TYPESCRIPT CONFIGURATION

TypeScript configuration for Angular 2 developers

TypeScript is a primary language for Angular application development.

TypeScript is a dialect of JavaScript with design-time support for type-safety and tooling.

Browsers can't execute TypeScript directly. It has to be "transpiled" into JavaScript with the *tsc* compiler and that effort requires some configuration.

This chapter covers some aspects of TypeScript configuration and the TypeScript environment that are important to Angular developers.

- tsconfig.json - TypeScript compiler configuration.
- typings - TypesScript declaration files.

tsconfig.json

We typically add a TypeScript configuration file (`tsconfig.json`) to our project to guide the compiler as it generates JavaScript files.

> Get details about `tsconfig.json` from the official

> TypeScript wiki.

We created the following `tsconfig.json` for the QuickStart:

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false
  }
}
```

The options and flags in this file are essential for Angular 2 applications.

noImplicitAny and *suppressImplicitAnyIndexErrors*

TypeScript developers disagree about whether the `noImplicitAny` flag should be `true` or `false`. There is no correct answer and we can change the flag later. But our choice now can make a difference in larger projects so it merits discussion.

When the `noImplicitAny` flag is `false` (the default), the compiler silently defaults the type of a variable to `any` if it cannot infer the type based on how the variable is used. That's what we mean by *implicit* `any`.

We initialized the `noImplicitAny` flag to `false` in the QuickStart to make learning TypeScript development easier.

When the `noImplicitAny` flag is `true` and the TypeScript compiler cannot infer the type, it still generates the JavaScript

files. But it also **reports an error**. Many seasoned developers prefer this stricter setting because type checking catches more unintentional errors at compile time.

We can set a variable's type to `any` even when the `noImplicitAny` flag is `true`. We do so when that seems like the best choice for the situation, deliberately and explicitly, after giving the matter some thought.

If we set the `noImplicitAny` flag to `true`, we may get *implicit index errors* as well. Most developers feel that *this particular error* is more annoying than helpful. We can suppress them with the following additional flag.

```
"suppressImplicitAnyIndexErrors":true
```

TypeScript Typings

Many JavaScript libraries such as jQuery, the Jasmine testing library, and Angular itself, extend the JavaScript environment with features and syntax that the TypeScript compiler doesn't recognize natively. When the compiler doesn't recognize something, it throws an error.

We use TypeScript type definition files — *d.ts files* — to tell the compiler about the libraries we load.

TypeScript-aware editors leverage these same definition files to display type information about library features.

Many libraries include their definition files in their npm packages where both the TypeScript compiler and editors can find them. Angular is one such library. Peek into the `node_modules/@angular/core/` folder of any Angular application to see several `...d.ts` files that describe parts of Angular.

We need do nothing to get *typings* files for library packages which include *d.ts* files — as all Angular packages do.

Installable typings files

Sadly, many libraries — jQuery, Jasmine, and Lodash among them — do *not* include `d.ts` files in their npm packages. Fortunately, either their authors or community contributors have created separate *d.ts* files for these libraries and published them in well-known locations. The *typings* tool can find and fetch these files for us.

We installed the typings tool with npm (it's listed among the *devDependencies* in the `package.json`) and added an npm script to run that tool automatically after *npm* installation completes.

package.json (postinstall)

```
{
  "scripts": {
    "postinstall": "typings install"
  }
}
```

This *typings* tool command installs the *d.ts* files that we identify in a `typings.json` file into the **typings** folder. We created a `typings.json` file in the QuickStart:

typings.json

```
{
  "globalDependencies": {
    "core-js": "registry:dt/core-
js#0.0.0+20160602141332",
    "jasmine":
"registry:dt/jasmine#2.2.0+20160621224255",
    "node":
"registry:dt/node#6.0.0+20160621231320"
  }
}
```

We identified three *typings* file in the QuickStart, the *d.ts* files for

- core-js that brings ES2015/ES6 capabilities to our ES5 browsers
- jasmine typings for the Jasmine test framework
- node for code that references objects in the nodejs environment; see the webpack chapter for an example.

QuickStart itself doesn't require these typings but many of the documentation samples do. Most of us would be disappointed if we couldn't code against typical ES2015 features or support testing right out-of-the-box.

We can also run the *typings* tool ourselves. The following command (re)installs the typings files, as is sometimes necessary when the `postInstall` hook fails to do so.

```
npm run typings install
```

This command lists the installed typings files:

```
npm run typings list
```

The following command installs or updates the typings file for the Jasmine test library from the *DefinitelyTyped* repository and updates the `typings.config` so we that we get it automatically the next time we install typings.

```
npm run typings -- install dt~jasmine --save --
global
```

> The `--` option is important; it tells npm to pass all arguments to the right of `--` to the *typings* command.
>
> Learn about the features of the *typings* tool at its site on github.