

DWC

(Desarrollo Web en entorno cliente)



JavaScript

Tema 16

**Desarrollo de proyectos en
JavaScript**

Índice

1.- Desarrollo de proyectos con JavaScript.....	1
2.- Modules	2
2.1.- ¿Qué es un módulo?	2
2.2.- Características de los módulos	5
2.3.- Aplicación en modo producción	8
3.- TypeScript.....	12
4.- Node Package Manager (npm)	18

1.- Desarrollo de proyectos con JavaScript



En los últimos años JavaScript ha evolucionado mucho y se ha adaptado a los lenguajes más utilizado en el desarrollo de aplicaciones.

En esta evolución hay varios aspectos a tener en cuenta, como son:

Evolución en el lenguaje JavaScript.

Las incorporaciones de nuevas funcionalidades al lenguaje a través de los EsmacScript le van dando más potencia al lenguaje.

Sistemas de Módulos

Con los módulos se consigue código reutilizable a través de los import / export.

Gestión de dependencias

Nuestros proyectos van a utilizar diferentes librerías tanto nuestras como externas, para realizar una gestión automatizada de todas ellas se incorpora npm

Estructura del proyecto (Scaffolding)

A medida que se van complicando los proyectos es necesario determinar una estructura de carpetas clara y que permita utilizar las nuevas herramientas de JavaScript.

Modo desarrollo y modo producción

Como en todos los proyectos tendremos dos formas de trabajar, en desarrollo cuando estamos realizando la aplicación y en modo producción la aplicación se prueba ya como versión definitiva en el servidor.

La principal diferencia es que en desarrollo nuestro código hace referencia a numerosas dependencias y en producción se optimiza todas las dependencias y el código.

Empaquetadores (WebPack)

Son las herramientas que permiten generar “bundles” que son como empaquetar todos los módulos y referencias externas en un solo fichero de JavaScript.

WebPack es uno de los más utilizados, estas herramientas suelen utilizar minificadores (Minify), cargadores de aplicaciones (loaders), “traductores” (Babel) y ejecución de scripts

TypeScript

Un supertipo de JavaScript que aporta tipado de datos y mejora el uso de clases. Los navegadores no entienden TypeScript con lo cual nuestros programas en TypeScript deberán ser compilados (transpilados) a JavaScript para utilizarlos en el navegador.

FrameWorks de desarrollo (Angular)

Son herramientas de desarrollo que incorporan todo lo comentado anteriormente y que permiten realizar todo de una manera automática

2.- Modules

A medida que nuestra aplicación crece, queremos dividirla en varios archivos, los llamados "módulos". Un módulo puede contener una clase o una biblioteca de funciones para un propósito específico.

Durante mucho tiempo, JavaScript existió sin una sintaxis de módulo a nivel de idioma. Eso no fue un problema, porque inicialmente los scripts eran pequeños y simples, por lo que no era necesario.

Pero finalmente los scripts se volvieron cada vez más complejos, por lo que la comunidad inventó una variedad de formas de organizar el código en módulos, bibliotecas especiales para cargar módulos a pedido.

Para nombrar algunos (por razones históricas):

- AMD: uno de los sistemas de módulos más antiguos, implementado inicialmente por la biblioteca require.js.
- CommonJS: el sistema de módulos creado para el servidor Node.js.
- UMD: un sistema de módulos más, sugerido como universal, compatible con AMD y CommonJS.

Ahora, todos estos se convierten lentamente en parte de la historia, pero aún podemos encontrarlos en antiguos scripts.

El sistema de módulos a nivel de lenguaje apareció en el estándar en 2015, evolucionó gradualmente desde entonces y ahora es compatible con todos los navegadores principales y en Node.js. Así que estudiaremos los módulos de JavaScript modernos a partir de ahora.

2.1.- ¿Qué es un módulo?

Un módulo es solo un archivo. **Un script es un módulo.** Tan sencillo como eso.

Los módulos pueden cargarse entre sí y utilizar directivas especiales **export** e **import** para intercambiar funcionalidades y llamar a funciones de un módulo desde otro:

- **Export** La palabra clave etiqueta variables y funciones que deberían ser accesibles desde fuera del módulo actual.
- **Import** permite la importación de funcionalidades de otros módulos.

Por ejemplo, tenemos un archivo **say.js** que exporta una función:

```
// ■ say.js

export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

Entonces otro archivo puede importarlo y usarlo:

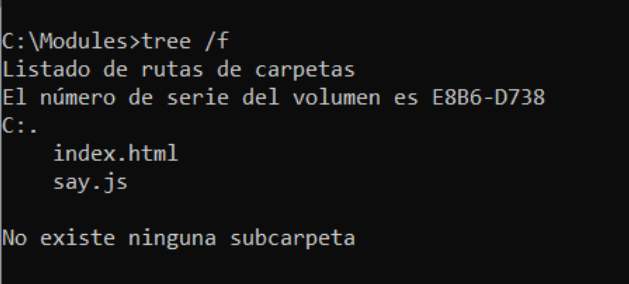
```
// ■ main.js

import {sayHi} from './sayHi.js';

sayHi('John'); // Hello, John!
```

La directiva `import` carga el módulo por ruta `./sayHi.js` relativa al archivo actual y asigna la función exportada `sayHi` a la variable correspondiente.

Ejecutemos el ejemplo en el navegador. Para ello vamos a crear una carpeta llamada `Modules` y en ella colocaremos los archivos `sayHi.js` e `index.html`



```
C:\Modules>tree /f
Listado de rutas de carpetas
El número de serie del volumen es E8B6-D738
C:.
  index.html
  say.js

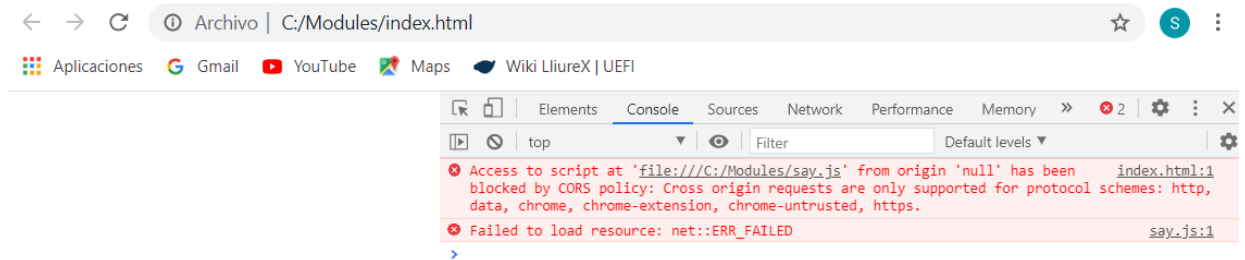
No existe ninguna subcarpeta
```

Como los módulos admiten palabras clave y funciones especiales, debemos decirle al navegador que un script debe tratarse como un módulo, para ello deberemos utilizar el atributo **`<script type="module">`**.

```
<!doctype html>
<script type="module">
  import {sayHi} from './say.js';

  sayHi('Jhon');
</script>
```

El navegador busca y evalúa automáticamente el módulo importado (y sus importaciones si es necesario) y luego ejecuta el script.



Los módulos funcionan solo a través de HTTP (s), no en archivos locales

Si intentamos abrir una página web localmente, a través del protocolo file://, veremos que las directivas import/export no funcionan. Para evitar este problema deberemos utilizar un servidor web local

Existen numerosas alternativas, una de ellas podría ser **http-server** un servidor web local muy sencillo desarrollado en Node.js

Disponible en <https://www.npmjs.com/package/http-server>

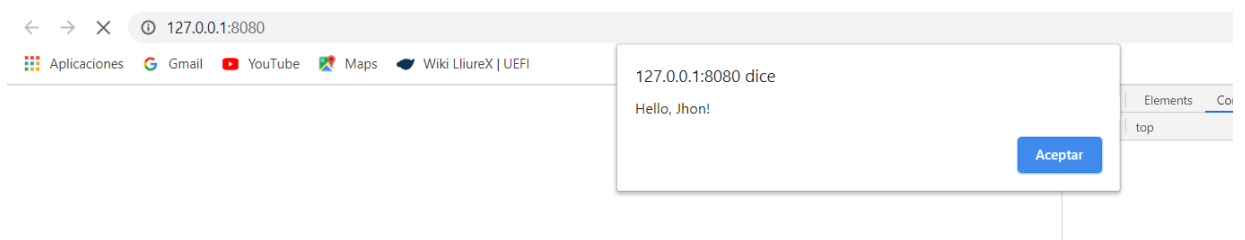
Instalamos el servidor web local

```
C:\Modules>npm install --global http-server
```

Arrancamos el servidor Web desde el directorio Modules ./ (path) y -o (abrir navegador)

```
C:\Modules>http-server ./ -o
Starting up http-server, serving ./
Available on:
  http://192.168.56.1:8080
  http://192.168.1.139:8080
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
open: http://127.0.0.1:8080
[2020-12-02T19:25:20.280Z] "GET /say.js" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.66 Safari/537.36"
(node:11460) [DEP0066] DeprecationWarning: OutgoingMessage.prototype._headers is deprecated
```

Y el resultado es:



2.2- Características de los módulos

Hay características importantes tanto para el navegador como para JavaScript del lado del servidor.

use strict

Los módulos siempre utilizan use strict, por defecto. Por ejemplo, asignar a una variable no declarada dará un error.

```
<script type="module">
  a = 5; // error
</script>
```

scope de las variables

Cada módulo tiene su propio alcance de nivel superior. En otras palabras, las variables y funciones de nivel superior de un módulo no se ven en otros scripts.

En el siguiente ejemplo, se importan dos scripts **hello.js** que intenta utilizar la variable **user** declarada en **user.js** y falla:

user.js

```
let user = "John";
```

hello.js

```
alert(user); // no such variable (each module has independent variables)
```

index.html

```
<!doctype html>
<script type="module" src="user.js"></script>
<script type="module" src="hello.js"></script>
```

Se espera que los módulos tengan en export lo que quieren que sea accesible desde el exterior y en import lo que necesitan.

Así que debemos importar user.js en hello.js y obtener la funcionalidad requerida de él en lugar de depender de variables globales.

Esta es la variante correcta:

user.js

```
export let user = "John";
```

hello.js

```
import {user} from './user.js';

document.body.innerHTML = user; // John
```

index.html

```
<!doctype html>
<script type="module" src="hello.js"></script>
```

Sólo se evalúan la primera vez

Si el mismo módulo se importa a varios otros lugares, su código **se ejecuta solo la primera vez**, luego las exportaciones se entregan a todos los importadores. Eso tiene importantes consecuencias. Veámoslos usando ejemplos:

Primero, si la ejecución de un código de módulo trae efectos secundarios, como mostrar un mensaje, importarlo varias veces lo activará solo una vez, la primera vez:

■ alert.js

```
alert("Module is evaluated!");
// Import the same module from different files
```

```
// ■ 1.js
import `./alert.js`; // Module is evaluated!

// ■ 2.js
import `./alert.js`; // (shows nothing)
```

En la práctica, el código del módulo de nivel superior se utiliza principalmente para la inicialización, la creación de estructuras de datos internas y, si queremos que algo sea reutilizable, exportarlo.

Ahora, un ejemplo más avanzado.

Digamos que un módulo exporta un objeto:

■ admin.js

```
export let admin = {
  name: "John"
};
```


Si este módulo se importa en varios archivos, el módulo solo se evalúa la primera vez, que se crea el objeto y luego se pasa a todos los demás importadores.

Todos los importadores obtienen exactamente el único objeto admin:

```
// ■ 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// ■ 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// Both 1.js and 2.js imported the same object
// Changes made in 1.js are visible in 2.js
```

El módulo se ejecuta solo una vez. Las exportaciones se generan y luego se comparten entre los importadores, por lo que, si algo cambia en el objeto admin, otros módulos lo verán.

Tal comportamiento nos permite **configurar** módulos en la primera importación. Podemos configurar sus propiedades una vez, y luego, en otras importaciones, estará listo.

Por ejemplo, el módulo admin.js puede proporcionar cierta funcionalidad, pero se espera que las credenciales lleguen al objeto admin desde el exterior:

■ admin.js

```
export let admin = { };

export function sayHi() {
  alert(`Ready to serve, ${admin.name}!`);
}
```

En init.js, el primer script de nuestra aplicación, configuramos admin.name. Entonces todos lo verán, incluidas las llamadas realizadas desde su interior en admin.js:

■ init.js

```
import {admin} from './admin.js';
admin.name = "Pete";
```

Otro módulo también puede ver admin.name:

■ other.js

```
import {admin, sayHi} from './admin.js';

alert(admin.name); // Pete

sayHi(); // Ready to serve, Pete!
```

import.meta

El objeto import.meta contiene la información sobre el módulo actual.

Su contenido depende del medio ambiente. En el navegador, contiene la URL del script, o una URL de la página web actual si está dentro de HTML:

```
<script type="module">
  alert(import.meta.url); // script url (url of the html page for an inline script)
</script>
```

this no está definido

Esa es una característica menor, pero para completar, debemos mencionarla.

En un módulo, el nivel superior this no está definido en comparación con los scripts que no son de módulo, donde this es un objeto global:

```
<script>
  alert(this); // window
</script>
```

```
<script type="module">
  alert(this); // undefined
</script>
```

2.3.- Aplicación en modo producción

En la vida real, los módulos del navegador rara vez se utilizan en su forma "cruda". Por lo general, los agrupamos con una herramienta especial como **Webpack** y los implementamos en el servidor de producción.

Los principales beneficios de usar agrupadores:

- Dan más control sobre cómo se resuelven los módulos, lo que permite módulos simples
- Tener módulos CSS / HTML.

Las herramientas de compilación hacen lo siguiente:

1. Crean un módulo "principal", en el que se pondrá `<script type="module">` en HTML.
2. Analizan sus dependencias: importaciones y luego importaciones de importaciones, etc.
3. Crean un solo archivo con todos los módulos (o varios archivos, eso es ajustable), reemplazando las llamadas `import` nativas con funciones de empaquetado, para que funcione. También se admiten los tipos de módulos "especiales", como los módulos HTML / CSS.
4. En el proceso, se pueden aplicar otras transformaciones y optimizaciones:
 - Se elimina el código inalcanzable.
 - Los `export` no utilizados son eliminados ("tree-shaking").
 - Las declaraciones específicas de desarrollo como `console` y `debugger` son eliminadas.
 - La sintaxis de JavaScript moderna (últimos ECMAScript) se pueden transformar a una anterior con una funcionalidad similar usando **Babel**.
 - El archivo resultante se minimiza (se eliminan los espacios, se reemplazan las variables con nombres más cortos, etc.).

Si usamos herramientas de este tipo, a medida que los scripts se agrupan en un solo archivo (o pocos archivos), las declaraciones `import/export` dentro de esos scripts se reemplazan por funciones especiales de agrupamiento. De esta manera, el script "empaquetado" resultante no contiene ningún `import/export`, y **no requiere `type="module"`**, y podemos ponerlo en un script normal:

```
<!-- Assuming we got bundle.js from a tool like Webpack -->
<script src="bundle.js"></script>
```

Un ejemplo de Webpack sería el utilizado por Angular cuando pasamos a producción nuestro proyecto.

En ese momento se "empaquetan" todos los módulos de JavaScript empleados en el proyecto, los css y html y se generan los bundles correspondientes.

Se añade el o los bundles al index.html y se genera una carpeta llamada dist donde se incluye el proyecto con todos los recursos necesarios.

```
C:\Users\salsa\Downloads\padre-hijo>ng build --prod
Your global Angular CLI version (9.1.3) is greater than your local
version (7.2.2). The local Angular CLI version is used.

To disable this warning use "ng config -g cli.warnings.versionMismatch false".
10% building 3/5 modules 2 active ...es\bootstrap\dist\css\bootstrap.min.cssBrowserslist: caniuse-lite is outdated. Ple
ase run next command `npm update caniuse-lite browserslist`

Date: 2020-12-03T18:01:28.880Z
Hash: 7ef53ec39172128636ac
Time: 17771ms
chunk {0} runtime.b57bf819d5bdce77f1c7.js (runtime) 1.41 kB [entry] [rendered]
chunk {1} main.c4884664b4de9974dd32.js (main) 268 kB [initial] [rendered]
chunk {2} polyfills.20d0bb3b90b253b298ff.js (polyfills) 41 kB [initial] [rendered]
chunk {3} styles.03495b9d637d89aeb9d5.css (styles) 136 kB [initial] [rendered]
```

La carpeta dist del proyecto ya estaría disponible para subirla al servidor en modo producción.

```
C:\Users\salsa\Downloads\padre-hijo\dist>tree /f
Listado de rutas de carpetas
El número de serie del volumen es E8B6-D738
C:..
├── crud
│   ├── 3rdpartylicenses.txt
│   ├── favicon.ico
│   ├── index.html
│   ├── main.c4884664b4de9974dd32.js
│   ├── polyfills.20d0bb3b90b253b298ff.js
│   ├── runtime.b57bf819d5bdce77f1c7.js
│   └── styles.03495b9d637d89aeb9d5.css
└── assets
    ├── foto1.jpg
    ├── foto2.jpg
    ├── foto3.jpg
    └── foto4.jpg
```

Fichero index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Crud</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet" href="styles.03495b9d637d89aeb9d5.css"></head>
<body>
  <app-root></app-root>
  <script type="text/javascript" src="runtime.b57bf819d5bdce77f1c7.js"></script>
  <script type="text/javascript" src="polyfills.20d0bb3b90b253b298ff.js"></script>
  <script type="text/javascript" src="main.c4884664b4de9974dd32.js"></script>
</body>
</html>
```

Fichero main.js

En este fichero se encuentra el resultado del “empaquetamiento” de todos los módulos usados en el proyecto y además minificados.

Se entiende por "minificación" el proceso mediante el cual se eliminan datos innecesarios o redundantes de un recurso sin que se vea afectada la forma en que los navegadores lo procesan. Por ejemplo, eliminar comentarios y formato innecesario, retirar código que no se usa, emplear variables y nombres de funciones más cortos, etc.

```
(window.webpackJsonp=window.webpackJsonp||[]).push([[1],{0:function(t,e,n){t.exports=n("zUnb")}},crn
```

3.- TypeScript

TypeScript es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft. Es un superconjunto de JavaScript, que esencialmente añade tipos estáticos y objetos basados en clases.

TypeScript puede ser usado para desarrollar aplicaciones JavaScript que se ejecutarán en el lado del cliente o del servidor (Node.js)

TypeScript extiende la sintaxis de JavaScript, por tanto, cualquier código JavaScript existente debería funcionar sin problemas. Está pensado para grandes proyectos, los cuales a través de un compilador de TypeScript se traducen a código JavaScript original. **Este proceso de compilación se conoce como transpilación.**

Para poder comenzar a trabajar con TypeScript deberemos instalarlo con **npm**.

```
C:\>npm install -g typescript
```

El archivo **tsconfig.json** es específico de TypeScript y su presencia en un directorio indica que el directorio es la raíz de un proyecto de TypeScript.

El contenido de este archivo especifica los archivos raíz y las opciones del compilador necesarias para compilar el proyecto.

Para esto lo primero será crear una carpeta para el proyecto. Vamos a crear una carpeta llamada **ProyectoTS**

```
C:\>mkdir ProyectoTS  
C:\>cd ProyectoTS  
C:\ProyectoTS>tree /f  
Listado de rutas de carpetas  
El número de serie del volumen es E8B6-D738  
C:.  
No existe ninguna subcarpeta
```

Una vez creada la carpeta crearemos el archivo **tsconfig.json**.

Para crear un archivo **tsconfig.json** predeterminado utilizaremos el comando **tsc --init**

```
C:\ProyectoTS>tsc -init
message TS6071: Successfully created a tsconfig.json file.

C:\ProyectoTS>tree /f
Listado de rutas de carpetas
El número de serie del volumen es E8B6-D738
C:.
    tsconfig.json

No existe ninguna subcarpeta
```

Este comando genera un fichero con todas las opciones de compilación

```
{
  "compilerOptions": {
    /* Visit https://aka.ms/tsconfig.json to read more about this file */

    /* Basic Options */
    // "incremental": true,           /* Enable incremental compilation */
    "target": "es5",                /* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES6', 'ES2015', 'ES2016', 'ES2017', 'ES2018', 'ES2019', 'ES2020', 'ES2021', 'ESNext'. */
    "module": "commonjs",           /* Specify module code generation: 'none', 'commonjs', 'amd', 'system', 'umd', 'es6', 'es2015'. */
    // "lib": [],                     /* Specify library files to be included in the compilation. */
    // "allowJs": true,               /* Allow javascript files to be compiled. */
    // "checkJs": true,               /* Report errors in .js files. */
    // "jsx": "preserve",             /* Specify JSX code generation: 'preserve', 'react-native', 'react'. */
    // "declaration": true,           /* Generates corresponding '.d.ts' file. */
    // "declarationMap": true,        /* Generates a sourcemap for each corresponding '.d.ts' file. */
    // "sourceMap": true,             /* Generates corresponding '.map' file. */
    // "outFile": "./",               /* Concatenate and emit output to single file. */
    // "outDir": "./",               /* Redirect output structure to the directory. */
    // "rootDir": "./",              /* Specify the root directory of input files. Useful to control the output directory relative to the root of the project. */
    // "composite": true,             /* Enable project compilation */
    // "tsBuildInfoFile": "./",       /* Specify file to store incremental compilation information */
    // "removeComments": true,        /* Do not emit comments to output. */
    // "noEmit": true,                /* Do not emit outputs. */
    // "importHelpers": true,         /* Import emit helpers from 'tslib'. */
    // "downlevelIteration": true,    /* Provide full support for iterables in 'for-of' loops and default parameters. */
    // "isolatedModules": true,       /* Transpile each file as a separate module (similar to 'ts.transpileModule') */

    /* Strict Type-Checking Options */
    "strict": true,                 /* Enable all strict type-checking options. */
    // "noImplicitAny": true,         /* Raise error on expressions and declarations with an implied 'any' type. */
    // "strictNullChecks": true,     /* Enable strict null checks. */
    // "strictFunctionTypes": true,   /* Enable strict checking of function types. */
    // "strictBindCallApply": true,   /* Enable strict 'bind', 'call', and 'apply' method checks. */
    // "strictPropertyInitialization": true, /* Enable strict checking of property initialization. */
    // "noImplicitThis": true,        /* Raise error on 'this' expressions with an implied 'any' type. */
    // "alwaysStrict": true,          /* Parse in strict mode and emit "use strict" for each source file. */
  }
}
```

Nosotros sólo necesitaremos las básicas, con lo cual pasaremos a modificar el archivo y dejar las opciones básicas.

```
{
  "compilerOptions": {
    "target": "es6", /*Specify ECMAScript target version*/
    "rootDir": "ts-src", /*ts source directory*/
    "outDir": "web-app/js" /*where to compile javascript files*/
  }
}
```

Ahora ya tenemos un fichero en la raíz de nuestra carpeta de proyecto que indica que es un proyecto en TypeScript.

El siguiente paso sería implementar nuestros archivos de TypeScript.

En el fichero de configuración hemos indicado que la carpeta ts-src será la que contenga los archivos de TypeScript.

Creamos la carpeta ts-src

```
C:\ProyectoTS>mkdir ts-src

C:\ProyectoTS>tree /f
Listado de rutas de carpetas
El número de serie del volumen es E8B6-D738
C:.\
├── tsconfig.json
└── ts-src
```

Añadimos un archivo llamado codigoTS.ts con un código sencillo de TypeScript que incluye una función y una clase llamada Alumno.

```
function getMessage(): string {
    return "Hola...";
}

class Alumno{
    nombre:string;
    constructor (nombre:string){
        this.nombre=nombre;
    }

    saludo(){
        alert("Hola " +this.nombre)
    }
}
```

Hasta ahora lo que tenemos es la siguiente estructura de carpetas (**scaffolding**)

```
C:\ProyectoTS>tree /f
Listado de rutas de carpetas
El número de serie del volumen es E8B6-D738
C:.\
├── tsconfig.json
└── ts-src
    └── codigoTS.ts
```

Ahora nuestro código en TypeScript lo podríamos transpilar a JavaScript para poder ejecutarlo en el navegador. Hemos indicado en las opciones de compilación que la carpeta donde se compilara a JavaScript sea **web-app/js**.

Para compilar utilizamos el comando `tsc --watch`

```
C:\ProyectoTS>tsc --watch
[13:07:51] Starting compilation in watch mode...

[13:07:53] Found 0 errors. Watching for file changes.
```

Esto lo que hace es iniciar el proceso de compilación de los ficheros TypeScript a ficheros JavaScript.

Se habrá creado la carpeta `web-app/js` y dentro de ella debería aparecer el fichero en Javascript

```
C:\ProyectoTS>tree /f
Listado de rutas de carpetas
El número de serie del volumen es E8B6-D738
C:.
|
|-- tsconfig.json
|
|-- ts-src
|   |-- codigoTS.ts
|
|-- web-app
|   |-- js
|       |-- codigoTS.js
```

Si vemos el contenido del fichero `codigoTS.js` podremos ver que es un fichero Javascript en EmacScript ES6 como le indicamos en la opción de compilación.

```
function getMessage() {
    return "Hola...";
}
class Alumno {
    constructor(nombre) {
        this.nombre = nombre;
    }
    saludo() {
        alert("Hola " + this.nombre);
    }
}
```

En este caso las modificaciones son sencilla, lo que se ha hecho es quitar el tipado de datos que proporciona TypeScript y que JavaScript no soporta. Las clases ya vienen incorporadas en el ES6 y no se produce ningún cambio importante.

Si hubiéramos indicado en el `tsconfig.json` que queríamos un **EmacScript ES5** que no soporta clases la transformación de la clase `Alumno` se realizaría de la forma nativa de JavaScript basada en **prototype**

Modificamos el EmacScript a ES5

```
{
  "compilerOptions": {
    "target": "es5", /*Specify ECMAScript target version*/
    "rootDir": "ts-src", /*ts source direcotry*/
    "outDir": "web-app/js" /*where to compile javascript files*/
  }
}
```

El fichero codigoTS.js quedaría así

```
function getMessage() {
  return "Hola...";
}
var Alumno = /** @class */ (function () {
  function Alumno(nombre) {
    this.nombre = nombre;
  }
  Alumno.prototype.saludo = function () {
    alert("Hola " + this.nombre);
  };
  return Alumno;
})();
```

El modo **watch** lo que hace es que se quede visible el proceso de compilación y si se modificara algún archivo fuente (archivos en TypeScript) sean compilados automáticamente a Javascript.

```
C:\ProyectoTS>tsc --watch
[13:07:51] Starting compilation in watch mode...

[13:07:53] Found 0 errors. Watching for file changes.

[13:22:38] File change detected. Starting incremental compilation...

[13:22:38] Found 0 errors. Watching for file changes.
```

Como ya hemos visto los ficheros de JavaScript se incluyen en nuestros ficheros HTML para gestionar el funcionamiento de nuestras páginas web.

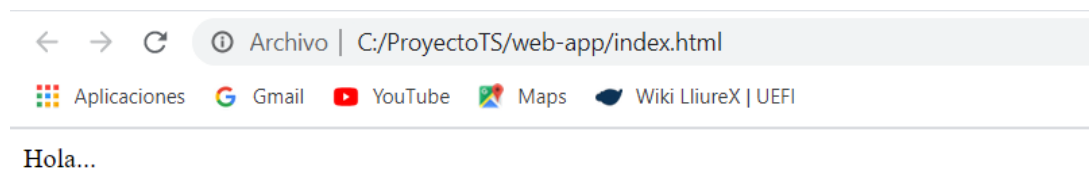
Vamos a crear nuestro archivo **index.html** de nuestro proyecto.

```
<html>
  <head>
    <script type = "text / javascript" src = "../js/codigoTS.js"></script>
  </head>
  <body>
    <div id="display-div" > </ div>
  </body>
  <script>
    document.getElementById( "display-div" ).innerHTML = getMessage();
  </script>
</html>
```

Lo creamos en la carpeta web-app

```
C:\ProyectoTS>tree /f
Listado de rutas de carpetas
El número de serie del volumen es E8B6-D738
C:..
|
|   tsconfig.json
|
|---ts-src
|    codigoTS.ts
|
|---web-app
|    index.html
|    |
|    |---js
|         codigoTS.js
```

Abrimos la página en el navegador para comprobar que el código implementado en TypeScript ha sido compilado a Javascript y que funciona correctamente en el navegador.



4.- Node Package Manager (npm)

Repasar los conceptos básicos vistos en despliegue.

- npm sintaxis
- npm init
- npm install
- node_modules
- package.json
- package-lock.json: evita el comportamiento general de actualizar versiones **minor** o **fix** de modo que cuando alguien clona nuestro repositorio y ejecuta **npm install** en su equipo, **npm** examinará **package-lock.json** e instalará la versión exacta de los paquete que nosotros habíamos instalado, ignorando así los **^** y **~** de **package.json**.