

TypeScript 3.2 is now available. [Download \(/#download-links\)](#) our latest version today!

Documentation ▼

# Module Resolution

This section assumes some basic knowledge about modules. Please see the [Modules \(./modules.html\)](#) documentation for more information.

*Module resolution* is the process the compiler uses to figure out what an import refers to. Consider an import statement like `import { a } from "moduleA"`; in order to check any use of `a`, the compiler needs to know exactly what it represents, and will need to check its definition `moduleA`.

At this point, the compiler will ask “what’s the shape of `moduleA`?” While this sounds straightforward, `moduleA` could be defined in one of your own `.ts` / `.tsx` files, or in a `.d.ts` that your code depends on.

First, the compiler will try to locate a file that represents the imported module. To do so the compiler follows one of two different strategies: Classic or Node. These strategies tell the compiler *where* to look for `moduleA`.

If that didn’t work and if the module name is non-relative (and in the case of `"moduleA"`, it is), then the compiler will attempt to locate an ambient module declaration ([./modules.html#ambient-modules](#)). We’ll cover non-relative imports next.

Finally, if the compiler could not resolve the module, it will log an error. In this case, the error would be something like `error TS2307: Cannot find module 'moduleA'.`

## Relative vs. Non-relative module imports

Module imports are resolved differently based on whether the module reference is relative or non-relative.

A *relative import* is one that starts with `/`, `./` or `../`. Some examples include:

- `import Entry from "../components/Entry";`
- `import { DefaultHeaders } from "../constants/http";`
- `import "/mod";`

Any other import is considered **non-relative**. Some examples include:

- `import * as $ from "jquery";`
- `import { Component } from "@angular/core";`

A relative import is resolved relative to the importing file and *cannot* resolve to an ambient module declaration. You should use relative imports for your own modules that are guaranteed to maintain their relative location at runtime.

A non-relative import can be resolved relative to `baseUrl`, or through path mapping, which we’ll cover below. They can also resolve to ambient module declarations ([./modules.html#ambient-modules](#)). Use non-relative paths when importing any of your external dependencies.

## Module Resolution Strategies

There are two possible module resolution strategies: Node and Classic. You can use the `--moduleResolution` flag to specify the module resolution strategy. If not specified, the default is Classic for `--module AMD | System | ES2015` or Node otherwise.

### *Classic*

This used to be TypeScript's default resolution strategy. Nowadays, this strategy is mainly present for backward compatibility.

A relative import will be resolved relative to the importing file. So `import { b } from "../moduleB"` in source file `/root/src/folder/A.ts` would result in the following lookups:

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`

For non-relative module imports, however, the compiler walks up the directory tree starting with the directory containing the importing file, trying to locate a matching definition file.

For example:

A non-relative import to `moduleB` such as `import { b } from "moduleB"`, in a source file `/root/src/folder/A.ts`, would result in attempting the following locations for locating `"moduleB"`:

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`
3. `/root/src/moduleB.ts`
4. `/root/src/moduleB.d.ts`
5. `/root/moduleB.ts`
6. `/root/moduleB.d.ts`
7. `/moduleB.ts`
8. `/moduleB.d.ts`

### *Node*

This resolution strategy attempts to mimic the Node.js (<https://nodejs.org/>) module resolution mechanism at runtime. The full Node.js resolution algorithm is outlined in Node.js module documentation ([https://nodejs.org/api/modules.html#modules\\_all\\_together](https://nodejs.org/api/modules.html#modules_all_together)).

#### *How Node.js resolves modules*

To understand what steps the TS compiler will follow, it is important to shed some light on Node.js modules. Traditionally, imports in Node.js are performed by calling a function named `require`. The behavior Node.js takes will differ depending on if `require` is given a relative path or a non-relative path.

Relative paths are fairly straightforward. As an example, let's consider a file located at `/root/src/moduleA.js`, which contains the import `var x = require("../moduleB");` Node.js resolves that import in the following order:

1. Ask the file named `/root/src/moduleB.js`, if it exists.
2. Ask the folder `/root/src/moduleB` if it contains a file named `package.json` that specifies a `"main"` module. In our example, if Node.js found the file `/root/src/moduleB/package.json` containing `{ "main": "lib/mainModule.js" }`, then Node.js will refer to `/root/src/moduleB/lib/mainModule.js`.
3. Ask the folder `/root/src/moduleB` if it contains a file named `index.js`. That file is implicitly considered that folder's `"main"` module.

You can read more about this in Node.js documentation on file modules ([https://nodejs.org/api/modules.html#modules\\_file\\_modules](https://nodejs.org/api/modules.html#modules_file_modules)) and folder modules ([https://nodejs.org/api/modules.html#modules\\_folders\\_as\\_modules](https://nodejs.org/api/modules.html#modules_folders_as_modules)).

However, resolution for a non-relative module name is performed differently. Node will look for your modules in special folders named `node_modules`. A `node_modules` folder can be on the same level as the current file, or higher up in the directory chain. Node will walk up the directory chain, looking through each `node_modules` until it finds the module you tried to load.

Following up our example above, consider if `/root/src/moduleA.js` instead used a non-relative path and had the import `var x = require("moduleB");`. Node would then try to resolve `moduleB` to each of the locations until one worked.

1. `/root/src/node_modules/moduleB.js`
2. `/root/src/node_modules/moduleB/package.json` (if it specifies a `"main"` property)
3. `/root/src/node_modules/moduleB/index.js`
4. `/root/node_modules/moduleB.js`
5. `/root/node_modules/moduleB/package.json` (if it specifies a `"main"` property)
6. `/root/node_modules/moduleB/index.js`
7. `/node_modules/moduleB.js`
8. `/node_modules/moduleB/package.json` (if it specifies a `"main"` property)
9. `/node_modules/moduleB/index.js`

Notice that Node.js jumped up a directory in steps (4) and (7).

You can read more about the process in Node.js documentation on loading modules from `node_modules` ([https://nodejs.org/api/modules.html#modules\\_loading\\_from\\_node\\_modules\\_folders](https://nodejs.org/api/modules.html#modules_loading_from_node_modules_folders)).

### *How TypeScript resolves modules*

TypeScript will mimic the Node.js run-time resolution strategy in order to locate definition files for modules at compile-time. To accomplish this, TypeScript overlays the TypeScript source file extensions (`.ts`, `.tsx`, and `.d.ts`) over the Node's resolution logic. TypeScript will also use a field in `package.json` named `"types"` to mirror the purpose of `"main"` - the compiler will use it to find the "main" definition file to consult.

For example, an import statement like `import { b } from "./moduleB"` in `/root/src/moduleA.ts` would result in attempting the following locations for locating `./moduleB`:

1. `/root/src/moduleB.ts`
2. `/root/src/moduleB.tsx`
3. `/root/src/moduleB.d.ts`
4. `/root/src/moduleB/package.json` (if it specifies a `"types"` property)
5. `/root/src/moduleB/index.ts`
6. `/root/src/moduleB/index.tsx`
7. `/root/src/moduleB/index.d.ts`

Recall that Node.js looked for a file named `moduleB.js`, then an applicable `package.json`, and then for an `index.js`.

Similarly a non-relative import will follow the Node.js resolution logic, first looking up a file, then looking up an applicable folder. So `import { b } from "moduleB"` in source file `/root/src/moduleA.ts` would result in the following lookups:

1. `/root/src/node_modules/moduleB.ts`
2. `/root/src/node_modules/moduleB.tsx`
3. `/root/src/node_modules/moduleB.d.ts`
4. `/root/src/node_modules/moduleB/package.json` (if it specifies a `"types"` property)
5. `/root/src/node_modules/@types/moduleB.d.ts`
6. `/root/src/node_modules/moduleB/index.ts`
7. `/root/src/node_modules/moduleB/index.tsx`

8. `/root/src/node_modules/moduleB/index.d.ts`
9. `/root/node_modules/moduleB.ts`
10. `/root/node_modules/moduleB.tsx`
11. `/root/node_modules/moduleB.d.ts`
12. `/root/node_modules/moduleB/package.json` (if it specifies a `"types"` property)
13. `/root/node_modules/@types/moduleB.d.ts`
14. `/root/node_modules/moduleB/index.ts`
15. `/root/node_modules/moduleB/index.tsx`
16. `/root/node_modules/moduleB/index.d.ts`
17. `/node_modules/moduleB.ts`
18. `/node_modules/moduleB.tsx`
19. `/node_modules/moduleB.d.ts`
20. `/node_modules/moduleB/package.json` (if it specifies a `"types"` property)
21. `/node_modules/@types/moduleB.d.ts`
22. `/node_modules/moduleB/index.ts`
23. `/node_modules/moduleB/index.tsx`
24. `/node_modules/moduleB/index.d.ts`

Don't be intimidated by the number of steps here - TypeScript is still only jumping up directories twice at steps (9) and (17). This is really no more complex than what Node.js itself is doing.

### Additional module resolution flags

A project source layout sometimes does not match that of the output. Usually a set of build steps result in generating the final output. These include compiling `.ts` files into `.js`, and copying dependencies from different source locations to a single output location. The net result is that modules at runtime may have different names than the source files containing their definitions. Or module paths in the final output may not match their corresponding source file paths at compile time.

The TypeScript compiler has a set of additional flags to *inform* the compiler of transformations that are expected to happen to the sources to generate the final output.

It is important to note that the compiler will *not* perform any of these transformations; it just uses these pieces of information to guide the process of resolving a module import to its definition file.

### Base URL

Using a `baseUrl` is a common practice in applications using AMD module loaders where modules are "deployed" to a single folder at run-time. The sources of these modules can live in different directories, but a build script will put them all together.

Setting `baseUrl` informs the compiler where to find modules. All module imports with non-relative names are assumed to be relative to the `baseUrl`.

Value of `baseUrl` is determined as either:

- value of `baseUrl` command line argument (if given path is relative, it is computed based on current directory)
- value of `baseUrl` property in `'tsconfig.json'` (if given path is relative, it is computed based on the location of `'tsconfig.json'`)

Note that relative module imports are not impacted by setting the `baseUrl`, as they are always resolved relative to their importing files.

You can find more documentation on `baseUrl` in RequireJS (<http://requirejs.org/docs/api.html#config-baseUrl>) and SystemJS (<https://github.com/systemjs/systemjs/blob/master/docs/config-api.md#baseurl>) documentation.

### Path mapping

Sometimes modules are not directly located under *baseUrl*. For instance, an import to a module "jquery" would be translated at runtime to "node\_modules/jquery/dist/jquery.slim.min.js". Loaders use a mapping configuration to map module names to files at run-time, see RequireJs documentation (<http://requirejs.org/docs/api.html#config-paths>) and SystemJS documentation (<https://github.com/systemjs/systemjs/blob/master/docs/config-api.md#paths>).

The TypeScript compiler supports the declaration of such mappings using "paths" property in `tsconfig.json` files. Here is an example for how to specify the "paths" property for jquery.

```
{
  "compilerOptions": {
    "baseUrl": ".", // This must be specified if "paths" is.
    "paths": {
      "jquery": ["node_modules/jquery/dist/jquery"] // This mapping is relative to "baseUrl"
    }
  }
}
```

Please notice that "paths" are resolved relative to "baseUrl". When setting "baseUrl" to another value than ".", i.e. the directory of `tsconfig.json`, the mappings must be changed accordingly. Say, you set "baseUrl": "../src" in the above example, then jquery should be mapped to "../node\_modules/jquery/dist/jquery".

Using "paths" also allows for more sophisticated mappings including multiple fall back locations. Consider a project configuration where only some modules are available in one location, and the rest are in another. A build step would put them all together in one place. The project layout may look like:

```
projectRoot
├── folder1
│   ├── file1.ts (imports 'folder1/file2' and 'folder2/file3')
│   └── file2.ts
├── generated
│   ├── folder1
│   └── folder2
│       └── file3.ts
└── tsconfig.json
```

The corresponding `tsconfig.json` would look like:

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

This tells the compiler for any module import that matches the pattern "" (i.e. all values), to look in two locations:

1. "" : meaning the same name unchanged, so map `<moduleName> => <baseUrl>/<moduleName>`
2. "generated/" meaning the module name with an appended prefix "generated", so map `<moduleName> => <baseUrl>/generated/<moduleName>`

Following this logic, the compiler will attempt to resolve the two imports as such:

- import 'folder1/file2'
  1. pattern '\*' is matched and wildcard captures the whole module name
  2. try first substitution in the list: '\*' -> folder1/file2
  3. result of substitution is non-relative name - combine it with *baseUrl* -> projectRoot/folder1/file2.ts.
  4. File exists. Done.
- import 'folder2/file3'
  1. pattern '\*' is matched and wildcard captures the whole module name
  2. try first substitution in the list: '\*' -> folder2/file3
  3. result of substitution is non-relative name - combine it with *baseUrl* -> projectRoot/folder2/file3.ts.
  4. File does not exist, move to the second substitution
  5. second substitution 'generated/\*' -> generated/folder2/file3
  6. result of substitution is non-relative name - combine it with *baseUrl* -> projectRoot/generated/folder2/file3.ts.
  7. File exists. Done.

#### Virtual Directories with *rootDirs*

Sometimes the project sources from multiple directories at compile time are all combined to generate a single output directory. This can be viewed as a set of source directories create a "virtual" directory.

Using 'rootDirs', you can inform the compiler of the *roots* making up this "virtual" directory; and thus the compiler can resolve relative modules imports within these "virtual" directories *as if* were merged together in one directory.

For example consider this project structure:

```
src
├─ views
│   ├─ view1.ts (imports './template1')
│   └─ view2.ts
└─ generated
    └─ templates
        └─ views
            └─ template1.ts (imports './view2')
```

Files in `src/views` are user code for some UI controls. Files in `generated/templates` are UI template binding code auto-generated by a template generator as part of the build. A build step will copy the files in `/src/views` and `/generated/templates/views` to the same directory in the output. At run-time, a view can expect its template to exist next to it, and thus should import it using a relative name as `"./template"`.

To specify this relationship to the compiler, use `"rootDirs"`. `"rootDirs"` specify a list of *roots* whose contents are expected to merge at run-time. So following our example, the `tsconfig.json` file should look like:

```
{
  "compilerOptions": {
    "rootDirs": [
      "src/views",
      "generated/templates/views"
    ]
  }
}
```

Every time the compiler sees a relative module import in a subfolder of one of the `rootDirs`, it will attempt to look for this import in each of the entries of `rootDirs`.

The flexibility of `rootDirs` is not limited to specifying a list of physical source directories that are logically merged. The supplied array may include any number of ad hoc, arbitrary directory names, regardless of whether they exist or not. This allows the compiler to capture sophisticated bundling and runtime features such as conditional inclusion and project specific loader plugins in a type safe way.

Consider an internationalization scenario where a build tool automatically generates locale specific bundles by interpolating a special path token, say `{locale}`, as part of a relative module path such as `./#{locale}/messages`. In this hypothetical setup the tool enumerates supported locales, mapping the abstracted path into `./zh/messages`, `./de/messages`, and so forth.

Assume that each of these modules exports an array of strings. For example `./zh/messages` might contain:

```
export default [
  "您好吗",
  "很高兴认识你"
];
```

By leveraging `rootDirs` we can inform the compiler of this mapping and thereby allow it to safely resolve `./#{locale}/messages`, even though the directory will never exist. For example, with the following `tsconfig.json`:

```
{
  "compilerOptions": {
    "rootDirs": [
      "src/zh",
      "src/de",
      "src/#{locale}"
    ]
  }
}
```

The compiler will now resolve `import messages from './#{locale}/messages'` to import messages from `./zh/messages` for tooling purposes, allowing development in a locale agnostic manner without compromising design time support.

## Tracing module resolution

As discussed earlier, the compiler can visit files outside the current folder when resolving a module. This can be hard when diagnosing why a module is not resolved, or is resolved to an incorrect definition. Enabling the compiler module resolution tracing using `--traceResolution` provides insight in what happened during the module resolution process.

Let's say we have a sample application that uses the `typescript` module. `app.ts` has an import like `import * as ts from "typescript"`.

```

|   tsconfig.json
|   └── node_modules
|       └── typescript
|           └── lib
|               └── typescript.d.ts
|   └── src
|       └── app.ts

```

Invoking the compiler with `--traceResolution`

```
tsc --traceResolution
```

Results in an output such as:

```

===== Resolving module 'typescript' from 'src/app.ts'. =====
Module resolution kind is not specified, using 'NodeJs'.
Loading module 'typescript' from 'node_modules' folder.
File 'src/node_modules/typescript.ts' does not exist.
File 'src/node_modules/typescript.tsx' does not exist.
File 'src/node_modules/typescript.d.ts' does not exist.
File 'src/node_modules/typescript/package.json' does not exist.
File 'node_modules/typescript.ts' does not exist.
File 'node_modules/typescript.tsx' does not exist.
File 'node_modules/typescript.d.ts' does not exist.
Found 'package.json' at 'node_modules/typescript/package.json'.
'package.json' has 'types' field './lib/typescript.d.ts' that references 'node_modules/typescript/
lib/typescript.d.ts'.
File 'node_modules/typescript/lib/typescript.d.ts' exist - use it as a module resolution result.
===== Module name 'typescript' was successfully resolved to 'node_modules/typescript/lib/typesc
ript.d.ts'. =====

```

#### Things to look out for

- Name and location of the import

```
===== Resolving module 'typescript' from 'src/app.ts'. =====
```

- The strategy the compiler is following

Module resolution kind is not specified, using **'NodeJs'**.

- Loading of types from npm packages

'package.json' has **'types'** field './lib/typescript.d.ts' that references 'node\_modules/typescript/lib/typescript.d.ts'.

- Final result

```
===== Module name 'typescript' was successfully resolved to 'node_modules/typescript/lib/typescript.d.ts'. =====
```

#### Using `--noResolve`

Normally the compiler will attempt to resolve all module imports before it starts the compilation process. Every time it successfully resolves an `import` to a file, the file is added to the set of files the compiler will process later on.

The `--noResolve` compiler options instructs the compiler not to "add" any files to the compilation that were not passed on the command line. It will still try to resolve the module to files, but if the file is not specified, it will not be included.



For instance:

*app.ts*

```
import * as A from "moduleA" // OK, 'moduleA' passed on the command-line
import * as B from "moduleB" // Error TS2307: Cannot find module 'moduleB'.
```

```
tsc app.ts moduleA.ts --noResolve
```

Compiling `app.ts` using `--noResolve` should result in:

- Correctly finding `moduleA` as it was passed on the command-line.
- Error for not finding `moduleB` as it was not passed.

## Common Questions


*Why does a module in the exclude list still get picked up by the compiler?*

`tsconfig.json` turns a folder into a “project”. Without specifying any “`exclude`” or “`files`” entries, all files in the folder containing the `tsconfig.json` and all its sub-directories are included in your compilation. If you want to exclude some of the files use “`exclude`”, if you would rather specify all the files instead of letting the compiler look them up, use “`files`”.

That was `tsconfig.json` automatic inclusion. That does not embed module resolution as discussed above. If the compiler identified a file as a target of a module import, it will be included in the compilation regardless if it was excluded in the previous steps.

So to exclude a file from the compilation, you need to exclude it and **all** files that have an `import` or `/// <reference path="..." />` directive to it.

Made with ❤ in Redmond Follow @Typescriptlang (<https://twitter.com/typescriptlang>)

Privacy (<https://go.microsoft.com/fwlink/?LinkId=521839>) ©2012-2018 Microsoft  Microsoft