Desarrollo de microservicios con Typescript y NestJS

Jassyr Juárez Vázquez

Presiona espacio para avanzar →

Bienvenida y presentación

- **Instructor**: Jassyr Juárez Vázquez
- **Objetivos**:
  - Conocer TypeScript y sus ventajas
  - Obtener una visión general de NestJS
  - Desarrollar una aplicación sencilla con NestJS y TypeScript

Agenda

1. Introducción a TypeScript
2. Introducción a NestJS
3. Profundizando en NestJS y TypeScript
4. Mejores Prácticas y Técnicas Avanzadas
5. Q&A y Cierre

¿Qué es TypeScript?

TypeScript es un lenguaje de código abierto desarrollado por Microsoft. Se trata de un supraconjunto de JavaScript, lo que significa que al saber Javascript se puede extender su conocimiento al incluir características que antes no estaban disponibles.

Tipado estático opcional

La característica principal de TypeScript es su sistema de tipos. Se puede identificar el tipo de datos de una variable o un parámetro mediante una sugerencia de tipo. Con las sugerencias de tipo, se describe la forma de un objeto, lo que proporciona una documentación mejor y permite a TypeScript validar que el código funciona correctamente.

Herramientas de desarrollo mejoradas

Mediante la comprobación de tipos estáticos, TypeScript al principio del desarrollo detecta problemas de código que JavaScript normalmente no puede detectar hasta que el código se ejecuta en el explorador. Los tipos también permiten describir lo que debe hacer el código.

Prerequisitos

Node debe estar instalado.

Verificar la versión con `node --version`. Si no está instalado, ejecutar:

```
nvm install 18.20.4
nvm use 18.20.4
```

Instalar typescript

```
npm install -g typescript
```

Crear una carpeta para trabajar en los ejemplos e inicializar typescript en ella

```
mkdir proyecto-typescript
cd proyecto-typescript
tsc --init
```

Otra opción en línea es Typescript playground

Ejercicio práctico

Crear archivos con código Typescript, compilarlos con tsc y ejecutarlos con node

```typescript
// declaración de variables con tipos predefinidos
let nombre: string = 'Restaurante El Buen Sabor';
let capacidad: number = 50;
let abierto: boolean = true;

console.log(`Nombre: ${nombre}, Capacidad: ${capacidad}, Abierto: ${abierto}`);

// Inferencia de tipos
let ciudad = "Apizaco"
const poblacion = 100000
```

```
tsc ejemplo.ts
node ejemplo.js
```

Salida:

```
Nombre: Restaurante El Buen Sabor, Capacidad: 50, Abierto: true
```

Tipos de datos

boolean

```
let flag: boolean;
let yes = true;
let no = false;
```

number

```
let x: number;
let y = 0;
let z: number = 123.456;
let big: bigint = 100n;
```

string

```
let nombre: string = "Jassyr";
let presentacion: string = `Mi nombre es ${nombre}.
    Hola TypeScript!.`;
console.log(presentacion);
```

Tipos de datos

any

```
let randomValue: any = 10;
randomValue = 'Mateo';   // OK
randomValue = true;      // OK
```

unknown

```
let randomValue: unknown = 10;
randomValue = true;
randomValue = 'Mateo';

if (typeof randomValue === "string") {
  console.log((randomValue as string).toUpperCase());    // MATEO
} else {
  console.log("No es un string");    // Error
}
```

Tipos de datos

array

```
1   let lista: number[] = [1, 2, 3];
2   let arreglo: Array<number> = [1, 2, 3];
```

tuplas

```
1   let persona1: [string, number] = ['Maria', 35];
2   let persona2: [string, number] = ['Jose', 35, false];
```

```
Type '[string, number, boolean]' is not assignable to type '[string, number]'.
  Source has 3 element(s) but target allows only 2.
```

Interfaces

```
interface IEmpleado {
    nombre: string;
    apellido?: string; // opcional
    nombreCompleto(): string;

}

let empleado: IEmpleado = {
    nombre : "Juan",
    apellido: "Sánchez",
    nombreCompleto: (): string => {
        return `${empleado.nombre} ${empleado.apellido}`;
    }
}

empleado.edad = 10;
```

```
Property 'edad' does not exist on type 'IEmpleado'.
```

Tipos

```
type Empleado = {
  nombre: string;
    apellido: string;
    nombreCompleto(): string;
}

let empleado: Empleado = {
  nombre : "Juan",
    apellido: "Sánchez",
    nombreCompleto(){  return `${empleado.nombre} ${empleado.apellido}`; }
}
```

La principal diferencia es que un alias de tipos no se puede volver a abrir para agregar nuevas propiedades, mientras que una interfaz siempre es extensible.

Funciones

TypeScript simplifica el desarrollo de funciones y facilita la solución de problemas, ya que permite escribir parámetros y valores devueltos. TypeScript también agrega nuevas opciones para los parámetros. Por ejemplo, aunque todos los parámetros son opcionales en las funciones de JavaScript, puede optar por hacer que los parámetros sean necesarios u opcionales en TypeScript.

```
function sumar (x: number, y: number): number {
  return x + y;
}
console.log(sumar(1, 2)); // 3
```

Una expresión de función (o función anónima) es una función que no está cargada previamente en el contexto de ejecución y solo se ejecuta cuando el código la encuentra.

```
let suma = function (x: number, y: number): number {
  return x + y;
}
suma(1, 2);
```

Funciones flecha (Arrow functions 🏹)

Las funciones de flecha (también denominadas "expresión lambda" o "funciones de flecha Fat" debido al operador => usado para definirlas) proporcionan una sintaxis abreviada para definir una función anónima.

```
// Función anónima
let suma1 = function (x: number, y: number) {
  return x + y;
}
```

```
// Arrow function
let suma2 = (x: number, y: number) ⇒ x + y;
```

```
let suma = function (input: number[]): number {
  let total = 0;
  for (const n of input) {
    total += n
  }
  return total;
}
```

```
let suma2 = (input: number[]) ⇒
   input.reduce((a,b) ⇒ a+b, 0);
```

En algunos casos se puede reducir el cuerpo de la función a una sóla línea. Cuando esto ocurre se puede omitir el `return` y las llaves

Parametros opcionales

Typescript soporta parametros opciones con el signo de interrogación (`?`) al final del nombre del parámetro. Los parametros opcionales deben ir después de los obligatorios.

```
function sumar (x: number, y?: number): number {
  if (y === undefined) {
    return x;
    } else {
      return x + y;
    }
}

console.log(sumar(1, 2)); // 3
console.log(sumar(1));    // 1
```

Clases

Las clases de TypeScript amplían la funcionalidad de ES6 agregando características específicas de TypeScript como anotaciones de tipo para los miembros de clase, modificadores de acceso y la capacidad de especificar parámetros obligatorios u opcionales.

Tal como en otros lenguajes OO, las `clases` en Typescript definen datos y comportamientos de lo que serán los `objetos`

```
class Persona {
    nombre: string;
    edad: number;

    constructor(nombre: string, edad: number) {
        this.nombre = nombre;
        this.edad = edad;
    }

    saludar() {
        console.log(`Hola, me llamo ${this.nombre} y tengo ${this.edad} años.`);
    }
}
```

Componentes de clase

```
class Persona {
  private _nombre: string; // propiedades o campos, son los datos del objeto
  private edad: number;

  constructor(nombre: string, edad: number) { // función especial para crear e inizializa el objeto
      this._nombre = nombre;
      this.edad = edad;
  }

  saludar() { // metodos de la clase son funciones que definen el comportamiento del objeto
    return `Hola, me llamo ${this.nombre} y tengo ${this.edad} años.`;
  }

  get nombre() { return this._nombre } // descriptor de acceso get o set
  set nombre(nombre: string) { this._nombre = nombre }
}
```

Dependiendo del caso de uso, se podrían usar algunas de las características antes mencionadas. **No** todos los elementos son necesarios.

Constructor

```
1   class Vehiculo {
2     marca: string;
3
4     modelo: string;
```

```
Property 'modelo' has no initializer and is not definitely assigned in the constructor.
```

```
5
6     // sólo un constructor por clase
7     constructor(marca: string) {
8       this.marca = marca;
9     }
10  }
```

Como las funciones, los constructores aceptan una lista de parametros que pueden ser obligatorio, opcionales o deconstrucciones. Para indicar un miembro de la clase se utiliza la palabra clave `this`.

Método de clase

Se puede definir cualquier función de TypeScript dentro de una clase y llamarla como un método en el objeto o desde otras funciones dentro de la clase. Estos miembros de la clase describen los comportamientos que las clases pueden realizar.

```
acelerar(velocidad: number): string {
    return `${this.ref()} esta acelerando a ${velocidad} kph.`
}
frenar(): string {
    return `${this.ref()} esta frenando.`
}
girar(direccion: 'izquierda' | 'derecha'): string {
    return `${this.ref()} gira a la ${direccion}`;
}

ref(): string {
    return this.modelo;
}
```

Los métodos no necesitan la palabra clave `function` cuando se declaran dentro de una clase.

Creación de instancias de una clase

Para crear una instancia, se usa la palabra clave `new`

```
class Vehiculo {
  marca: string;
  _modelo: string;
  constructor(marca: string, modelo: string) {
    this.marca = marca;
    this._modelo = modelo;
  }

// demás métodos
  get modelo() { return `El modelo es ${this._modelo}`}
}

let auto = new Vehiculo('Mazda', 'CX30');

console.log(auto._modelo);
console.log(auto.modelo);
```

```
CX30
El modelo es CX30
```

Por default las propiedades son de acceso público. También se pueden declarar como `private` o `protected`

NestJs

¿Qué es NestJs?

Un framework para la construcción de aplicaciones Node.js del lado del servidor. Contiene una serie de características que lo colocan como una referencia en la industria para desarrollar APIs y microservicios. Soporta TypeScript y sus arquitectura esta inspirada en Angular.

Facilita el desarrollo de aplicaiones escalables, fáciles de probar y mantener.

Instalación de NestJs y aplicaciones requeridas

NestJS

```
npm i -g @nestjs/cli
```

DynamoDB

```
docker run -d -p 8000:8000 amazon/dynamodb-local
npm install -g dynamodb-admin
```

Docker debe estar corriendo en la maquina local.

Posteriormente, en una nueva ventana, iniciar `dynamodb-admin` La interfaz de Dynamo estará disponible en la dirección: `http://localhost:8001/`

Creación de un nuevo proyecto con NestJS

```
nest new seminario
> npm
....
cd seminario
npm run start
```

Ahora en postman, crear una nueva coleccion y hacer una petición GET a `http://localhost:3000/`

Desarrollo de una API REST básica

Crear un CRUD

El CLI de NestJS ofrece la posibilidad de generar el esquelo de un CRUD mediante el siguiente comando:

```
nest g resource taller
```

Esto creará los siguientes elementos:

- Un módulo: `taller.module.ts`
- Un controlador: `taller.controller.ts` configurado para la ruta /taller
- Un servicio: `taller.service.ts` donde esta la logica de negocio
- Entidades y dtos: Para la transferencia de datos en las capas de la aplicación

Controladores

Los controladores son responsables por manejar las solicitudes y respuestas a los clientes.

Un controlador es una clase anotada con `@Controller()`. La anotación `@Controller` puede recibir un parametro string que definirá su ruta

```
import { Controller, Get, Post, Body, Patch, Param, Delete } from '@nestjs/common';

@Controller('taller')
export class TallerController {
  constructor(private readonly tallerService: TallerService) {}

  @Post()
  create(@Body() createTallerDto: CreateTallerDto) {
    return this.tallerService.create(createTallerDto);
  }

  // Otros métodos del controlador
}
```

El mecanismo de ruteo de NesJS determina que controlador recibirá las peticiones. Un controlador puede tener mas de una ruta, y cada ruta puede realizar diferentes acciones.

Servicios

Los servicios contienen la lógica de negocio. Son los elementos reusables dentro de las aplicaciones de NestJS

```
import { Injectable } from '@nestjs/common';
import { CreateTallerDto } from './dto/create-taller.dto';

@Injectable()
export class TallerService {
  create(createTallerDto: CreateTallerDto) {
    return 'This action adds a new taller';
  }

  findAll() {
    return `This action returns all taller`;
  }
  /// otros métodos del servicio
}
```

Modulos

Agrupan los componentes de manera lógica (por dominio). Un módulo puede contener uno o varios controladores y servicios. Los módulos pueden exportarse e importarse en otros módulos.

```
import { Module } from '@nestjs/common';
import { TallerService } from './taller.service';
import { TallerController } from './taller.controller';

@Module({
  controllers: [TallerController],
  providers: [TallerService],
})
export class TallerModule {}
```

Todos los archivos se crean dentro de la carpeta taller. De esta manera NestJS separa los conceptos de dominio en módulos.

Repositorios para acceder a la base de datos DynamoDB

Crear un módulo donde se ubique el código para acceder a DynamoDB

```
npm install @aws-sdk/lib-dynamodb
npm install @aws-sdk/client-dynamodb
npm install uuid
npm install @types/uuid --save-dev

nest g module database
cd .\src\database\
nest g service DynamoDB --flat
nest g service TallerRepository --flat
```

Definición de entidades

Definir las propiedades de la clase Taller y sus DTOs

```
export class Taller {
  pk: string;
  titulo: string;
  fecha: string;
  creditos: number;
}

export class CreateTallerDto {
  titulo: string;
  fecha: string;
  creditos: number;
}
```

Configurar el proyecto para usar DynamoDB

1. Actualizar el módulo DataBase

```
@Module({
  providers: [
    DynamoDbService,
    TallerRepositoryService,
    {
      provide: DynamoDBDocumentClient,
      useFactory: () => {
        const client = new DynamoDBClient({
          region: 'us-east-2',
          endpoint: 'http://localhost:8000',
        });
        return DynamoDBDocumentClient.from(client);
      },
    },
  ],
  exports: [TallerRepositoryService],
})
export class DatabaseModule {}
```

2. Implementar un servicio para comunicarse con DynamoDB

Descargar el archivo base DynamoDBService

Implementar los métodos de la clase `TallerRepository`

- `create(createTallerDto: CreateTallerDto): Promise<Taller>`
- `findAll(): Promise<Taller[]>`
- `findById(id: string): Promise<Taller>`
- `update(id: string, updateTallerDto: UpdateTallerDto)`
- `delete(id: string)`

Validación y manejo de errores

Uso de Pipes

Usando `class-validator`. NestJS ya proporciona mecanismos de validación. Ver Validación

```
npm install class-validator --save
npm install class-transformer --save
```

```
export class CreateTallerDto {
  @IsString()
  titulo: string;
  @IsDateString()
  fecha: string;
  @IsPositive()
  creditos: number;
}
```

El controlador debe especificar el `pipe` de validación

```
@UsePipes(new ValidationPipe({ transform: true }))
```

Extra

Implementar un CRUD para el registro de asistentes a un taller

- Crear el controller, servicio y entidades con `nest g resource asistente`
- Validar los datos con `class-validator`
  - `@IsEmail()`
- Actualizar el taller
  - Agregar cupo máximo
  - Agregar cupo actual
  - Devolver 404 cuando el cupo se ha llenado

Material adicional a investigar por cuenta propia

- Jest para pruebas unitarias (El cli de NestJS ya crea las plantillas de las pruebas)
- PinoLogger para no usar console.log y disponer de un servicio dedicado, compatible con servicios de observabilidad como Kibana
- Authenticación/Authorizacion con `Guards`

Q&A y Cierre

- Preguntas y respuestas
- Recursos adicionales:
  - Repositorios de ejemplo en GitHub
  - Documentación oficial y tutoriales

¡Gracias por asistir!

- Instructor: Jassyr Juárez Vázquez
- Contacto: linkedin.com/in/jassyrj

Powered by