Universidad Icesi

Node.js & TypeScript

Ingeniería de Software — Universidad Icesi

Universidad ICESI

¿Qué es Node.js?

Entorno de Ejecución

Permite ejecutar código JavaScript fuera del navegador, directamente en el servidor.

Motor V8 de Google

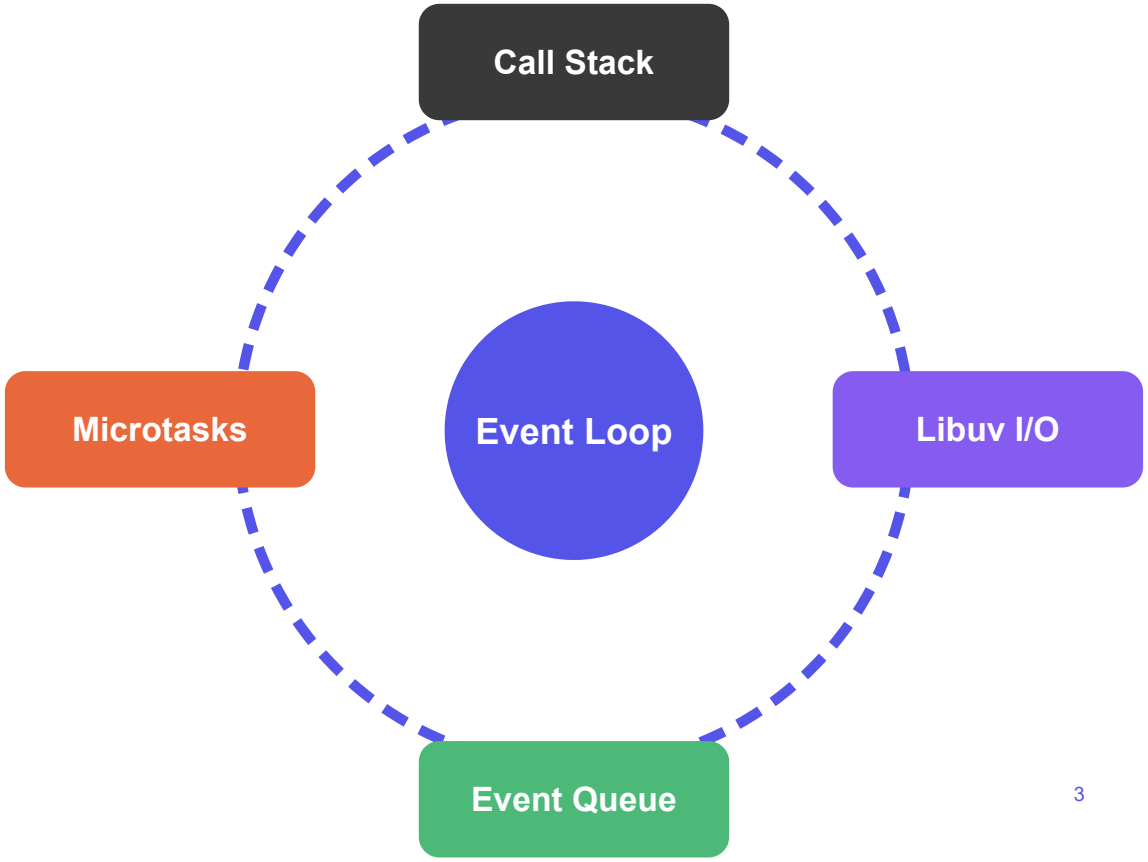
Compila JavaScript a código máquina nativo para ofrecer máxima velocidad.

Asincrónico y No Bloqueante

Arquitectura orientada a eventos, ideal para aplicaciones concurrentes y APIs.

El Event Loop de Node.js

1. **Single Thread:** Ejecuta el código JavaScript principal en un solo hilo.
2. **Delegación de I/O:** Las operaciones pesadas (disco, red) se delegan a Libuv.
3. **Event Queue:** Al finalizar el I/O, el resultado entra a la cola de eventos.
4. **Callback Execution:** El Event Loop toma los callbacks y los ejecuta cuando el Call Stack está libre.



Universidad ICESI

¿Por qué usar Node.js?

RENDIMIENTO

Procesa miles de conexiones concurrentes con un uso de memoria significativamente menor a servidores multihilo tradicionales.

UN SOLO LENGUAJE

Permite usar JavaScript o TypeScript tanto en el Front-End como en el Back-End, reduciendo el cambio de contexto mental.

ECOSISTEMA NPM

Acceso directo al registro de paquetes más grande del mundo con más de 2 millones de librerías listas para producción.

ADOPCIÓN GLOBAL

Estándar de la industria utilizado por gigantes tecnológicos como Netflix, Uber, LinkedIn, PayPal y NASA.

Navegador vs Node.js

JS en el Navegador

- Objeto global: `window`
- Acceso al DOM (`document` , HTML)
- APIs Web: `fetch` , `localStorage`
- Sandbox de seguridad estricto
- **Sin acceso al Sistema de Archivos**

Node.js (Servidor)

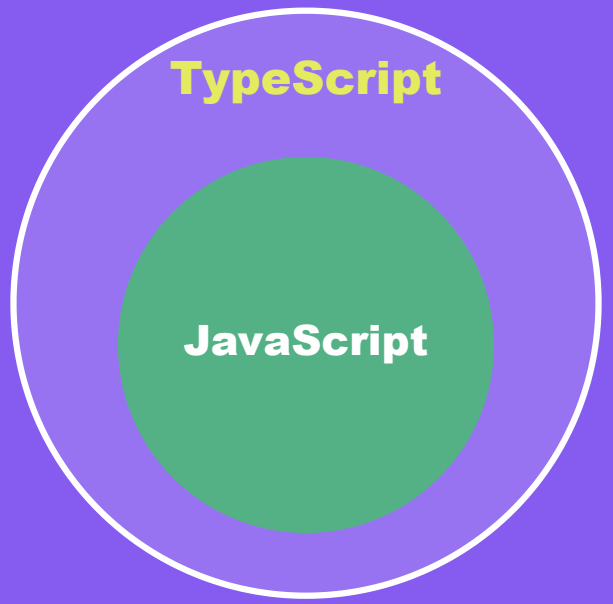
- Objeto global: `global`
- Sin DOM ni interfaz gráfica
- Módulos nativos: `fs` , `path` , `os` , `http`
- Acceso completo al Sistema Operativo
- **Puede leer y escribir archivos**

Universidad ICESI

TypeScript

Superset Estático de JavaScript

Universidad ICESI



¿Qué es TypeScript?

Superset de JavaScript

Todo código JS válido es código TS válido. Agrega tipado estático opcional.

Transpilación a JS

TS se transpila a JavaScript estándar para poder ejecutarse en Node.js o el browser.

Detección de Errores Temprana

Captura bugs en tiempo de compilación antes de enviar el código a producción.

Universidad ICESI

Ventajas de TypeScript

AUTOCOMPLETADO

Intellisense avanzado en IDEs como VS Code, ofreciendo autocompletado exacto de propiedades y métodos.

REFACTORING SEGURO

Permite renombrar funciones o reestructurar objetos en todo el proyecto con la certeza de que el compilador avisará si algo se rompe.

DOCUMENTACIÓN VIVA

Los tipos e interfaces funcionan como contratos explícitos que documentan el código sin necesidad de comentarios externos.

ECOSISTEMA ROBUSTO

Adoptado de forma masiva en frameworks modernos como NestJS, Angular, React y Next.js.

Instalación y Uso de TypeScript

1. Instalación Global vía npm

```
npm install -g typescript
```

2. Verificar Versión del Compilador (`tsc`)

```
tsc --version
```

3. Compilar un Archivo TypeScript a JavaScript

```
tsc app.ts            # Genera app.js
tsc app.ts --watch    # Modo observador (recompila automáticamente)
```

tsconfig.json — Configuración

Creación del Archivo

El archivo `tsconfig.json` indica que el directorio es la raíz de un proyecto TypeScript y especifica las opciones del compilador.

```
tsc --init
```

Genera un archivo con las mejores prácticas predeterminadas y explicaciones detalladas en comentarios.

Estructura Básica Recomendada

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "strict": true,
    "outDir": "./dist",
    "rootDir": "./src",
    "esModuleInterop": true
  }
}
```

tsconfig.json — Opciones Esenciales

Opción	Valor Típico	Descripción
`target`	`ES2020`	Versión de ECMAScript a la que se transpirará el código JS.
`module`	`commonjs`	Sistema de módulos a utilizar en el JS generado (CommonJS para Node.js).
`strict`	`true`	Activa todas las comprobaciones estrictas de tipos (recomendado).
`outDir`	`./dist`	Carpeta donde se colocarán los archivos `.js` compilados.
`rootDir`	`./src`	Carpeta raíz donde residen los archivos fuente `.ts`.

Universidad ICESI

Tipos en TypeScript

Tipos Primitivos en TypeScript

```ts
// String
let username: string = 'Operator007';
username.toLowerCase();
console.log(username);

// Number
let age: number = 10;
age.toFixed();
console.log(age);

// Boolean
let isLoggedIn: boolean = false;
console.log(isLoggedIn.valueOf());
```

""

string:

Cadenas de texto. Admite comillas simples, dobles y template literals.

#

number:

Valores numéricos (enteros y flotantes). En JS todos los números son de 64 bits.

0/1

boolean:

Valores lógicos (`true` o `false`).

Inferencia de Tipos (Type Inference)

TypeScript es lo suficientemente inteligente como para deducir el tipo de una variable en función del valor inicial asignado.

```ts
/* Type inference */
let greeting = 'Hello World';

// TypeScript infiere que 'greeting' es de tipo string
greeting.toLowerCase();
console.log(greeting);
```

Detección de Errores

Si intentamos asignar un tipo diferente más adelante, el compilador generará un error de forma inmediata:

```ts
greeting = 1;

// Error de compilación:
// Type 'number' is not assignable
// to type 'string'.
```

Tipo any — El Anti-Patrón

Evitar: any Implícito

Desactiva la verificación de tipos y pierde la protección del compilador.

```
let address; // Tipo 'any' implícito

function getAddress() {
    return "Cll 123 # 45";
}

address = getAddress(); // Pierde chequeo
```

Recomendado: Tipo Explícito

Conserva autocompletado, refactorización segura y contratos claros.

```
let address1: string;

function getAddress1(): string {
    return "Cll 123 # 45";
}

address1 = getAddress1(); // Seguro y tipado
```

Universidad ICESI

Funciones Tipadas

```
// Función tradicional con retorno explícito
function addTwo(a: number): number {
    return a + 2;
}

// Arrow function tipada
const concat = (start: string, end: string): string => {
    return start + end;
};

addTwo(10);
concat('Hello', 'World');
```

```
// Parámetros opcionales / valores por defecto
const registerAnswer = (
  id: number,
  message: string = "No message"
): string => {
    return `${id} - ${message}`;
};

// Inferencia de tipos en callbacks
const userTypes = ["admin", "editor", "subscriber"];

userTypes.map((userType): string => {
    return userType.toUpperCase();
});
```

Objetos con Type Aliases

```ts
/* Type alias para un objeto */
type UserType = {
    id?: number; // propiedad opcional (?)
    name: string;
    lastname: string;
    isActive: boolean;
}

const user: UserType = {
    id: 1,
    name: "Dexter",
    lastname: "Morgan",
    isActive: true,
};
```

```ts
// Uso de Type Aliases en parámetros de función
function createUser(
 {name, lastname, isActive}: UserType
): UserType {
    return {
        id: Math.floor(Math.random() * 1000),
        name,
        lastname,
        isActive
    };
}

createUser({
  name: "Kevin",
  lastname: "David",
  isActive: true
});
```

Universidad ICESI

Clases y Objetos en TypeScript

```typescript
class Person {
    name: string;
    age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    greet(): string {
        return `Hello, my name is ${this.name}`;
    }
}

const person1 = new Person("Dexter", 35);
console.log(person1.greet());
```

```typescript
// Shorthand con modificadores de acceso
class UserAccount {
    constructor(
    public id: number,
    public username: string,
    private email: string
  ) {}

    getEmail(): string {
        return this.email;
    }
}

const account = new UserAccount(1, "dexter", "d@g.com");
// account.email; // Error: es privado
```

Union & Intersection Types

Unión de Tipos (`|`)

```ts
type activeStatus = 'active' | 'inactive' | 'pending';

function getId(id: number | string): string {
    if (typeof id === 'string') {
        return id.toLowerCase();
    }
    return `${id + 2}`;
}
```

Intersección (`&`) y `readonly`

```ts
type point = { x: number; y: number; };
type point3D = point & { z: number; };

type User = {
    readonly id: number; // No modificable
    name: string;
    isActive: activeStatus;
};
```

Arrays Tipados

TypeScript permite restringir los elementos de un arreglo a tipos específicos.

```
type fruits = "apple" | "banana" | "orange";
type vegetables = string[];

// Unión de tipos en arrays
type groceries = Array | vegetables;

const myGroceries: groceries = [
    "apple",
    "banana",
    "orange",
    "carrot"
];
```

Operaciones sobre Arrays

```
function getElement(
  arr: (number | string)[]
): (number | string)[] {
    return arr.filter(
      item => typeof item === 'number'
    );
}

const userTypes = ["admin", "editor"];
userTypes.push("subscriber"); // OK
// userTypes.push(123); // Error
```

Enums (Enumeraciones)

Permiten definir un conjunto de constantes con nombre, aumentando la legibilidad y evitando números o strings mágicos.

```
enum ProceedingStatus {
    RESERVED = 1,
    ACTIVE = 2,
    INACTIVE, // Toma automáticamente 3
    PENDING = 4
}

ProceedingStatus.RESERVED; // Retorna 1
ProceedingStatus.ACTIVE;   // Retorna 2
const status = ProceedingStatus.RESERVED;
```

Const Enums (Optimización)

Los `const enum` se eliminan completamente durante la compilación y se reemplazan in-line por sus valores constantes.

```
const enum UserRole {
    ADMIN = 'ADMIN',
    USER = 'USER'
}

const role = UserRole.ADMIN;
// Se transpila directamente a:
// var role = "ADMIN";
```

Type vs Interface — Sintaxis

Interface

Ideal para definir contratos de objetos y clases.

```
interface IShape {
    name: string;
    callIt(): string;
}
interface Admin extends IShape {
    role: string;
}
```

Type Alias

Ideal para uniones, tuplas y tipos primitivos.

```
type ShapeType = {
    name: string;
    callIt(): string;
};
type AdminType = ShapeType & {
    role: string;
};
```

Type vs Interface — ¿Cuándo usar cada uno?

Característica	Interface	Type Alias
Definición de Objetos/Clases	**Ideal (estándar en OOP)**	Soportado
Uniones / Primitivos	No soportado	**Ideal (`'A' | 'B'`)**
Extensión / Herencia	Sintaxis `extends`	Intersección `&`
Declaration Merging	Sí (permite fusionar tipos)	No

Regla General: Usa `interface` para definir contratos de APIs, modelos y clases. Usa `type` para uniones, tuplas y tipos compuestos.

Genéricos (Generics)

Permiten crear componentes y funciones reutilizables que funcionan con una variedad de tipos en lugar de uno solo, manteniendo la seguridad de tipos.

```
function identityAny(arg: any): any {
    return arg; // Pierde el tipo de retorno
}

function identityGeneric(arg: Type): Type {
    return arg; // Preserva el tipo exacto
}

identityGeneric("Hello, World!");
identityGeneric(42);
```

Genéricos en Arrow Functions

```
const searchElements = (
  array: T[],
  searchElement: T
): boolean => {
    return array.includes(searchElement);
};

searchElements([1, 2, 3], 1); // true
searchElements(['a', 'b'], 'a'); // true
```

Universidad ICESI

Entorno de Desarrollo Moderno

De ts-node & nodemon a tsx

Universidad ICESI

Forma Antigua:

`ts-node + nodemon (Lento)`

Estándar Moderno:

`tsx / node --watch`

¿Por qué tsx?

Rendimiento Ultra Rápido

Potenciado por esbuild. Ejecuta y transpila TypeScript en milisegundos.

Sustituto de ts-node + nodemon

Se integra de forma nativa con el flag `--watch` introducido en Node.js reciente.

Soporte ESM & CJS Nativo

Maneja archivos ECMAScript Modules y CommonJS sin configuraciones complejas.

Configuración del Proyecto con tsx

1. Inicializar e Instalar

```
npm init -y
npm install -D typescript tsx @types/node
```

2. Compilar para Producción

```
npx tsc
```

3. Scripts en package.json

```
{
  "name": "my-express-app",
  "scripts": {
    "dev": "tsx watch src/index.ts",
    "build": "tsc",
    "start": "node dist/index.js"
  }
}
```

Universidad ICESI

Principios SOLID en TS / Node.js

Universidad ICESI

Principios SOLID

S — SRP

Single Responsibility: Una clase o módulo debe tener una sola razón para cambiar.

O — OCP

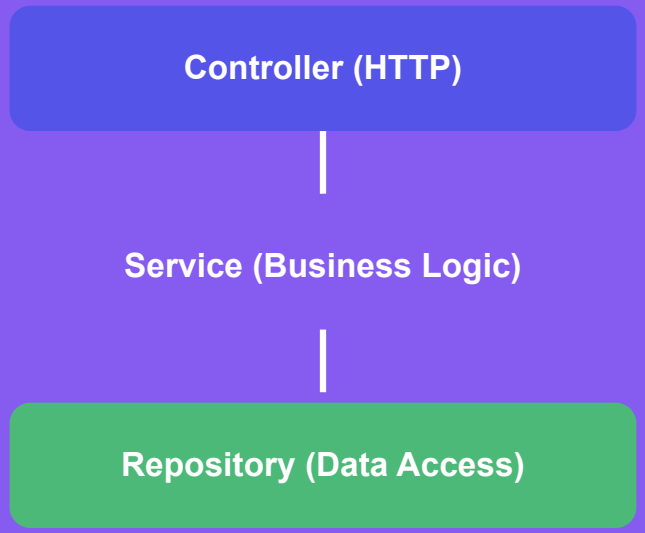
Open/Closed: Entidades abiertas para extensión, pero cerradas para modificación.

L — LSP

Liskov Substitution: Los objetos de una subclase deben poder reemplazar a los de la superclase.

I / D — ISP & DIP

Interface Segregation & Dependency Inversion: Depender de abstracciones (interfaces), no de implementaciones concretas.



SOLID en la Práctica

Interfaces como Contratos

TypeScript nos da la capacidad de definir abstracciones puras para desacoplar componentes.

Inyección de Dependencias

Inyectar repositorios y servicios en controladores facilita las pruebas unitarias con Mocks.

Universidad ICESI

ExpressJS

Framework Web Minimalista para Node.js

Universidad ICESI

express

Fast, unopinionated, minimalist

¿Qué es Express.js?

Framework Minimalista

Ligero y sin opiniones estrictas. Otorga libertad total sobre la estructura del proyecto.

Routing de Métodos HTTP

Gestión intuitiva de rutas (GET, POST, PUT, DELETE, PATCH).

Middlewares

Funciones que interceptan las peticiones para autenticación, logs y validaciones.

Hola Mundo: Express + TypeScript

```typescript
import express, { Request, Response } from 'express';

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware para parsear JSON en el body
app.use(express.json());

// Ruta principal
app.get('/', (req: Request, res: Response) => {
    res.json({ message: "¡Hola Mundo desde Express & TypeScript!" });
});

app.listen(PORT, () => {
    console.log(`Servidor corriendo en http://localhost:${PORT}`);
});
```

Universidad ICESI

Express vs Spring Boot

Express.js (Node.js)

- **Filosofía:** Unopinionated (Libertad total).
- **Arranque:** Instantáneo (<100ms).
- **Memoria RAM:** Muy bajo consumo (~30-50MB).
- **Lenguaje:** JavaScript / TypeScript.
- **Inyección de Dep.:** Manual o librerías livianas.

Spring Boot (Java)

- **Filosofía:** Opinionated (Convención sobre config).
- **Arranque:** Más lento (segundos por carga de JVM/Beans).
- **Memoria RAM:** Mayor consumo (~250-500MB+).
- **Lenguaje:** Java / Kotlin.
- **Inyección de Dep.:** IoC Container robusto (@Autowired).

Ventajas de Express frente a Spring

DESARROLLO ÁGIL

Menos código boilerplate para iniciar un servidor HTTP o prototipar un microservicio.

FLEXIBILIDAD

Tú eliges cómo organizar las carpetas, la base de datos y los patrones arquitectónicos.

ARQUITECTURA SERVERLESS

Su tiempo de arranque ultra rápido lo hace ideal para funciones AWS Lambda / Serverless.

LENGUAJE UNIFICADO

Permite compartir modelos, DTOs y validaciones entre el servidor y clientes web/móviles.

Estructura de Proyecto en Express

```
mi-proyecto-express/
├── .env
├── .gitignore
├── package.json
├── README.md
├── tsconfig.json
└── src/
    ├── index.ts
    ├── core/
    │   └── domain/
    └── features/
        ├── auth/
        └── users/
```

Propuesta Arquitectónica

Debido a que Express no impone una estructura, se propone una organización por **Features (Vertical Slicing)** combinada con principios DDD / Clean Architecture.

- `src/index.ts`: Punto de entrada del servidor.
- `src/core/domain/`: Entidades globales y contratos.
- `src/features/`: Módulos independientes por dominio de negocio.

Estructura Interna de un Feature

```
src/features/users/
├── users.router.ts
├── users.controller.ts
├── users.service.ts
├── users.repository.ts
└── users.dto.ts
```

router.ts: Rutas del endpoint (`GET /users`).

controller.ts: Maneja `req` y `res` (HTTP).

service.ts: Lógica de negocio pura.

repository.ts: Acceso a la base de datos.

dto.ts: Data Transfer Objects & tipos.

Universidad ICESI

¡Gracias! ¿Preguntas?