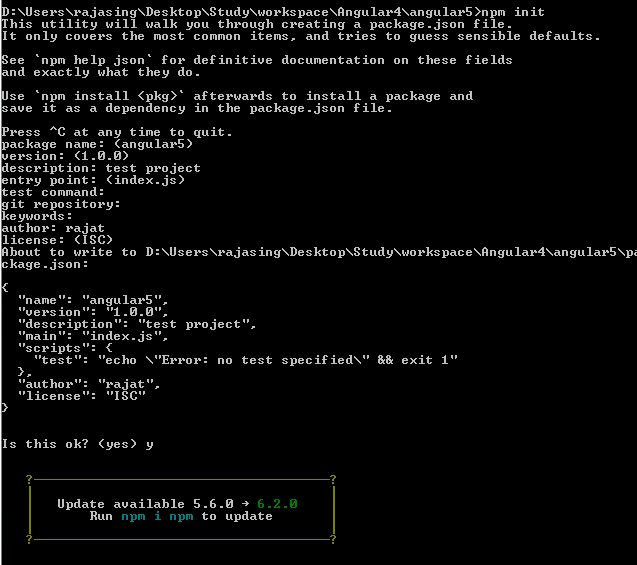
How to create Angular project from scrap

1. Make a folder for a project name as angulartest (“Project name should be small in latter”).
2. Open cmd and navigate till this folder
3. Try Cmd on cmd prompt “ npm init” => It will create package.jason file with the provided input , npm init -y, which creates package.json for us. The -y option accepts the defaults.
4. setup the third-party dependencies .
5. **Core.Js :** Patches the global object (window) with essential features of ES2015 (ES6). You may substitute it with an alternative polyfill that provides the same core APIs. When these APIs are implemented by the major browsers, this dependency will become unnecessary. Essentially, only Reflect polyfill is required in all major browsers (actually, if you use [AoT compilation](https://medium.com/@urish/a-deep-deep-deep-deep-deep-dive-into-the-angular-compiler-5379171ffb7a" \t "_blank), which is the recommended way for production, you can even skip the Reflect polyfill).
6. **Rxjs :** Reactive Extensions Library for JavaScript, which includes methods for transforming, composing, and querying streams of data. It is utilized by several parts of the Angular framework, such as the HTTP and Forms modules. The library provides an Observable implementation, which is currently a [proposed feature](https://github.com/tc39/proposal-observable) to be included in future versions of the JavaScript language.
7. **zone.js :** A polyfill for the [Zone specification](https://gist.github.com/mhevery/63fdcdf7c65886051d55), which has also been proposed for inclusion in the JavaScript language. Zone.js provides the mechanism to hook into asynchronous operations and track outstanding async tasks. Angular does that by creating its own [NgZone](https://angular.io/api/core/NgZone" \t "_blank) which waits until all asynchronous operations like [timers](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout) and [XHR](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) requests are completed and triggers [change detection](https://blog.angularindepth.com/everything-you-need-to-know-about-change-detection-in-angular-8006c51d206f).
8. To install this dependency use below cmd . where save is used to save the dependency in package.json.

**npm i --save core-js zone.js rxjs**

we can specify the particular version by providing @symbol **i.e. npm i –save** [**core-js@version**](mailto:core-js@version)[**zone.js@version**](mailto:zone.js@version) **rxjs@version ,**

**for angular 6 use “*npm i — save rxjs-compat”***

1. we will also use the SystemJS module loader: i.e. “**npm i --save systemjs**
2. To properly work SystemJS needs some configuration. Let’s create a configuration file named systemjs.config.js and add with the following content:

|  |
| --- |
| System.config({  paths: {  'npm:': '/node\_modules/'  },  map: {  app: 'dist/app',  '[@angular/core](http://twitter.com/angular/core)': 'npm:[@angular/core](http://twitter.com/angular/core)/bundles/core.umd.js',  '[@angular/common](http://twitter.com/angular/common)': 'npm:[@angular/common](http://twitter.com/angular/common)/bundles/common.umd.js',  '[@angular/compiler](http://twitter.com/angular/compiler)': 'npm:[@angular/compiler](http://twitter.com/angular/compiler)/bundles/compiler.umd.js',  '[@angular/platform-browser](http://twitter.com/angular/platform-browser)': 'npm:[@angular/platform-browser](http://twitter.com/angular/platform-browser)/bundles/platform-browser.umd.js',  '[@angular/platform-browser-dynamic](http://twitter.com/angular/platform-browser-dynamic)': 'npm:[@angular/platform-browser-dynamic](http://twitter.com/angular/platform-browser-dynamic)/bundles/platform-browser-dynamic.umd.js',  'core-js': 'npm:core-js',  'zone.js': 'npm:zone.js',  'rxjs': 'npm:rxjs',  'tslib': 'npm:tslib/tslib.js'  },  packages: {  'dist/app': {},  'rxjs': {},  'core-js': {},  'zone.js': {}  } }); |

The paths and map sections of the config basically define the full path to the source code files for each of the ESM modules in our app. As you can see, everything resides inside node\_modules, expect for the app code itself, which will live inside dist/app.

The packages section lists the meta data for your packages. In this case, we don’t define any metadata, but adding packages configuration allows us to import files residing in these packages without having to specify the file extension, e.g.:

import { AppComponent } from './app.component';

This is because once the package is listed in the configuration, SystemJS will automatically append the .js to any file inside the package (matched by path) by default.

1. Now we are going to add angular dependency

|  |
| --- |
| 1. **@angular/core**   Critical run-time parts of the framework needed by every application. Includes all metadata decorators, [Component](https://angular.io/api/core/Component), [Directive](https://angular.io/api/core/Directive), dependency injection, and the component life-cycle hooks. Contains core functionality component views, DI and change detection.   1. **@angular/compiler**   [Angular’s Template Compiler](https://medium.com/@urish/a-deep-deep-deep-deep-deep-dive-into-the-angular-compiler-5379171ffb7a). It reads your templates and can convert them to code that makes the application run and render. Typically you don’t interact with the compiler directly; rather, you use it indirectly via platform-browser-dynamic or the offline template compiler.   1. **@angular/common**   Provides the commonly needed services, pipes, and directives such as ngIfand ngFor.   1. **@angular/platform-browser**   Contains the functionality to bootstrap the application in a browser. Basically it includes everything DOM and browser related, especially the pieces that help render into the DOM. May not be required if you use Angular on the platform other than browser (e.g. [angular-iot](https://medium.com/@urish/building-simon-with-angular2-iot-fceb78bb18e5)).  This package also includes the bootstrapStatic() method for bootstrapping applications for production builds that pre-compile templates offline.   1. **@angular/platform-browser-dynamic**   Contains implementations for the dynamic bootstrap of the application. Includes [providers](https://angular.io/api/core/Provider) and a [bootstrap](https://angular.io/guide/ngmodule#bootstrap) method for applications that compile templates on the client (thus, you can skip this module if you use ahead-of-time compilation). Use this package for bootstrapping your application during development |

1. To install Angular dependencies run the command (copy the below into a single line):

|  |
| --- |
| **npm i --save** [**@angular/core**](http://twitter.com/angular/core)[**@angular/compiler**](http://twitter.com/angular/compiler)[**@angular/common**](http://twitter.com/angular/common)[**@angular/platform-browser**](http://twitter.com/angular/platform-browser)[**@angular/platform-browser-dynamic**](http://twitter.com/angular/platform-browser-dynamic) |

1. Setting up typescript : it transform our TypeScript code into JavaScript code.

|  |
| --- |
| **npm i --save-dev typescript** |

1. Now we will create configuration file, tsconfig.json, with the following configuration:

|  |
| --- |
| {  "compilerOptions": {  "outDir": "dist",  "module": "commonjs",  "moduleResolution": "node",  "experimentalDecorators": true,  "emitDecoratorMetadata": true,  "lib": [  "dom",  "es2015"  ]  } } |

This is a very basic configuration file, which basically tells the compiler to write the compiled JavaScript files into the dist directory, to convert ESM modules we use in TypeScript into the CommonJS module format (one of the formats natively supported by System.js), and to add decorator support (so we can use @Component, @NgModule, etc). The emitDecoratorMetadata option is required if you want to specify dependencies using class type instead of @Inject() decorator .

1. add a scripts section to our package.json: this will couse the run typescript compiler whenever we write the cmd “ npm run build” .

|  |
| --- |
| "scripts": {  "build": "tsc" }, |

App Skeleton :

1. create the index.html file, which will be the entry point of our application. It will basically load and configure System.js, and then run the bootstrap code of our app.

|  |
| --- |
| <html>  <head>  <title>Hello, Angular</title>  </head>  <body>  <app-main>Loading...</app-main>  <script src="node\_modules/systemjs/dist/system.src.js"></script>  <script src="systemjs.config.js"></script>  <script>  System.import('dist/main.js').catch(function (err) {  console.error(err);  });  </script>  </body> </html> |

1. The <app-main> element is the placeholder where our app will be rendered. We load System.js and its configuration file that we created above, and then instruct System.js to use a load dist/main.js as the entry point for our application.
2. After we have the index file set up, it is time to start creating the actual app code. We will start by creating a very basic Angular component, that says: “Hello, Angular”. This will be an entry point of our application, that is the component that will be rendered when the application loads. We will name the file src/app/app.component.ts, following the Angular style guide naming conventions:

|  |
| --- |
| import { Component } from '[@angular/core](http://twitter.com/angular/core)';  [@Component](http://twitter.com/Component)({  selector: 'app-main',  template: '<h1>Hello, {{name}}</h1>' }) export class AppComponent {  name = 'Angular'; } |

1. As you can see, this is a very basic component — with a simple template, and also a data binding, just to show that Angular actually works in this context. We used app-main as the selector for our component, which is the same as the placeholder element we created in index.html.
2. As you probably know we use decorators in Angular to supply information to the framework. In the example above we use the @Component decorator and pass a decorator descriptor specifying component selector and template
3. we would need to create an Angular module that will bootstrap this component. As explained earlier modules are the Angular way of organizing our applications — each module groups related components, directives and services. We will create the main module of our application, in a file called src/app/app.module.ts

|  |
| --- |
| import { AppComponent } from './app.component'; import { NgModule } from '[@angular/core](http://twitter.com/angular/core)'; import { BrowserModule } from '[@angular/platform-browser](http://twitter.com/angular/platform-browser)';  [@NgModule](http://twitter.com/NgModule)({  imports: [BrowserModule],  declarations: [AppComponent],  bootstrap: [AppComponent] }) export class AppModule { } |

1. this code is purely declarative: we import the BrowserModule, which is required for rendering in browser environment, and then we declare the component we created in the previous step, and finally set it as the bootstrap component — that will be rendered as soon as this module is bootstrapped.
2. Create a file src/main.ts as below content

|  |
| --- |
| import 'core-js/es7/reflect'; import 'zone.js/dist/zone';  import { platformBrowserDynamic }   from '[@angular/platform-browser-dynamic](http://twitter.com/angular/platform-browser-dynamic)'; import { AppModule } from './app/app.module';  platformBrowserDynamic().bootstrapModule(AppModule); |

1. The first two lines import the polyfills that we need — Zone.js and the Reflect polyfill, required by Angular.
2. The last line is where all the magic happens — we ask Angular to bootstrap our module. Since we use platformBrowserDynamic(), angular first invokes the Angular Compiler, which transforms our code into highly optimized code, tuned for high runtime performance

Loading it in the browser

1. So we have our app ready, but we still need to compile it and serve it before we can load it into the browser. So first, let’s compile it by running:

|  |
| --- |
| npm run build |

1. For serving the app, we will use a simple http server called live-server. It has built-in live reload feature, so your application will automatically reload whenever you change one of the source file. Install it by running:

|  |
| --- |
| npm i --save-dev live-server |

1. and then update the scripts section in your package.json file:

|  |
| --- |
| "scripts": {  "build": "tsc",  "start": "live-server" }, |

1. And that’s it! Run the following command to view the app in the browser:

|  |
| --- |
| npm start |

1. If port 8080 is used then we can change it via Node-Inspector for that first install node-inspector by below cmd

|  |
| --- |
| npm install -g node-inspector |

1. You can check if node-inspector is install or not via cmd

|  |
| --- |
| node-inspector |

1. You can change the port via cmd

|  |
| --- |
| node-inspector --web-port=5500 |

1. To check which port in window use we can use below cmd

|  |
| --- |
| netstat -ano | find "8099" |

**Systemjs = >** to enable loading ESM modules into a browser. , module loader , SystemJS supports both CommonJS and AMD module formats and defines its own System.register format.

When tsc compiles typescript into JavaScript, you end up with a bunch of js files on your local system. They somehow need to be loaded into a browser. Since browsers don't support native ES6 module loading yet, you have two options, either put them all into your index.html file in the correct order of dependencies, or you can use a loader to do that all for you. You specify the root for all modules, and then all files are loaded and executed by that loader in the correct order of dependencies. There are many loaders: requirejs, webpack, systemjs and others. In your particular case it's systemjs

Looking at the transpiled javascript of the ts files it shows all import statements are converted into require() statements.

Yes, this is a way for SystemJs to load bundles. It uses require() and exports syntax because that's the CommonJS syntax for loading bundles and you specified this type in your tsconfig.json:

{

"compilerOptions": {

"target": "es5",

"module": "commonjs",

If you were to put module:'es6', you would see that in your compiled javascript files the import and export statements are preserved. However, as mentioned before, you still can't use this syntax as browsers don't support it natively. If you were to put module:'amd', you would see different syntax that uses define(). I guess the systemjs loader is preferred in angular2 starter tutorial since it actually can load all module types supported by tsc. However, if you want to load modules as es6 modules, you have to put module: 'system' in your tsconfig.json. It's a module system designed to adhere to es6 modules standard and used until there's a full support of es6 modules in browsers

**How the setup works**

In your index.html you add the following script:

<script>

System.import('app').catch(function (err) {

console.error(err);

});

</script>

which is executed when index.html is loaded. The import('app') method instructs systemjs to load app module which is mapped to app folder in your project directory structure as specified by the configuration in systemjs.config.js:

map: {

// our app is within the app folder

app: 'app',

SystemJs looks for main.js file in that folder. When app/main.js is found and loaded into a browser, inside it's code the call of require is found:

var app\_module\_1 = require('./app.module');

and systemjs then fetches app.module.js file from local system. This one in turn has its own dependcies, like:

var core\_1 = require('@angular/core');

And the cycle repeats - load, search for dependencie, load them and execute. And this is the way all dependecies are resolved, loaded and executed in a browser by the systemjs.

**Why the mappings to core @angular libraries are required**

In the systemjs.config.ts file there are mapping to the core @angular modules:

map: {

...

// angular bundles

'@angular/core': 'npm:@angular/core/bundles/core.umd.js',

'@angular/common': 'npm:@angular/common/bundles/common.umd.js',

The first thing to understand here is that these are *mappings*, not dependencies. It means that if none of your files import @angular/core, it will not be loaded to a browser. However, you may see that this particular module is imported inside app/app.module.ts:

import { NgModule } from '@angular/core';

Now, why the mappings are there. Suppose systemjs loaded your app/app.module.js into the browser. It parses its content and finds the following:

var core\_1 = require('@angular/core');

Now systemjs understands that it needs to resolve and load @angular/core. It first goes through the process of checking mappings, as specified in the docs:

The map option is similar to paths, but acts very early in the normalization process. It allows you to map a module alias to a location or package.

I would call it a resolution by a named module. So, it finds the mapping and substitutes @angular/core with node\_modules/@angular/core and this is where the real files are placed.

I think systemjs tries to imitate the approach used in node.js where you can specify a module without relative path identifiers ['/', '../', or './'], simply like this require('bar.js') and node.js:

then Node.js starts at the parent directory of the current module, and adds /node\_modules, and attempts to load the module from that location.

If you wanted, you could avoid using named mappings and import by using relative path like this:

import {NgModule} from '../node\_modules/@angular/core';

However, this should be done in all references to @angular.core in the project and lib files, including @angular, which is not a good solution to say the least.

Webpack positions itself as a module bundler for modern JavaScript applications, serving a different purpose than SystemJS, although when it comes to module loading their functionality somewhat overlap. Webpack bundles all modules into one or several chunks — a bunch of modules packaged in a single file. SystemJS can also do that but in this respect it’s much more limited than Webpack. Where they differ the most is when it comes to loading modules dynamically. While SystemJS can load any module dynamically on demand during runtime, Webpack can only dynamically load chunks defined and created during build time.

Typescript compiler : tsconfig.json

| **Option** | **Type** | **Default** | **Description** |
| --- | --- | --- | --- |
| --allowJs | boolean | false | Allow JavaScript files to be compiled. |
| --allowSyntheticDefaultImports | boolean | module === "system"or --esModuleInterop | Allow default imports from modules with no default export. This does not affect code emit, just typechecking. |
| --allowUnreachableCode | boolean | false | Do not report errors on unreachable code. |
| --allowUnusedLabels | boolean | false | Do not report errors on unused labels. |
| --alwaysStrict | boolean | false | Parse in strict mode and emit "use strict" for each source file |
| --baseUrl | string |  | Base directory to resolve non-relative module names. See [Module Resolution documentation](https://www.typescriptlang.org/docs/handbook/module-resolution.html#base-url) for more details. |
| --charset | string | "utf8" | The character set of the input files. |
| --checkJs | boolean | false | Report errors in .js files. Use in conjunction with --allowJs. |
| --declaration -d | boolean | false | Generates corresponding .d.ts file. |
| --declarationDir | string |  | Output directory for generated declaration files. |
| --declarationMap | boolean | false | Generates a sourcemap for each corresponding ‘.d.ts’ file. |
| --diagnostics | boolean | false | Show diagnostic information. |
| --disableSizeLimit | boolean | false | Disable size limitation on JavaScript project. |
| --downlevelIteration | boolean | false | Provide full support for iterables in for..of, spread and destructuring when targeting ES5 or ES3. |
| --emitBOM | boolean | false | Emit a UTF-8 Byte Order Mark (BOM) in the beginning of output files. |
| --emitDeclarationOnly | boolean | false | Only emit ‘.d.ts’ declaration files. |
| --emitDecoratorMetadata[1] | boolean | false | Emit design-type metadata for decorated declarations in source. See [issue #2577](https://github.com/Microsoft/TypeScript/issues/2577) for details. |
| --esModuleInterop | boolean | false | Emit \_\_importStar and \_\_importDefault helpers for runtime babel ecosystem compatibility and enable --allowSyntheticDefaultImports for typesystem compatibility. |
| --experimentalDecorators[1] | boolean | false | Enables experimental support for ES decorators. |
| --forceConsistentCasingInFileNames | boolean | false | Disallow inconsistently-cased references to the same file. |
| --help -h |  |  | Print help message. |
| --importHelpers | boolean | false | Import emit helpers (e.g. \_\_extends, \_\_rest, etc..) from [tslib](https://www.npmjs.com/package/tslib) |
| --inlineSourceMap | boolean | false | Emit a single file with source maps instead of having a separate file. |
| --inlineSources | boolean | false | Emit the source alongside the sourcemaps within a single file; requires --inlineSourceMap or --sourceMap to be set. |
| --init |  |  | Initializes a TypeScript project and creates a tsconfig.json file. |
| --isolatedModules | boolean | false | Transpile each file as a separate module (similar to “ts.transpileModule”). |
| --jsx | string | "Preserve" | Support JSX in .tsx files: "React" or "Preserve". See [JSX](https://www.typescriptlang.org/docs/handbook/jsx.html). |
| --jsxFactory | string | "React.createElement" | Specify the JSX factory function to use when targeting react JSX emit, e.g. React.createElement or h. |
| --keyofStringsOnly | boolean | false | Resolve keyof to string valued property names only (no numbers or symbols). |
| --lib | string[] |  | List of library files to be included in the compilation. Possible values are:  ► ES5  ► ES6  ► ES2015  ► ES7  ► ES2016  ► ES2017  ► ES2018  ► ESNext  ► DOM  ► DOM.Iterable  ► WebWorker  ► ScriptHost  ► ES2015.Core  ► ES2015.Collection  ► ES2015.Generator  ► ES2015.Iterable  ► ES2015.Promise  ► ES2015.Proxy  ► ES2015.Reflect  ► ES2015.Symbol  ► ES2015.Symbol.WellKnown  ► ES2016.Array.Include  ► ES2017.object  ► ES2017.Intl  ► ES2017.SharedMemory  ► ES2017.TypedArrays  ► ES2018.Intl  ► ES2018.Promise  ► ES2018.RegExp  ► ESNext.AsyncIterable  ► ESNext.Array   Note: If --lib is not specified a default list of librares are injected. The default libraries injected are:  ► For --target ES5: DOM,ES5,ScriptHost ► For --target ES6: DOM,ES6,DOM.Iterable,ScriptHost |
| --listEmittedFiles | boolean | false | Print names of generated files part of the compilation. |
| --listFiles | boolean | false | Print names of files part of the compilation. |
| --locale | string | *(platform specific)* | The locale to use to show error messages, e.g. en-us.  Possible values are:  ► English (US): en  ► Czech: cs  ► German: de  ► Spanish: es  ► French: fr  ► Italian: it  ► Japanese: ja  ► Korean: ko  ► Polish: pl  ► Portuguese(Brazil): pt-BR  ► Russian: ru  ► Turkish: tr  ► Simplified Chinese: zh-CN  ► Traditional Chinese: zh-TW |
| --mapRoot | string |  | Specifies the location where debugger should locate map files instead of generated locations. Use this flag if the .map files will be located at run-time in a different location than the .js files. The location specified will be embedded in the sourceMap to direct the debugger where the map files will be located. |
| --maxNodeModuleJsDepth | number | 0 | The maximum dependency depth to search under node\_modules and load JavaScript files. Only applicable with --allowJs. |
| --module -m | string | target === "ES3" or "ES5" ? "CommonJS" : "ES6" | Specify module code generation: "None", "CommonJS", "AMD", "System", "UMD", "ES6", "ES2015" or "ESNext". ► Only "AMD" and "System" can be used in conjunction with --outFile. ► "ES6" and "ES2015" values may be used when targeting "ES5" or lower. |
| --moduleResolution | string | module === "AMD" or "System" or "ES6" ? "Classic" : "Node" | Determine how modules get resolved. Either "Node" for Node.js/io.js style resolution, or "Classic". See [Module Resolution documentation](https://www.typescriptlang.org/docs/handbook/module-resolution.html) for more details. |
| --newLine | string | *(platform specific)* | Use the specified end of line sequence to be used when emitting files: "crlf" (windows) or "lf" (unix).” |
| --noEmit | boolean | false | Do not emit outputs. |
| --noEmitHelpers | boolean | false | Do not generate custom helper functions like \_\_extends in compiled output. |
| --noEmitOnError | boolean | false | Do not emit outputs if any errors were reported. |
| --noErrorTruncation | boolean | false | Do not truncate error messages. |
| --noFallthroughCasesInSwitch | boolean | false | Report errors for fallthrough cases in switch statement. |
| --noImplicitAny | boolean | false | Raise error on expressions and declarations with an implied any type. |
| --noImplicitReturns | boolean | false | Report error when not all code paths in function return a value. |
| --noImplicitThis | boolean | false | Raise error on this expressions with an implied any type. |
| --noImplicitUseStrict | boolean | false | Do not emit "use strict" directives in module output. |
| --noLib | boolean | false | Do not include the default library file (lib.d.ts). |
| --noResolve | boolean | false | Do not add triple-slash references or module import targets to the list of compiled files. |
| --noStrictGenericChecks | boolean | false | Disable strict checking of generic signatures in function types. |
| --noUnusedLocals | boolean | false | Report errors on unused locals. |
| --noUnusedParameters | boolean | false | Report errors on unused parameters. |
| - | string |  | DEPRECATED. Use --outFile instead. |
| --outDir | string |  | Redirect output structure to the directory. |
| --outFile | string |  | Concatenate and emit output to single file. The order of concatenation is determined by the list of files passed to the compiler on the command line along with triple-slash references and imports. See output file order documentation for more details. |
| paths[2] | Object |  | List of path mapping entries for module names to locations relative to the baseUrl. See [Module Resolution documentation](https://www.typescriptlang.org/docs/handbook/module-resolution.html#path-mapping) for more details. |
| --preserveConstEnums | boolean | false | Do not erase const enum declarations in generated code. See [const enums documentation](https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md" \l "94-constant-enum-declarations) for more details. |
| --preserveSymlinks | boolean | false | Do not resolve symlinks to their real path; treat a symlinked file like a real one. |
| --preserveWatchOutput | boolean | false | Keep outdated console output in watch mode instead of clearing the screen |
| --pretty | boolean | true unless piping to another program or redirecting output to a file | Stylize errors and messages using color and context. |
| --project -p | string |  | Compile a project given a valid configuration file. The argument can be a file path to a valid JSON configuration file, or a directory path to a directory containing a tsconfig.json file. See [tsconfig.json](https://www.typescriptlang.org/docs/handbook/tsconfig-json.html) documentation for more details. |
| --reactNamespace | string | "React" | DEPRECATED. Use --jsxFactoryinstead. Specifies the object invoked for createElement and \_\_spread when targeting "react" JSX emit. |
| --removeComments | boolean | false | Remove all comments except copy-right header comments beginning with /\*! |
| --resolveJsonModule | boolean | false | Include modules imported with .jsonextension. |
| --rootDir | string | *(common root directory is computed from the list of input files)* | Specifies the root directory of input files. Only use to control the output directory structure with --outDir. |
| rootDirs[2] | string[] |  | List of *root* folders whose combined content represent the structure of the project at runtime. See [Module Resolution documentation](https://www.typescriptlang.org/docs/handbook/module-resolution.html#virtual-directories-with-rootdirs) for more details. |
| --skipDefaultLibCheck | boolean | false | DEPRECATED. Use --skipLibCheckinstead. Skip type checking of [default library declaration files](https://www.typescriptlang.org/docs/handbook/triple-slash-directives.html#-reference-no-default-libtrue). |
| --skipLibCheck | boolean | false | Skip type checking of all declaration files (\*.d.ts). |
| --sourceMap | boolean | false | Generates corresponding .map file. |
| --sourceRoot | string |  | Specifies the location where debugger should locate TypeScript files instead of source locations. Use this flag if the sources will be located at run-time in a different location than that at design-time. The location specified will be embedded in the sourceMap to direct the debugger where the source files will be located. |
| --strict | boolean | false | Enable all strict type checking options.  Enabling --strict enables --noImplicitAny, --noImplicitThis, --alwaysStrict, --strictNullChecks, --strictFunctionTypes and --strictPropertyInitialization. |
| --strictFunctionTypes | boolean | false | Disable bivariant parameter checking for function types. |
| --strictPropertyInitialization | boolean | false | Ensure non-undefined class properties are initialized in the constructor. This option requires --strictNullChecksbe enabled in order to take effect. |
| --strictNullChecks | boolean | false | In strict null checking mode, the nulland undefined values are not in the domain of every type and are only assignable to themselves and any (the one exception being that undefinedis also assignable to void). |
| --stripInternal[1] | boolean | false | Do not emit declarations for code that has an /\*\* @internal \*/ JSDoc annotation. |
| --suppressExcessPropertyErrors | boolean | false | Suppress excess property checks for object literals. |
| --suppressImplicitAnyIndexErrors | boolean | false | Suppress --noImplicitAny errors for indexing objects lacking index signatures. See [issue #1232](https://github.com/Microsoft/TypeScript/issues/1232#issuecomment-64510362) for more details. |
| --target -t | string | "ES3" | Specify ECMAScript target version: "ES3" (default), "ES5", "ES6"/"ES2015", "ES2016", "ES2017" or "ESNext".   Note: "ESNext" targets latest supported [ES proposed features](https://github.com/tc39/proposals). |
| --traceResolution | boolean | false | Report module resolution log messages. |
| --types | string[] |  | List of names of type definitions to include. See [@types, –typeRoots and –types](https://www.typescriptlang.org/docs/handbook/tsconfig-json.html#types-typeroots-and-types) for more details. |
| --typeRoots | string[] |  | List of folders to include type definitions from. See [@types, –typeRoots and –types](https://www.typescriptlang.org/docs/handbook/tsconfig-json.html#types-typeroots-and-types) for more details. |
| --version -v |  |  | Print the compiler’s version. |
| --watch -w |  |  | Run the compiler in watch mode. Watch input files and trigger recompilation on changes. |