son funciones que permiten usar el estado y otras características de React en **componentes funcionales**.

3. Definición de Props y Estado

El tipado correcto de props y estado es fundamental para asegurar la robustez del código.

- **Tipado estricto:** Siempre definir los tipos para las props y el estado. Evita el uso de `any`.
    - Ejemplo:

```
type PerfilUsuario = {
      name: string;
      age: number;
      esAdmin: boolean;
};
```

- **Interfaces**: Definir interfaces para props y estados.

  Ejemplo:

```
interface Domicilio {
  calle: string;
  numero: string;
  localidad: string;
  provincia: string;
  referencia: string;
}

const [domicilioRetiro, setDomicilioRetiro] =
useState<Domicilio>({
  calle: "",
  numero: "",
  localidad: "",
  provincia: "",
  referencia: "",
});
```

4. Manejo de Eventos

- **Funciones de Manejo**: Usar nombres descriptivos y específicos para funciones de manejo de eventos.
    - Ejemplo:
      ```
      const handleFileChange = (e:
      React.ChangeEvent<HTMLInputElement>) => {
        // código aquí
      };
      ```
- Asigna los manejadores de eventos a través de props y mantén la lógica en un nivel superior cuando sea posible para evitar duplicación.

5. Efectos y Llamadas Asíncronas

- **Manejo de Errores**: Usa `try-catch` para manejo de errores en funciones asíncronas.

  Ejemplo:
  ```
  const publicarPedido = async () => {
    try {
      await postPublicarPedido();
      mostrarMensaje("Pedido publicado con éxito", "success");
    } catch (error) {
      mostrarMensaje("No se pudo publicar el pedido", "error");
    }
  };
  ```

6. Estilo de Código

Mantener un estilo de código consistente ayuda a la legibilidad y a la colaboración.

- Usar **2 espacios** para la indentación.
- Asegúrate de que cada archivo no exceda las **300 líneas**. Si un archivo crece demasiado, divídelo.
- Usar **espacios en blanco** de manera adecuada para separar secciones lógicas del código.
- Limitar el **ancho de línea a 80-100** caracteres.
- Usar espacios alrededor de operadores y después de comas.
- Explicar el “por qué” detrás del código en los comentarios, no el “qué”.

7. Organización del Código

- **Importaciones**: Ordenar las importaciones en el siguiente orden: bibliotecas externas, componentes internos, estilos, etc.

  Ejemplo:
  ```
  import { Button, TextField } from "@mui/material";
  import { useState } from "react";
  import clases from "../Styles/Componente.module.css";
  ```

8. JSX y Componentes

- Mantener los elementos JSX limpios y bien estructurados.
- Usar **paréntesis** para JSX de múltiples líneas.
- Los componentes deben ser lo más **puros** posible, evitando efectos secundarios dentro del render.
- Usar módulos CSS para **estilos** locales y específicos del componente.

  Ejemplo:
  ```
  <div className={clases.body}>
    {/* contenido aquí */}
  </div>
  ```

9. Pruebas

- Las pruebas son esenciales para garantizar la calidad del código.
- Escribir **pruebas unitarias** para todas las funciones y componentes principales.
- Usar **librerías** como `Jest` y `React Testing Library` para pruebas unitarias y de integración.

10. Detalles Específicos del Archivo `footer.tsx`

1. **Importaciones**: Importar componentes y estilos de manera ordenada.

   Ejemplo:
   ```
   import { Typography, Grid, Link, IconButton } from
   "@mui/material";
   import { Facebook, Twitter, Instagram } from
   "@mui/icons-material";
   import clases from "../Styles/Componente.module.css";
   ```

2. **Estructura del JSX**: Usar comentarios para separar secciones dentro del JSX para una mejor organización.

   Ejemplo:
   ```
   {/* Sección 1: Nombre de la App */}
   <Grid item xs={12} sm={4}>
     <Typography variant="h5" className={clases.footerTitle}>
       TANGO APP
     </Typography>
     <Typography variant="body2" color="textSecondary"
   className={clases.footerSubtitle}>
       Creado por el Grupo N° 2
     </Typography>
   </Grid>
   ```

3. **Uso de Clases CSS**: Usar clases definidas en módulos CSS para estilos específicos del componente.

   Ejemplo:
   ```
   <footer className={clases.footer}>
     {/* contenido aquí */}
   </footer>
   ```

4. **Accesibilidad y Usabilidad**: Asegurarse de que los enlaces sean accesibles y las secciones sean claramente identificables.

5. **Consistencia en el Código**: Mantener el mismo estilo y estructura de código que en otros archivos de componentes.

Bibliografía

https://google.github.io/styleguide/tsguide.html#naming
https://ts.dev/style/#syntax