Digital NAO

× TECMILENIO

Nombre:

Sanders Josué Leiva Larios.

NAO ID:

2990

Fecha:

12/07/2024

Trayectoria:

Full Stack Developer Core

Caso:

Buenas prácticas para el uso de tipos en TypeScript – Sprint 1

TypeScript, un superconjunto de JavaScript que añade tipado estático, ha ganado popularidad en los últimos años por su capacidad para mejorar la legibilidad, la mantenibilidad y la robustez del código. El uso efectivo del sistema de tipos de TypeScript puede marcar la diferencia entre un proyecto sólido y uno propenso a errores.

1. Aprovechar la inferencia de tipos

TypeScript ofrece un potente sistema de inferencia de tipos que permite al compilador deducir los tipos de variables y expresiones a partir de su uso. Esto significa que no es necesario especificar explícitamente el tipo de cada variable, lo que ahorra tiempo y mejora la legibilidad del código.

Ejemplo:

TypeScript

```
const suma = (x: number, y: number) => x + y;
const resultado = suma(5, 3);
```

En este ejemplo, TypeScript infiere que las variables x e y son de tipo number y que la función suma devuelve un valor de tipo number. No es necesario especificar explícitamente estos tipos, ya que el compilador los deduce a partir de las operaciones realizadas.

2. Utilizar interfaces para definir estructuras de datos

Las interfaces en TypeScript permiten definir la forma de los objetos, especificando las propiedades que deben tener y sus tipos de datos correspondientes. Esto ayuda a garantizar la consistencia del código y facilita la comprensión de la estructura de los datos.

Ejemplo:

TypeScript

```
interface Usuario {
```

```
  id: number;
  nombre: string;
  correoElectronico: string;
}

const usuario: Usuario = {
  id: 1,
  nombre: "Juan Pérez",
  correoElectronico: "juan.perez@ejemplo.com"
};
```

En este ejemplo, la interfaz Usuario define la estructura de un objeto que representa a un usuario, con propiedades para su ID, nombre y correo electrónico. La variable usuario se declara como de tipo Usuario, lo que garantiza que tenga las propiedades y tipos de datos correctos.

3. Emplear tipos genéricos para mayor flexibilidad

Los tipos genéricos en TypeScript permiten crear funciones, clases e interfaces que pueden trabajar con diferentes tipos de datos. Esto aumenta la reutilizabilidad del código y lo hace más flexible.

Ejemplo:

TypeScript

```
function identidad<T>(valor: T): T {
  return valor;
}
const numero: number = identidad(5);
const texto: string = identidad("Hola");
```

En este ejemplo, la función identidad es genérica, lo que significa que puede trabajar con cualquier tipo de dato (T). Las variables número y texto se declaran como de tipo number y string, respectivamente, y la función identidad las devuelve sin modificar su tipo original.

4. Aprovechar las uniones y tipos literales

Las uniones de tipos en TypeScript permiten especificar que una variable o expresión puede tener uno de varios tipos posibles. Esto es útil para manejar situaciones en las que el valor de una variable puede variar.

Ejemplo:

TypeScript

```
function obtenerValor(mixto: number | string): string {
  if (typeof mixto === "number") {
    return mixto.toString();
  } else {
    return mixto;
  }
}
const resultado1 = obtenerValor(10); // Resultado: "10"
const resultado2 = obtenerValor("Hola"); // Resultado: "Hola"
```

En este ejemplo, la función obtenerValor recibe un parámetro de tipo number | string, lo que indica que puede ser un número o una cadena de texto. La función comprueba el tipo del valor y lo devuelve como una cadena de texto.

Los tipos literales en TypeScript permiten especificar valores exactos para variables y expresiones. Esto es útil para garantizar la precisión de los datos y evitar errores.

Ejemplo:

TypeScript

```
const estado: "pendiente" | "aprobado" | "rechazado" =
"pendiente";
```

En este ejemplo, la variable estado se declara como de tipo literal "pendiente" | "aprobado" | "rechazado", lo que significa que solo puede tener uno de esos tres valores. Esto ayuda a prevenir errores como asignar un valor no válido a la variable.

5. Implementar comprobaciones de tipo estrictas

TypeScript ofrece diferentes niveles de comprobación de tipos, desde la más permisiva hasta la más estricta. Se recomienda utilizar un nivel de comprobación estricto para detectar errores de tipo en tiempo de compilación y prevenir problemas durante la ejecución del código.

Ejemplo:

TypeScript

```
// Habilitar comprobación de tipo estricta
"use strict";

function sumar(x: number, y: number): number {
  return x + y;
}
const resultado = sumar(5, "10"); // Error en tiempo de
compilación: Se esperaba un número para el argumento 'y'
```

En este ejemplo, al habilitar la comprobación de tipo estricta, el compilador detecta que se está pasando una cadena de texto ("10") al segundo argumento de la función sumar, que espera un número. Esto genera un error en tiempo de compilación, previniendo un posible error durante la ejecución del código.

6. Aprovechar las herramientas de TypeScript

Existen diversas herramientas disponibles para facilitar el uso de TypeScript, como editores de código con soporte para TypeScript, extensiones para IDEs y linters estáticos. Estas herramientas pueden ayudar a identificar errores de tipo, sugerir mejoras y automatizar tareas relacionadas con el sistema de tipos.

Ejemplos de herramientas:

- **Visual Studio Code:** Uno de los editores de código más populares con soporte integrado para TypeScript.
- **WebStorm:** Un IDE completo con potentes funciones para TypeScript, como refactorización, completado de código y análisis de estático.
- **TSLint:** Un linter estático para TypeScript que ayuda a identificar y corregir errores de estilo y violaciones de las mejores prácticas.

7. Documentar el código utilizando comentarios de tipo

Los comentarios de tipo en TypeScript permiten documentar la estructura de datos, las funciones y las interfaces, proporcionando información valiosa para otros desarrolladores que trabajan con el código.

Ejemplo:

TypeScript

```
/**
 * Función que suma dos números
 *
 * @param x El primer número
 * @param y El segundo número
 * @returns La suma de x e y
 */
function sumar(x: number, y: number): number {
  return x + y;
}
```

En este ejemplo, los comentarios de tipo documentan la función sumar, especificando los tipos de sus parámetros y valor de retorno. Esto mejora la legibilidad del código y facilita su comprensión.

Aplicación de las mejores prácticas en un proyecto real

Consideremos un proyecto de aplicación web para gestionar productos en una tienda online.

1. Definir interfaces para los productos:

TypeScript

```
interface Producto {
  id: number;
  nombre: string;
  precio: number;
  stock: number;
}
```

2. Utilizar tipos genéricos para funciones de gestión de productos:

TypeScript

```
function obtenerProductos(): Promise<Producto[]>;

function agregarProducto(producto: Producto): Promise<void>;

function actualizarProducto(producto: Producto): Promise<void>;

function eliminarProducto(id: number): Promise<void>;
```

3. Aprovechar las uniones de tipos para manejar diferentes tipos de búsqueda:

TypeScript

```
function buscarProducto(criterio: string | number): Producto[] {
  // Implementar la lógica de búsqueda
}
```

4. Implementar comprobaciones de tipo estrictas para mayor seguridad:

TypeScript

```
// Habilitar comprobación de tipo estricta
"use strict";

// ... código del proyecto ...
```

5. Documentar el código con comentarios de tipo:

TypeScript

```
/**
 * Función que obtiene todos los productos
 *
 * @returns Una lista de productos
 */
function obtenerProductos(): Promise<Producto[]> {
  // Implementar la lógica para obtener productos
}
```

Al seguir estas buenas prácticas, los desarrolladores pueden crear un código TypeScript más robusto, legible y mantenible, lo que se traduce en un proyecto de mayor calidad y menos propenso a errores.