Configuring TypeScript compiler

A detailed manual for essential TypeScript configuration options

Max Koretskyi aka Wizard Follow
Feb 1, 2017 · 11 min read

> **We're organizing the first Angular conference in Kiev, Ukraine as part of Angular In Depth community. Read more about it here.** I'll be happy to see you at the conference. We'll have experts from Angular team and the community and we made tickets affordable for most developers (around $150), even if you pay out of your own pocket.

Setting up tools

TypeScript files are compiled into JavaScript using TypeScript compiler. The compiler can be installed as `typescript` package through `npm` . As with any npm package, you can install it locally or globally, or both, and compile the TS files by running `tsc` on the command line for global installations or `$(npm bin)/tsc` for local installations.

All compiler options described in this article are listed here.

. . .

Input files location

TS compiler accepts a list of files to compile as parameters. For example:

```
$ tsc main.ts router/index.ts
```

However, most of the time, we don't specify files list manually. TS automatically compiles all files in a project directory and its sub-

directories. It treats every directory with `tsconfig.json` file in the root as a project directory. When we run `tsc` on the command line, it searches for `tsconfig.json` starting in the current directory and continuing up the parent directory chain.

`tsconfig.json` can be created by the compiler automatically using `init` flag:

```
tsc --init
```

But it generates the configuration file with a few predefined options. For our purposes we will create the empty `tsconfig.json` manually and run `tsc` compiler inside this folder:

```
$ echo {} > tsconfig.json && tsc
```

Or we can use `-p` compiler option with the path to the project directory, i.e. the directory with the `tsconfig.json` file in the root.

```
$ tsc -p /path/to/folder/with/tsconfig
```

At the moment, TS compiles recursively searches for all files in the root directory and sub-directories and compiles them. However, we can control, where the compiler will be looking for the files. This is done through `files` configuration option.

So, we can tell the compiler to only compile files `main.ts` and `router/b.ts` and leave everything else out.

```
{
  "compilerOptions": { ... },
  "files": [
    "main.ts",
    "router/b.ts"
  ]
}
```

> *Note: TS compiler will also compile files that are referenced inside any file from the* `files` *list. For example, if* `main.ts` *imports exports from* `a.ts` *, this file will also be compiled.*

Instead of listing each file manually, we can use `include` option to specify directories using glob-like file patterns. For example, we can compile all files inside `router` directory like this:

```
{
  "compilerOptions": { ... },
  "include": [
    "router/*"
  ]
}
```

> *Note: There is* `rootDir` *compiler option, which* ***is not*** *used to specify input to a compiler. It's used to control the output directory structure alongside with* `outDir` *.*

If you want to exclude some files or folders from the compilation, you can use `exclude` option, which takes a glob-like file patterns. Suppose, we want to compile all files in the project directory except for files inside `navigation` folder. In this case, we can use the configuration like this:

```
{
  "compilerOptions": { ... },
  "exclude": [
    "navigation/*"
  ]
}
```

In the case of conflicts, the priority is set in the following order:

1. Files
2. Exclude
3. Include

This means that if a file listed in the `files` option it's included regardless of the configuration in the `exclude` option. If a file is listed in both `exclude` and `include` option, the file **is excluded**. By

default, `tsc` excludes files in `node_modules`, `bower_components`, `jspm_packages` and `<outDir>`. We'll talk about `outDir` option in the next section.

. . .

Output location

By default, TS compiler outputs transpiled files to the same directory where the original TS files is found. However, this can be changed using `outDir` compiler option.

```
{
  "compilerOptions": {
    "outDir": "dist"
  }
}
```

Now, when we run `tsc` all output will copied into `dist` folder preserving the original directory structure.

> *Note: the options described in this and the following sections go under* `compilerOptions` *as opposed to the options in the previous section that were defined in the root*

But TS can also concatenate all files into one file if we specify `outFile` compiler option. So, with the following configuration defined:

```
{
  "compilerOptions": {
    "outFile": "dist/bundle"
  }
}
```

All output will be concatenated into `bundle.js` file and put inside `dist` folder.

> *Note:* `outFile` *option is only supported if resulting modules are either `amd` or 'system'. We'll talk about module systems later.*

If both `outDir` and `outFile` options are specified, the latter takes precedence and the `outDir` option is ignored.

By default, TS compiler produces output even if there are errors during compilation. This behavior can be changed using `noEmitOnError` option:

```
{
  "compilerOptions": {
    "noEmitOnError": true
  }
}
```

. . .

Output files types

Using the default configuration, the compiler only emits `.js` files. To be able to debug TS files during runtime we need source maps. To enable source maps generation we can use `sourceMap` option:

```
"compilerOptions": {
  "sourceMap": true
}
```

When you run the compiler, you will see that mapping files will be emitted by the compiler alongside their corresponding `ts` files. So, if you have `main.ts` file, after the compilation you will have 3 files:

```
main.ts
main.js
main.js.map
```

Inside the `main.js` file, you will see the URL to the source map file:

```
//# sourceMappingURL=main.js.map
```

You can modify the generated URL that is added to map files like this:

```
"compilerOptions": {
  "mapRoot": "/sourcemap/directory/on/webserver",
}
```

Which produces the following path:

```
//#
sourceMappingURL=/sourcemap/directory/on/webserver/main.js.m
ap
```

The `map` file references the source using these two keys:

```
"sourceRoot": "",
"sources": [
  "/typescript/main.ts"
],
```

You can modify the root for the source file using `sourceRoots` option:

```
"sourceRoot": "/path/to/sources",
```

which produces the following output:

```
"sourceRoot": "/path/to/sources",
"sources": [
  "main.ts"
],
```

If you wish to put sources inside a mapping file (either because you want to save a browser a request to your webserver or your production doesn't serve sources as separate files), you can use the following option:

```
{
  "compilerOptions": {
    "sourceMap": true,
    "inlineSources": true
  }
}
```

In this way the compiler will put original TS sources into `sourcesContent` property:

```
{
  "version": 3,
  "file": "main.js",
  "sourceRoot": "",
  "sources": [
    "main.ts"
  ],
  "names": [],
  "mappings": ";AAAA;IAAA;IAAgB,CAAC;...",
  "sourcesContent": [
    "export class Main {}"
  ]
}
```

Also, TS allows putting the source map files content inside `.js` files using:

```
"compilerOptions": {
  "inlineSourceMap": true
}
```

In this way, instead of having a separate file `main.js.map`, the contents of that file will be included into the `main.js` file like this:

```
//#
sourceMappingURL=data:application/json;base64,eyJ2ZXJza...
```

Where `sourceMappingURL` is Data URI.

> *Note: You can specify either* `sourceMap` *to produce a separate map file or* `inlineSourceMap` *to inline map file into the transpiled* `.js`

> *file, but not both.* `inlineSources` *can be used with either option.*

You might have guessed that by combining `inlineSourceMap` and `inlineSources` you can have only `js` file with source maps and sources included into it.

• • •

Transpiling

TypeScript is a superset of ES6, so you're essentially writing TS code using ES6 version of JavaScript. However, when compiled, the resulting JS code can be in ES5 or earlier. You need to define which version of JS the compiler should transpile into. This can be set using `target` option:

```
{
  "compilerOptions": {
    "target": "es6"
  }
}
```

At the time of this writing, all browsers support more than 90% of the spec, so `es6` may be a good option with some shims. Since the default target is `ES3`, you probably will want to set `target` to the most recent supported version, which is at least `es5.`

You write TS sources using ES6 modules, however as of January 2016 no browser natively supports this module system. So you may want to transpile ES6 modules into a different module system: CommonJS, AMD, SystemJS. This can be done using `module` option. There are build-time or run-time transpilers that transpile the ES6 module system into one of the module systems supported by a build system (Webpack) or a module loader (SystemJS). If not specified, the `module` defaults to `ES6` if `target` is `ES6`, or `CommonJS` otherwise. I prefer to set `target` to `CommonJS` explicitly:

```
{
  "compilerOptions": {
    "module": "CommonJS"
  }
}
```

TS supports decorators from the ES7 proposal. For example, they are heavily used during Angular2 TS development. In order to be able to use decorators in the TS sources, the following option should be set:

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

Angular2 DI also uses metadata information to understand what type of dependency to inject. To have this metadata present in the output, use the `emitDecoratorMetadata` option:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

To be able to use classes from the ES standard libraries in your TS sources, you should use `lib` option and specify all standard ES6 libraries *interfaces* used in your sources. For example, to use `Reflect` object or `Array.from` from ES6 and `DOM` configure the following:

```
{
  "compilerOptions": {
    "lib": ["es6", "dom"],
  }
}
```

By default, TS includes `DOM,ES5,ScriptHost` for the `ES5` target and `DOM,ES6,DOM.Iterable,ScriptHost` for `ES6`. If you set `lib` option, the default libraries are not injected by the compiler automatically and have to be listed manually.

> *Note: By specifying `lib` you simply tell TS compiler to not throw error if classes or API's from those libraries are encountered during*

> *transpilation process. This option has no effect on the output since a library is simply a* `d.ts` *file with lib API interfaces.*

. . .

Module resolution

As with `node`'s `require` in ES6 modules there are relative and absolute/non-relative module references. Modules are resolved differently based on whether the module reference is relative or non-relative. A relative module reference starts with `/`, `./` or `../` and such module references are resolved relative to the importing file.

Non-relative modules resolution algorithm can be defined using `moduleResolution` option and is described in great details here. If not specified, it's set to `node` for `module===CommonJS` and `classic` for other module systems.

With module resolution strategy set to `node`, TS compiler looks up modules in `node_modules` folder. But if your module is located in another folder, you can use `paths` option to add a custom folder to the list of folders to look up modules in. Suppose, in your TS code you reference a module like this:

```
import { jQuery} from 'jquery';
```

And your `jquery` folder is placed inside `libs` folder. So you can use the following configurations:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "jquery": [
        "libs/jquery"
      ]
    }
  }
}
```

which tells to the compiler that when `jquery` module is referenced, it should go look inside `libs/jquery`. The compiler will look for

`jquery.[ts|d.ts]` inside `lib` directory first, and if not found, will proceed to looking inside `libs/jquery` directory. It will first try to locate `package.json` file with `typings` property specifying the main file, and if not found will default to `index.[ts|d.ts].`

But what you're specifying in `paths` is actually a pattern and you can use `*` to match any module. So the above configuration can be replaced with the following:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "*": [
        "libs/*"
      ]
    }
  }
}
```

If you use `--traceResolution` you will see the following:

```
Module name 'jquery', matched pattern '*'.
Trying substitution 'libs/*', candidate module location:
'libs/jquery'.
```

You can see that the compiler replaces the asterisk in the path with the matched pattern. This gives us great flexibility as we can match part of the module name. A common use case is when the module name doesn't match the directory structure. For example, you reference libraries inside your code like this:

```
import { jQuery } from 'package/vendors/jquery';
```

But in your directory structure the `jquery` library is placed inside `libs` folder. With the following configuration, the compiler will be able to locate `jquery` library:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "package/vendors/*": [
        "libs/*"
      ]
    }
  }
}
```

In this case, the compiler assigns `jquery` to `*`, and so `libs/*` becomes `libs/jquery` when resolving modules.

> *Note: If you set* `paths` *option,* `baseUrl` *is required. It specifies the base directory to resolve non-relative modules in.*

If `paths` option is set, the compiler goes through folders defined in `paths` and **only** checks `node_modules` folder **if nothing is found**. The first resolved module is used and no other paths are checked. So if you have a module placed inside both `node_modules` and you custom folder, the module in your custom folder will be picked up by the compiler. If you need the compiler to use the module inside `node_modules` folder, add it to `paths` before your custom folder:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "*": [
        "*",
        "node_modules/*",
        "generated/*"
      ]
    }
  }
}
```

> *Note: the* `typeRoots` *option* ***is not*** *used when resolving external modules (ES6 modules).*

. . .

Working with declaration files

TypeScript provides a mechanism to define a member (variable/class) that is not transpiled into JavaScript and the actual implementation is expected to be available during runtime. This feature was designed to enable integration with the existing JavaScript code, for example a browser API or open-source libraries like jQuery.

When you use an object that is not defined in TS project files the compiler reports and error:

```
logger.log();
Error:(2, 1) TS2304:Cannot find name 'logger'.
```

To fix the problem, you can write the following:

```
declare var logger: {log: () => void};
logger.log();
```

This is called **ambient** declaration and ambient declarations do not have any output and are only used during compilation. Ambient declarations are created using `declare` keyword. It's a good practice to have a common place for such declarations and so TS provides a special file type to group them—declaration files that have `.d.ts` extension. Such files can only contain ambient declaration and are heavily used during development. For example, when you use `console.log()` in your code, TS doesn't report an error because the `console` object has already been defined in `lib.d.ts` file that comes with `typescript` npm package.

You will most likely need to generate and consume declaration files yourself. I've shown an example of their usage here. As mentioned earlier, these files do not contain actual implementations, but define classes API and values available during runtime. To have the compiler emit `.d.ts` files, use the `declaration` option:

```
"compilerOptions": {
  "declaration": true
}
```

It is sometime convenient to output declaration files into separate directory or event concatenate them all into one file. Their location can be defined using `declarationDir` option:

```
{
  "compilerOptions": {
    "declaration": true,
    "declarationDir": "declarations"
  }
}
```

and concatenated using `outFile` option (same as for generated `.js` files):

```
{
  "compilerOptions": {
    "declaration": true,
    "outFile": "declarations/index.d.ts"
  }
}
```

When inspecting the generated `.d.ts` file you may see the following:

```
declare module "module1" { ... }
declare module "module2" { ... }
```

This is actually a syntax used before `1.5` for external/ES6 modules. It's used now to support declaring multiple ES6 modules in one file and can only be used in declaration files.

Do not confuse this *quoted* module declaration with the *unquoted* module declarations:

```
declare module module1 { ... }
declare module module2 { ... }
```

This *unquoted name* format was used before `1.5` to declare namespaces. Starting with `1.5` this format usage is discouraged and

is recommended to be replaced with `namespace` keyword:

```
declare namespace module1 { ... }
declare namespace module2 { ... }
```

Unlike *quoted* modules names, namespaces can be used both in `ts` and `d.ts` files.

. . .

WebStorm typescript integration

WebStorm provides integration with TypeScript either through built-in compiler or integrated TypeScript Language Service.

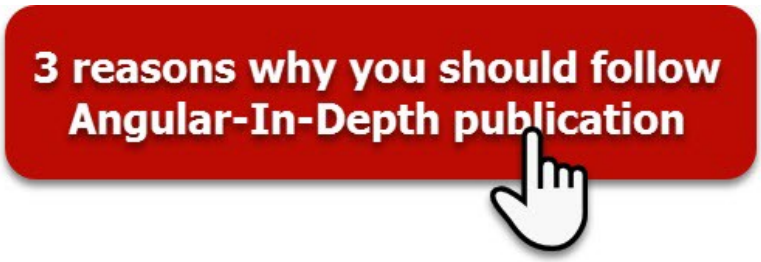
> *Note: It's important to have the same version of TypeScript used by WebStorm and used during build process (for example, used by Webpack loaders). Otherwise, you may gets confused why your build passes while IDE reports errors, or vice verse.*

By default WebStorm uses TS compiler from the `typescript` package located inside `node_modules` **in the project root**, or the package bundled with the IDE. You need to put the same version of `typescript` package which is used during build process in the `node_modules` in the project root.

However, a better way may be to use `custom directory` option to specify path to `typescriptServices.js` and `lib.d.ts`. These files are placed inside `typescript/lib` npm package. So you can tell the IDE to pick up global typescript version by putting `/path/to/nodejs/node_modules/typescript/lib`.

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story. For more insights follow me on Twitter and on Medium.


3 reasons why you should follow
Angular-In-Depth publication