Archivos de Definición

Antonio Espín Herranz

Introducción

- Los archivos de definición son archivos con extensión **.d.ts** que proporcionan tipos para bibliotecas o APIs de JavaScript.

- Su función principal es permitir que **TypeScript entienda las estructuras** de código de JavaScript que no están escritos originalmente en TypeScript.

- También podemos crear nuestros propios archivos de definición: **.d.ts**

Archivos de definición

- Pueden ser propios
- Pueden venir de librerías de JS que ya están desarrollados.
  - Básicamente lo que tenemos que instalar es un paquete @types

- O se pueden generar a partir de código JS
  - Es conveniente que el código JS esté documentado, si no, s que al generar el archivo **d.ts** los tipos se identifiquen como **any**

Archivos de definición propios

- **PASOS**:
  - Crear el archivo, tiene que tener extensión **.d.ts**
  - Declarar módulos o variables
    - Si estamos creando un módulo → declare module
    - Si creamos una función o constante → declare
  - Definir los tipos:
    - Definimos las funciones, clases, contantes, etc. con los tipos de cada uno.
  - Integrar en el proyecto
    - El archivo d.ts tiene que estar incluido en el proyecto.
    - Las rutas tienen que estar incluidas en el fichero: tsconfig.json

Generar ficheros d.ts a partir de JS

- Estos ficheros se pueden generar de forma automática:
- Necesitamos una carpeta para el proyecto:
  - Fichero.js
    - **Dentro del fichero de JS tenemos que exportar las funciones o las clases.**
  - Crear una carpeta **dist**
  - Crear el fichero **tsconfig.json → tsc --init**
  - Dentro del fichero de configuración tenemos:
    - "**allowJs**": **true** permite que TypeScript procese archivos .js.
    - "**declaration**": **true** genera el archivo .d.ts.
    - "**emitDeclarationOnly**": **true** asegura que solo se generen archivos de definición (sin generar .js transpilados).
    - "**outDir**": "./**dist**" define el directorio donde se guardarán los archivos.
  - Ejecutar el comando: **tsc**

Ejemplo

- Si tenemos una función de JS:

```
// miScript.js
function suma(a, b) {
   return a+ b;
}
module.export = {suma}
```

- Se puede definir el tipo:
  - // miScript.d.ts
  - declare function suma(a: any, b:any): any;

Indicar tipos en la documentación de JS

```
/**
 *
 * @param {number} a
 * @param {number} b
 * @returns number
 */
function suma2(a, b) {
  return a + b;
}

module.exports = { suma2 };
```

- Al aplicar el comando **tsc** → generar el fichero**: funciones.d.ts**

.d.ts generado

```
/**
 *
 * @param {number} a
 * @param {number} b
 * @returns number
 */
export function suma2(a: number, b: number): number;
```

Ventajas de utilizar d.ts

- Detectar errores al principio
- Mejor autocompletado en el código
- Documentación implícita, porque ya se indica el tipo de los parámetros y lo que devuelve la función
- Compatibilidad con bibliotecas externas.
  - Bibliotecas populares de JS suelen tener definición de tipos en un paquete llamado: @types
    - Estos paquetes se pueden instalar con npm
    - **npm install @types/jquery**
    - **npm install @types/express**
    - **npm install lodash @types/lodash**
- Facilita la migración del código JS a TS

Ejemplo con lodash

- En una carpeta instalar:
  - npm install lodash @types/lodash

  - Se instala en local.

```
import lib from "lodash"

const numeros = [1,2,3,4,5]
const suma = lib.sum(numeros) // 1 + 2 + 3 + 4 + 5 = 15

//const vacio = lib.sum();// Este falla

console.log(` La suma de los números es: ${suma}` ) // La suma de los números es: 15
```