Desarrollo de Sistemas Informáticos 2024-2025

Apuntes sobre TypeScript y algunas cosas más

View project on GitHub

Creación de un proyecto inicial para trabajar con TypeScript

Esta sección ilustra la creación de un proyecto inicial mínimo con el que poder empezar a trabajar con TypeScript. También se describe la configuración de algunas herramientas que podrían ser de utilidad a la hora de desarrollar.

Gestión de dependencias y configuración del compilador de TypeScript

Lo primero que haremos es abrir una terminal, ya sea a través del sistema operativo o de VSCode. En la misma crearemos un directorio con el nombre de nuestro proyecto:

```
[~/DSI()]$mkdir theory-examples
[~/DSI()]$cd theory-examples/
[~/DSI/theory-examples()]$
```

A continuación, ejecutaremos el siguiente comando para generar un fichero `package.json` en la raíz de nuestro proyecto, el cual va a permitir que gestionemos las dependencias de desarrollo y ejecución del mismo, además de otras características:

```
[~/DSI/theory-examples()]$npm init --yes
Wrote to /home/usuario/DSI/theory-examples/package.json:

{
  "name": "theory-examples",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
```

```
  "description": ""
}
```

Seguidamente, instalaremos el compilador de TypeScript:

```
[~/DSI/theory-examples()]$npm i --global typescript

added 1 package in 2s
[~/DSI/theory-examples()]$tsc --version
Version 5.7.3
```

La opción `--global` permite instalar un paquete de manera global. Además, crearemos un fichero denominado `tsconfig.json` en la raíz de nuestro proyecto, el cual va a contener las opciones de configuración de dicho compilador. Lo editaremos incluyendo el siguiente contenido:

```
[~/DSI/theory-examples()]$cat tsconfig.json
{
  "compilerOptions": {
    "target": "ES2024",
    "module": "commonjs",
    "rootDir": "./src",
    "outDir": "./dist"
  }
}
```

Básicamente, estas opciones de configuración le indican al compilador de TypeScript que, en primer lugar, se genere código compatible con una de las últimas versiones del estándar de JavaScript, esto es, el ECMAScript. En segundo lugar, se indica que el código JavaScript generado se basará en la especificación de módulos CommonJS. Esta es una especificación ampliamente utilizada cuando la aplicación se va a ejecutar sobre un entorno de ejecución como Node.js, que es el caso que nos ocupa. En tercer lugar, se especifica que el código fuente escrito en TypeScript se encuentra en el directorio `src`. Por último, que el código JavaScript producto de la compilación se almacenará en el directorio `dist`. Debido a esto último, debemos crear los directorios `src` y `dist` en la raíz de nuestro proyecto, aunque el directorio `dist` también se crearía automáticamente con la primera compilación de nuestro código:

```
[~/DSI/theory-examples()]$mkdir src
[~/DSI/theory-examples()]$mkdir dist
```

Otra opción a la hora de crear el fichero `tsconfig.json` sería ejecutar el siguiente comando para generarlo automáticamente y, posteriormente, proceder a editarlo para que se parezca a lo indicado más arriba:

```
[~/DSI/theory-examples()]$tsc --init
```

```
Created a new tsconfig.json with:

  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true


You can learn more at https://aka.ms/tsconfig
```

Compilación y ejecución automática tras detección de cambios

El siguiente paso consistirá en instalar un paquete que permitirá detectar los cambios realizados sobre los ficheros con código fuente en TypeScript (aquellos almacenados en el directorio `src`), recompilarlos para generar el código JavaScript correspondiente y ejecutarlo, todo ello de manera automática:

```
[~/DSI/theory-examples()]$npm install --save-dev tsc-watch

added 19 packages, and audited 20 packages in 3s

found 0 vulnerabilities
```

Para hacer uso de la funcionalidad de dicho paquete, debemos modificar la propiedad `"scripts"` del fichero `package.json`, eliminando la línea correspondiente a la propiedad `"test"` y añadiendo una propiedad que denominaremos `"start"`. El contenido del fichero después de ser modificado debe ser el siguiente:

```
[~/DSI/theory-examples()]$cat package.json
{
  "name": "theory-examples",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "tsc-watch --onSuccess \"node dist/index.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "devDependencies": {
    "tsc-watch": "^6.2.1"
  }
}
```

Tal y como puede observarse, solo se ejecutará el código JavaScript generado y almacenado en el fichero `dist/index.js` en caso de que el proceso de compilación a través del compilador de TypeScript haya sido satisfactorio, es decir, libre de errores.

Para que lo anterior funcione correctamente, debemos tener algún fichero con código escrito en TypeScript en nuestro proyecto. Dado que el código JavaScript generado se encontrará en el fichero `index.js` del directorio `dist`, tendremos que crear un fichero con código TypeScript denominado `index.ts` en el directorio `src` de nuestro proyecto:

```
[~/DSI/theory-examples()]$cd src/
[~/DSI/theory-examples/src()]$cat index.ts
const myString: string = "Hola Mundo";
console.log(myString);
```

Ahora, ejecutaremos el siguiente comando, el cual ejecutará el script definido en la propiedad `"start"` del fichero `package.json`:

```
[~/DSI/theory-examples/src()]$npm start

...

11:45:36 - Starting compilation in watch mode...


11:45:38 - Found 0 errors. Watching for file changes.
Hola Mundo
```

Cada vez que modifiquemos algo en el directorio `src` de nuestro proyecto, se recompilarán todos los ficheros con código TypeScript almacenados en dicho directorio y se ejecutará el fichero `index.js` generado de manera automática en el directorio `dist`.

En este punto, valdría la pena echar un vistazo a las diferencias entre los ficheros `src/index.ts` y `dist/index.js`:

```
[~/DSI/theory-examples()]$diff src/index.ts dist/index.js
1c1
< const myString: string = "Hola Mundo";
---
> const myString = "Hola Mundo";
```

Como puede observarse, la principal diferencia se encuentra en la declaración de la constante `myString`. Una de las principales funcionalidades de TypeScript es que utiliza tipado estático para tratar de evitar los problemas que surgen con JavaScript, el cual se basa en tipado dinámico.

Comprobación y resolución de errores en la lógica del código

Si queremos comprobar de manera estática nuestro código fuente en busca de posibles errores para poder solucionarlos, tendremos que utilizar un linter. El más conocido para trabajar con JavaScript y, por lo tanto, con TypeScript, es ESLint. Para empezar a utilizarlo, pueden instalar el paquete `eslint` de manera global haciendo uso de `npm`:

```
[~/DSI/theory-examples()]$npm i -g eslint

added 85 packages in 5s

22 packages are looking for funding
  run `npm fund` for details
[~/DSI/theory-examples()]$eslint --version
v9.18.0
```

A continuación, debemos crear un fichero de configuración de ESLint en base a una serie de preguntas que nos va a ir realizando el script de inicialización del comando `eslint`. Será importante indicar que nuestro proyecto va a hacer uso de TypeScript como lenguaje, además de indicar que nuestro entorno de ejecución va a ser Node.js y de instalar las dependencias sugeridas por el script de inicialización:

```
[~/DSI/theory-examples()]$eslint --init
You can also run this command directly using 'npm init @eslint/config@latest'.

> theory-examples@1.0.0 npx
> create-config

@eslint/create-config: v1.4.0

✔ How would you like to use ESLint? · problems
✔ What type of modules does your project use? · esm
✔ Which framework does your project use? · none
✔ Does your project use TypeScript? · typescript
✔ Where does your code run? · node
The config that you've selected requires the following dependencies:

eslint, globals, @eslint/js, typescript-eslint
✔ Would you like to install them now? · No / Yes
✔ Which package manager do you want to use? · npm
Installing...

up to date, audited 130 packages in 1s

36 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
Successfully created /home/usuario/DSI/theory-examples/eslint.config.mjs file.
```

El fichero de configuración de ESLint se encontrará en la raíz de nuestro proyecto con el nombre `eslint.config.mjs`:

```
import globals from "globals";
import pluginJs from "@eslint/js";
import tseslint from "typescript-eslint";


/** @type {import('eslint').Linter.Config[]} */
export default [
  {files: ["**/*.{js,mjs,cjs,ts}"]},
  {languageOptions: { globals: globals.node }},
  pluginJs.configs.recommended,
  ...tseslint.configs.recommended,
];
```

Se puede observar como, al haber indicado que nuestro proyecto usa TypeScript e instalado las dependencias necesarias, automáticamente, se ha añadido el uso de un plugin denominado typescript-eslint. Según sus autores, se trata de la herramienta que permite a ESLint y Prettier dar soporte a TypeScript. No lo hemos indicado aún, pero Prettier es un formateador de código, esto es, una herramienta que permite comprobar y corregir problemas relacionados con el formateo de nuestro código como, por ejemplo, con su identación. Un linter también suele incorporar reglas que permiten detectar problemas de formateo del código, pero solo se recomienda su uso para comprobar y corregir problemas relacionados con la lógica del código como, por ejemplo, que no se declare una interfaz vacía. Debido a lo anterior, se recomienda utilizar ambos tipos de herramientas en conjunto.

ESLint y, por lo tanto, typescript-eslint, trabajan con reglas. Una regla permite detectar y corregir un error concreto. En nuestro fichero de configuración, al haber ejecutado el script de inicialización, se han habilitado las reglas recomendadas por el equipo de ESLint, así como las reglas recomendadas por el equipo de typescript-eslint. En este último caso, también se han desactivado las reglas recomendadas por el equipo de ESLint que entrarían en conflicto con las de typescript-eslint, en el caso de estar habilitadas.

Por ejemplo, modifiquemos el fichero `index.ts` del directorio `src` para que la variable `myString` no se utilice en el código:

```
const myString: string = "Hola Mundo";
console.log("Hola Mundo");
```

Al ejecutar el linter para comprobar los errores se obtiene lo siguiente por la consola:

```
[~/DSI/theory-examples()]$eslint .

/home/usuario/DSI/theory-examples/src/index.ts
  1:7  error  'myString' is assigned a value but never used  @typescript-eslint/no-unused-vars

✖ 1 problem (1 error, 0 warnings)
```

Se puede observar como ESLint ha informado de que existe un error en nuestro código, en concreto, porque no se ha cumplido la regla `@typescript-eslint/no-unused-vars`, la cual es una de las que se ha habilitado gracias a haber utilizado el conjunto de reglas recomendado por typescript-eslint. En concreto, dicha regla se incumple cuando se declaran variables que luego no son utilizadas en el resto del código, cosa que ocurre con la variable `myString`. Además de lo anterior, si se ha instalado la extensión de ESLint en VSCode, el propio entorno de desarrollo será el que nos informe de dicho error, sin necesidad de ejecutar el linter en la consola.

A pesar de lo anterior, podríamos querer desactivar una regla concreta de las incluidas en la configuración recomendada de typescript-eslint, para lo cual tendremos que editar el fichero de configuración `eslint.config.mjs`:

```
import globals from "globals";
import pluginJs from "@eslint/js";
import tseslint from "typescript-eslint";


/** @type {import('eslint').Linter.Config[]} */
export default [
  {files: ["**/*.{js,mjs,cjs,ts}"]},
  {languageOptions: { globals: globals.node }},
  pluginJs.configs.recommended,
  ...tseslint.configs.recommended,
  {rules: {
    "@typescript-eslint/no-unused-vars": "off"
  }},
];
```

Tal y como puede observarse, hemos deshabilitado la regla `@typescript-eslint/no-unused-vars`, lo que hace que ya no se emita el error, ni por parte del entorno de desarrollo (en el caso de tener instalada la extensión de ESLint), ni en el caso de ejecutar el linter desde la consola.

Otra opción podría ser que el incumplimiento de la regla se informe como una advertencia, en lugar de como un error (nivel de severidad de la regla):

```
import globals from "globals";
import pluginJs from "@eslint/js";
import tseslint from "typescript-eslint";


/** @type {import('eslint').Linter.Config[]} */
export default [
  {files: ["**/*.{js,mjs,cjs,ts}"]},
  {languageOptions: { globals: globals.node }},
  pluginJs.configs.recommended,
  ...tseslint.configs.recommended,
  {rules: {
    "@typescript-eslint/no-unused-vars": "warn"
```

```
  }},
];
```

En este caso, si se ejecuta el linter desde la consola, se debería obtener algo como lo siguiente:

```
[~/DSI/theory-examples()]$eslint .

/home/usuario/DSI/theory-examples/src/index.ts
  1:7  warning  'myString' is assigned a value but never used  @typescript-eslint/no-unused-vars

✖ 1 problem (0 errors, 1 warning)
```

Por último, supongamos que queremos habilitar una regla que no se encuentra incluida entre las recomendadas por typescript-eslint como, por ejemplo, la regla `@typescript-eslint/init-declarations`, la cual permite establecer que se requiera inicializar una variable en la misma sentencia donde se declara. El fichero de configuración debe ser algo similar a lo que sigue:

```
import globals from "globals";
import pluginJs from "@eslint/js";
import tseslint from "typescript-eslint";


/** @type {import('eslint').Linter.Config[]} */
export default [
  {files: ["**/*.{js,mjs,cjs,ts}"]},
  {languageOptions: { globals: globals.node }},
  pluginJs.configs.recommended,
  ...tseslint.configs.recommended,
  {rules: {
    "init-declarations": "off",
    "@typescript-eslint/init-declarations": "error"
  }},
];
```

En este caso, se puede observar como se ha habilitado la regla de typescript-eslint `@typescript-eslint/init-declarations` con nivel de severidad `"error"`, además de haber deshabilitado la regla `init-declarations` de ESLint. Lo anterior se debe a que la regla `@typescript-eslint/init-declarations` es una regla de extensión, es decir, una regla de typescript-eslint que tiene la misma funcionalidad que la proporcionada por ESLint, pero que también funciona con TypeScript. Para que las reglas de ESLint y typescript-eslint no entren en conflicto, antes de habilitar la de typescript-eslint, es una buena práctica deshabilitar la de ESLint.

Si modificamos el código fuente del fichero `src/index.ts` a lo siguiente:

```
let myString: string;
console.log(myString);
```

Ejecutando el linter desde la consola deberíamos obtener lo siguiente:

```
[~/DSI/theory-examples()]$eslint .

/home/usuario/DSI/theory-examples/src/index.ts
  1:5  error  Variable 'myString' should be initialized on declaration  @typescript-eslint/init-decl

✖ 1 problem (1 error, 0 warnings)
```

Supongamos que queremos que ESLint no lleve a cabo comprobaciones sobre ciertos ficheros o directorios, por ejemplo, sobre aquellos ficheros JavaScript que el compilador de TypeScript ha generado en el directorio `dist` de nuestro proyecto a partir de nuestro código fuente escrito en TypeScript. Lo anterior puede especificarse a través del fichero de configuración `eslint.config.mjs`:

```
import globals from "globals";
import pluginJs from "@eslint/js";
import tseslint from "typescript-eslint";


/** @type {import('eslint').Linter.Config[]} */
export default [
  {files: ["**/*.{js,mjs,cjs,ts}"]},
  {languageOptions: { globals: globals.node }},
  pluginJs.configs.recommended,
  ...tseslint.configs.recommended,
  {rules: {
    "init-declarations": "off",
    "@typescript-eslint/init-declarations": "error"
  }},
  {ignores: [
    "dist/*"
  ]
  }
];
```

Se puede observar como se ha indicado, a través de la propiedad `ignores`, que se ignore el contenido del directorio `dist`. Dicha propiedad `ignores` espera una lista, por lo que se pueden indicar más directorios o ficheros concretos separándolos con comas para que sean ignorados por ESLint.

Por último, volvamos a dejar el fichero de configuración de ESLint con los conjuntos de reglas recomendados:

```
import globals from "globals";
import pluginJs from "@eslint/js";
import tseslint from "typescript-eslint";
```

```
/** @type {import('eslint').Linter.Config[]} */
export default [
  {files: ["**/*.{js,mjs,cjs,ts}"]},
  {languageOptions: { globals: globals.node }},
  pluginJs.configs.recommended,
  ...tseslint.configs.recommended,
  {ignores: [
    "dist/*"
  ]
  }
];
```

Y el fichero `src/index.ts` con el siguiente contenido:

```
const myString: string = "Hola Mundo";
console.log(myString);
```

Formateo del código

El formateador de código que vamos a utilizar es Prettier. Lo primero que haremos es instalar el paquete `prettier`, así como el paquete `eslint-config-prettier`. En concreto, este último permite desactivar todas las reglas de ESLint relacionadas con el formateo del código, con el objetivo de que ESLint y Prettier puedan coexistir sin problemas.

```
[~/DSI/theory-examples()]$npm i --save-dev prettier eslint-config-prettier

added 2 packages, and audited 132 packages in 2s

37 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

A continuación, para desactivar todas las reglas de formateo de ESLint y typescript-eslint que no son necesarias o que podrían entrar en conflicto con las de Prettier, debemos modificar el fichero de configuración `eslint.config.mjs` para importar el módulo `eslint-config-prettier`, el cual acabamos de instalar anteriormente y utilizarlo en la configuración de ESLint:

```
import globals from "globals";
import pluginJs from "@eslint/js";
import tseslint from "typescript-eslint";
import eslintConfigPrettier from "eslint-config-prettier";

/** @type {import('eslint').Linter.Config[]} */
export default [
  {files: ["**/*.{js,mjs,cjs,ts}"]},
```

```
  {languageOptions: { globals: globals.node }},
  pluginJs.configs.recommended,
  ...tseslint.configs.recommended,
  eslintConfigPrettier,
  {ignores: [
    "dist/*"
  ]
  }
];
```

Una vez hecho lo anterior, debemos crear un fichero de configuración de Prettier:

```
[~/DSI/theory-examples()]$echo {} > .prettierrc
```

Además, también podemos crear un fichero `.prettierignore` que contendrá aquellos ficheros y directorios que no queremos formatear:

```
[~/DSI/theory-examples()]$touch .prettierignore
[~/DSI/theory-examples()]$cat .prettierignore
dist
node_modules
.prettierrc
.eslint.config.mjs
package-lock.json
package.json
tsconfig.json
```

Antes de formatear el código fuente de nuestro fichero `index.ts`, vamos a modificarlo para que contenga lo siguiente:

```
function add(firstNumber: number, secondNumber: number, ...remainingNumbers: number[]) {
  let result = firstNumber + secondNumber;
  if (remainingNumbers.length) {
    result += remainingNumbers.reduce((prev, current) => prev + current);
  }
  return result;
}

console.log(add(1, 2));
console.log(add(1, 2, 3));
console.log(add(1, 2, 4));
```

Ahora, ejecutemos `prettier`, para formatear el código fuente del fichero anterior:

```
[~/DSI/theory-examples()]$npx prettier --write .
src/index.ts 123ms
```

El contenido del fichero será el siguiente una vez formateado:

```
function add(
  firstNumber: number,
  secondNumber: number,
  ...remainingNumbers: number[]
) {
  let result = firstNumber + secondNumber;
  if (remainingNumbers.length) {
    result += remainingNumbers.reduce((prev, current) => prev + current);
  }
  return result;
}

console.log(add(1, 2));
console.log(add(1, 2, 3));
console.log(add(1, 2, 4));
```

Se puede observar como el principal cambio se encuentra en la declaración de la función `add`, donde cada uno de sus parámetros han sido ubicados en líneas diferentes.

Para integrar la funcionalidad de Prettier en VSCode, podemos hacer uso de su extensión. Una vez instalada, debemos seleccionar Prettier como formateador por defecto en la configuración de VSCode. Para ello, podemos hacer clic con el segundo botón del ratón en el contenido de un fichero de código fuente que deseemos formatear, hacer clic en la opción `Format Document With...` y seleccionar la última opción `Configure Default Formatter` para luego seleccionar `Prettier - Code formatter`. Una vez hecho lo anterior, podremos formatear el código de un fichero que estemos editando haciendo uso de la combinación de teclas `Shift + Alt + F`.

Por último, cabe mencionar que, al existir un fichero de configuración `.prettierrc` en la raíz del proyecto, la configuración aplicada por VSCode a la hora de formatear el código será la especificada en dicho fichero. Si se desea aplicar la configuración por defecto de VSCode, se tendrá que eliminar el fichero `.prettierrc` de la raíz del proyecto. Puede consultar la siguiente documentación sobre las opciones de configuración de Prettier.