TRABAJO PRÁCTICO
N°6 “Implementación de US”

BUENAS PRÁCTICAS Y REGLAS DE ESTILO DE CÓDIGO

INGENIERÍA Y CALIDAD DE SOFTWARE

DOCENTES:

María Cecilia Massano

Constanza Garnero

CURSO 4K2 - GRUPO N°9:

Castro Wachs, Trinidad - 89828

Colque, Sebastián - 90316

Feretti, Lucía - 89553

Fuxa, Pedro Joaquín - 89303

Ibarra, Mauro - 90025

UTN
Facultad Regional Córdoba

La implementación de la User Story “Aceptar Cotización” será desarrollada utilizando **JavaScript** como lenguaje principal, con un enfoque tanto en el **frontend** como en el **backend**. En el frontend, emplearemos **React** junto con **Vite** como herramienta de construcción rápida. En el backend, usaremos **Node.js** junto con el framework **Express** para gestionar el servidor y las APIs.

El propósito de esta guía es establecer una base sólida de buenas prácticas que asegure un código limpio, legible y fácil de mantener a lo largo de todo el ciclo de vida del proyecto. Abordaremos convenciones de JavaScript. La consistencia en el estilo del código ayudará a mejorar la colaboración entre los miembros del equipo y facilitará la escalabilidad y mantenibilidad del sistema a largo plazo.

1. Estilo de Código

1.1. Indentación

Utiliza 2 espacios para la indentación en lugar de tabuladores. Esto asegura la coherencia en diferentes editores y plataformas. Ejemplo:

```
function example() {
  const foo = 'bar';
  if (foo) {
    console.log(foo);
  }
}
```

1.2. Uso de Punto y Coma

Finaliza siempre las declaraciones con un punto y coma (;) para evitar posibles errores en el comportamiento del código. Ejemplo:

```
const name = 'John';
console.log(name);
```

1.3. Comillas

Usa comillas simples (' ') para cadenas de texto en lugar de comillas dobles (" "), a menos que el contenido de la cadena contenga comillas simples. Ejemplo:

```
const message = 'Hello, World!';
const quote = "It's a beautiful day!";
```

1.4. Límites de Línea

Mantén las líneas de código con un límite de 80 caracteres para mejorar la legibilidad.

2. Nomenclatura

2.1. Nombres de Variables y Funciones

Para nombres de funciones, use camelCase, comenzando con un carácter en minúscula. Utilice nombres concisos, legibles por humanos y semánticos cuando sea apropiado, evitando abreviaturas innecesarias. Ejemplo:

```
const userName = 'Alice';
function getUserInfo() {
  // código aquí
}
```

2.2. Nombres de Clases

Utiliza PascalCase para nombres de clases. Ejemplo:

```
class UserAccount {
  constructor(name) {
    this.name = name;
  }
}
```

3. Funciones y Métodos

3.1. Funciones Flecha

Prefiere las funciones flecha (`=>`) para funciones anónimas y callbacks. Esto hace el código más conciso y mantiene el contexto de `this`. Ejemplo:

```
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2);
```

3.2. Longitud de las Funciones

Mantén las funciones cortas y concisas. Si una función es demasiado larga, considera dividirla en funciones más pequeñas.

4. Manejo de Errores

4.1. Uso de `try/catch`

Utiliza bloques `try/catch` para manejar errores de manera adecuada, especialmente en operaciones asincrónicas. Ejemplo:

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error al obtener datos:', error);
  }
```

```
}
```

4.2. Propagación de Errores

Propaga los errores adecuadamente usando `throw` o manejadores de errores para no silenciar errores importantes.

5. Asincronía

5.1. `async`/`await` sobre Promesas

Prefiere el uso de `async/await` sobre las promesas tradicionales (`then/catch`) para mejorar la legibilidad y el manejo de errores. Ejemplo:

```
async function getData() {
  try {
    const data = await fetchData();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
```

6. Documentación

6.1. Comentarios

Los comentarios son fundamentales para escribir buenos ejemplos de código ya que aclaran la intención del código y ayudan a los desarrolladores a entenderlo. Deben ser claros y concisos para explicar el propósito de funciones complejas o lógicas no triviales.

- Si el propósito o la lógica del código no es obvio, añade un comentario con tu intención, como se muestra debajo:

  ```
  let total = 0;

  // Calcula la suma de los cuatro primeros elementos de arr
  for (let i = 0; i < 4; i++) {
    total += arr[i];
  }
  ```

- Por otro lado, reformular el código en prosa no es un buen uso de los comentarios:

  ```
  let total = 0;

  // Bucle de 1 a 4
  for (let i = 0; i < 4; i++) {
    // Adicionar valor al total
    total += arr[i];
  ```

```
}
```

- Los comentarios tampoco son necesarios cuando las funciones tienen nombres explícitos que describen lo que están haciendo. Escriba:

```
closeConnection(); // Cierra la conexión
```

6.2. JSDoc

Utiliza JSDoc para documentar funciones, métodos y clases. Ejemplo:

```
/**
 * Calcula el área de un círculo.
 * @param {number} radius - El radio del círculo.
 * @return {number} El área del círculo.
 */
function calculateCircleArea(radius) {
  return Math.PI * radius * radius;
}
```

7. Uso de Variables y Constantes

7.1. Uso de `const` y `let`

Utiliza **`const`** para declarar variables que no cambian y **`let`** para aquellas que lo hacen. Evita el uso de `var`. Ejemplo:

```
const MAX_USERS = 100;
let currentUserCount = 0;
```

8. Organización de Código

8.1. Módulos

Divide el código en **módulos** (archivos separados) para mejorar la organización y mantener cada archivo con una responsabilidad específica. Ejemplo:

```
// userController.js
export function getUser() {
  // código aquí
}
```

9. Bucles y sentencias condicionales

9.1. Inicialización de bucle

Cuando los bucles son requeridos, elegir el apropiado entre for(;;), for...of, while, etc.

- Al iterar a través de todos los elementos de la colección, evite usar el clásico bucle for (;;); es preferible for...of o forEach(). Tenga en cuenta que si está utilizando una colección que no es un Array, tienes que comprobar que for...of es realmente compatible (requiere que la variable sea iterable), o que el método forEach() está realmente presente. Use for...of:

```
const perros = ["Rex", "Lassie"];
for (const perro of perros) {
  console.log(perro);
}
```

- O forEach():

```
const perros = ["Rex", "Lassie"];
perros.forEach((perro) => {
  console.log(perro);
});
```

- No use for (;;) — no solo tienes que agregar un índice extra, i, sino que también tienes que rastrear la longitud de la matriz. Esto puede ser propenso a errores para principiantes.

```
const perros = ["Rex", "Lassie"];
for (let i = 0; i < perros.length; i++) {
  console.log(perros[i]);
}
```

- Asegúrese de definir correctamente el inicializador utilizando la palabra clave const para for...of o let para los otros bucles. No lo omitas. Estos son ejemplos correctos:

```
const gatos = ["Athena", "Luna"];
for (const gato of gatos) {
  console.log(gato);
}
for (let i = 0; i < 4; i++) {
  result += arr[i];
}
```

- El siguiente ejemplo no sigue las pautas recomendadas para la inicialización (implícitamente crea una variable global y fallará en modo estricto):

```
const gatos = ["Athena", "Luna"];
for (i of gatos) {
  console.log(i);
}
```

- Cuando necesite acceder tanto al valor como al índice, puede usar .forEach() en lugar de for (;;). Escriba:

```
const gerbils = ["Zoé", "Chloé"];
gerbils.forEach((gerbil, i) => {
  console.log(`Gerbil #${i}: ${gerbil}`);
```

```
});
```

- No escriba:

```
const gerbils = ["Zoé", "Chloé"];
for (let i = 0; i < gerbils.length; i++) {
  console.log(`Gerbil #${i}: ${gerbils[i]}`);
}
```

Advertencia: Nunca utilice for...in en matrices y cadenas.

Nota: Considere no usar un bucle for en absoluto. Si estás utilizando un Array (o un String para algunas operaciones), considere usar más métodos de iteración semántica en su lugar, como map(), every(), findIndex(), find(), includes(), y muchos más.

9.2. Sentencias de control

Hay un caso notable a tener en cuenta para las sentencias de control if...else. Si la declaración if termina con return, no agregue una declaración else.

Continúe justo después de la instrucción if. Escriba:

```
if (prueba) {
  // Realizar algo si la prueba es verdadera
  // …
  return;
}
// Realizar algo si la prueba es falsa
// …
```

No escriba:

```
if (prueba) {
  // Realizar algo si la prueba es verdadera
  // …
  return;
} else {
  // Realizar algo si la prueba es falsa
  // …
}
```

9.3. Llaves con sentencias de flujo de control y bucles

Si bien las declaraciones de flujo de control como if, for y while no requieren el uso de llaves cuando el contenido se compone de una sola declaración, siempre debe usar llaves. Escriba:

```
for (const carro of carrosAlmacenados) {
  carro.pintar("rojo");
}
```

No escriba:

```
for (const carro of carrosAlmacenados) carro.pintar("rojo");
```

Esto evita olvidarse de agregar las llaves al agregar más declaraciones.

10. Operadores

10.1. Operadores condicionales

Cuando desee almacenar en una variable un valor literal dependiendo de una condición, use un operador condicional (ternario) en lugar de una sentencia if...else. Esta regla también se aplica cuando se devuelve un valor. Escriba:

```
const x = condicion ? 1 : 2;
```

No escriba:

```
let x;
if (condicion) {
  x = 1;
} else {
  x = 2;
}
```

El operador condicional es útil al crear cadenas para registrar información. En tales casos, el uso de una instrucción regular if...else conduce a largos bloques de código para una operación secundaria como el registro, ofuscando el punto central del ejemplo.

10.2. Operador de igualdad estricta

Preferir los operadores de igualdad estricta === y de desigualdad !== sobre los operadores de igualdad flexible == y de desigualdad !=.

Use los operadores estrictos de igualdad y desigualdad de esta forma:

```
nombre === "Shilpa";
edad !== 25;
```

No use los operadores sueltos de igualdad y desigualdad, como se muestra a continuación:

```
nombre == "Shilpa";
edad != 25;
```

Si necesita usar == o !=, recuerde que == null es el único caso aceptable. Como TypeScript fallará en todos los demás casos, no queremos tenerlos en nuestro código de ejemplo. Considere agregar un comentario para explicar por qué lo necesita.

11. Template literals

Para insertar valores en cadenas, use Plantillas literales.

- Aquí hay un ejemplo de la forma recomendada de usar plantillas literales. Su uso evita muchos errores de espaciado.

```
const nombre = "Shilpa";
console.log(`¡Hola! Soy ${nombre}!`);
```

- No concatenar cadenas así:

```
const nombre = "Shilpa";
console.log("¡Hola! Soy" + nombre + "!"); // ¡Hola! SoyShilpa!
```

- No abuses de las plantillas literales. Si no hay sustituciones, use una cadena literal normal en su lugar.