Chapter 12. Building and Running TypeScript

If you've deployed and run a JavaScript application in production, then you know how to run a TypeScript application too—once you compile it to JavaScript, the two aren't so different. This chapter is about productionizing and building TypeScript applications, but there isn't much here that's unique to TypeScript apps—it mostly applies to JavaScript applications too. We'll divide it up into four sections, covering:

- The things you have to do to build any TypeScript application
- Building and running TypeScript applications on the server
- Building and running TypeScript applications in the browser
- Building for and publishing your TypeScript application to NPM

Building Your TypeScript Project

Building a TypeScript project is straightforward. In this section, we'll cover the core ideas you'll need to understand in order to build your project for whatever environment you plan to run it in.

Project Layout

I suggest keeping your source TypeScript code in a top-level *src/* folder, and compiling it to a top-level *dist/* folder. This folder structure is a popular convention, and splitting your source code and generated code into two top-level folders can make your life easier down the line, when you're integrating with other tooling. It also makes it easier to exclude generated artifacts from source control.

Try to stick to this convention when you can:

```
my-app/
├──dist/
```

```
│ ├──index.d.ts
│ ├──index.js
│ └──services/
│   ├──foo.d.ts
│   ├──foo.js
│   ├──bar.d.ts
│   └──bar.js
├──src/
│ ├──index.ts
│ └──services/
│   ├──foo.ts
│   └──bar.ts
```

Artifacts

When you compile a TypeScript program to JavaScript, there are a few different artifacts that TSC can generate for you (Table 12-1).

Table 12-1. Artifacts that TSC can generate for you

Type	File extension	tsconfig.json flag	Emitted by default?
JavaScript	*.js*	`{"emitDeclarationOnly": false}`	Yes
Source maps	*.js.map*	`{"sourceMap": true}`	No
Type declarations	*.d.ts*	`{"declaration": true}`	No
Declaration maps	*.d.ts.map*	`{"declarationMap": true}`	No

The first type of artifact—JavaScript files—should be familiar. TSC compiles your TypeScript code to JavaScript that you can then run using a JavaScript platform like NodeJS or Chrome. If you run `tsc yourfile.ts`, TSC will typecheck *yourfile.ts* and compile it to JavaScript.

The second type of artifact—source maps—is special files that link each piece of your generated JavaScript back to the specific line and column of the TypeScript file that it was generated from. This is helpful for debugging your code (Chrome DevtTools will show your TypeScript code, instead of the generated JavaScript), and for mapping lines and columns in JavaScript exception stack traces back to TypeScript (tools like those mentioned in "Error Monitoring" do this lookup automatically if you give them your source maps).

The third artifact—type declarations—lets other TypeScript projects take advantage of your generated types.

Finally, declaration maps are used to speed up compilation times for your TypeScript projects. You'll read more about them in "Project References". We'll spend the rest of this chapter talking about how and why to generate these artifacts.

Dialing In Your Compile Target

JavaScript can be an unusual language to work with: not only does it have a quickly evolving specification with a yearly release cycle, but, as a programmer, you can't always control which JavaScript version the platform you're running your program on implements. On top of that, many JavaScript programs are *isomorphic*, meaning you can run them on either the server or the client. For example:

- If you run your backend JavaScript program on a server that you control, then you can control exactly which JavaScript version it will run on.
- If you then release your backend JavaScript program as an open source project, you don't know which JavaScript version will be supported by your consumers' JavaScript platforms. The best you can do in a NodeJS environment is declare a range of supported NodeJS versions, but in a browser environment you're out of luck.
- If you run your JavaScript in a browser, you have no idea which browser people will use to run it—the latest Chrome, Firefox, or Edge that supports most modern JavaScript features, a slightly outdated version of one of those browsers that's missing some bleeding-edge functionality, an antiquated browser like Internet Explorer 8, or an embedded browser like the one that runs on the PlayStation 4 in your garage. The best you can do is define a minimum set of features that people's browsers need to support to run your application, ship polyfills for as many of those features as you

can, and try to detect when users are on really old browsers that your app won't run on and show them a message saying that they need to upgrade.
- If you release an isomorphic JavaScript library (e.g., a logging library that runs on both browser and server), then you have to support both a minimum NodeJS version and a swath of browser JavaScript engines and versions.

Not every JavaScript environment supports every JavaScript feature out of the box, but you should still try to write code in the latest language version. There are two ways to do this:

1. *Transpile* (i.e., automatically convert) applications from the latest version of JavaScript to the oldest JavaScript version that a platform you target supports. We do this for language features like `for..of` loops and `async` / `await`, which can be automatically converted to `for` loops and `.then` calls, respectively.
2. *Polyfill* (i.e., provide implementations for) any modern features that are missing in the JavaScript runtime you're running on. We do this for features provided by the JavaScript standard library (like `Promise`, `Map`, and `Set`) and for prototype methods (like `Array.prototype.includes` and `Function.prototype.bind`).

TSC has built-in support for transpiling your code to older JavaScript versions, but it will not automatically polyfill your code. This is worth reiterating: TSC will transpile most JavaScript features for older environments, but it will not provide implementations for missing features.

TSC gives you three settings to dial in which environments you want to target:

- `target` sets the JavaScript version you want to transpile to: `es5`, `es2015`, etc.
- `module` sets the module system you want to target: `es2015` modules, `commonjs` modules, `systemjs` modules, etc.
- `lib` tells TypeScript which JavaScript features are available in the environments you're targeting: `es5` features, `es2015` features, the `dom`, etc. It doesn't actually implement these features—that's what polyfills are for—but it does tell TypeScript that the features are available (either natively or via a polyfill).

The environment you plan to run your application in dictates which JavaScript version you should transpile to with `target` and what to set `lib` to. If you're not sure, `es5` is usually a safe default for both. What you set `module` to depends on whether you're targeting a NodeJS or browser environment, and what module loader you're using if the latter.

TIP

If you need to support an unusual set of platforms, look up which JavaScript features your target platforms support natively in Juriy Zaytsev's (aka Kangax's) compatibility tables.

Let's dig a little deeper into `target` and `lib`; we'll leave `module` to the sections on "Running TypeScript on the Server" and "Running TypeScript in the Browser".

target

TSC's built-in transpiler supports converting most JavaScript features to older JavaScript versions, meaning you can write your code in the latest TypeScript version and transpile it down to whatever JavaScript version you need to support. Since TypeScript supports the latest JavaScript features (like `async` / `await`, which is not yet supported by all major JavaScript platforms at the time of writing), you'll almost always find yourself taking advantage of this built-in transpiler to convert your code to something that NodeJS and browsers understand today.

Let's take a look at which specific JavaScript features TSC does and does not transpile for older JavaScript versions (Table 12-2 and Table 12-3).[1]

NOTE

In the past, there was a new revision of the JavaScript language released every few years, with an incrementing language version (ES1, ES3, ES5, ES6). As of 2015, the JavaScript language now has a yearly release cycle, with each language version named after the year it's released in (ES2015, ES2016, and so on). Some JavaScript features, however, get TypeScript support before they're actually slated for a specific JavaScript version; we refer to these features as "ESNext" (as in, the next revision).

Table 12-2. TSC does transpile

Version	Feature
ES2015	`const`, `let`, `for..of` loops, array/object spread (`...`), tagged template strings, classes, generators, arrow functions, function default parameters, function rest parameters, destructuring declarations/assignments/parameters
ES2016	Exponentiation operator (`**`)
ES2017	`async` functions, `await` ing promises
ES2018	`async` iterators
ES2019	Optional parameter in catch clause
ESNext	Numeric separators (`123_456`)

Table 12-3. TSC does not transpile

Version	Feature
ES5	Object getters/setters
ES2015	Regex `y` and `u` flags
ES2018	Regex `s` flag
ESNext	BigInt (`123n`)

To set the transpilation target, pop open your *tsconfig.json* and set the `target` field to:

- `es3` for ECMAScript 3
- `es5` for ECMAScript 5 (this is a good default if you're not sure what to use)
- `es6` or `es2015` for ECMAScript 2015
- `es2016` for ECMAScript 2016
- `es2017` for ECMAScript 2017

- `es2018` for ECMAScript 2018
- `esnext` for whatever the most recent ECMAScript revision is

For example, to compile to ES5:

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

lib

As I mentioned, there's one hitch with transpiling your code to older JavaScript versions: while most language features can be safely transpiled (`let` to `var` , `class` to `function`), you still need to *polyfill* functionality yourself if your target environment doesn't support a newer library feature. Some examples are utilities like `Promise` and `Reflect` , and data structures like `Map` , `Set` , and `Symbol` . When targeting a bleeding-edge environment like the latest Chrome, Firefox, or Edge, you usually won't need any polyfills; but if you're targeting browsers a few versions back—or most NodeJS environments—you will need to polyfill missing features.

Thankfully, you won't need to write polyfills yourself. Instead, you can install them from a popular polyfill library like `core-js` , or add polyfills to your code automatically by running your typechecked TypeScript code through Babel with `@babel/polyfill` .

TIP

If you plan to run your application in a browser, be careful not to bloat the size of your JavaScript bundle by including every single polyfill regardless of whether or not the browser you're running your code in actually needs it—your target platform probably already supports some of the features you're polyfilling. Instead, use a service like Polyfill.io to load just those polyfills that your user's browser needs.

Once you've added polyfills to your code, it's time to tell TSC that your environment is guaranteed to support the features you polyfilled—enter your

tsconfig.json's `lib` field. For example, you could use this configuration if you've polyfilled all ES2015 features plus ES2016's `Array.prototype.includes`:

```
{
  "compilerOptions": {
    "lib": ["es2015", "es2016.array.includes"]
  }
}
```

If you're running your code in the browser, also enable DOM type declarations for things like `window`, `document`, and all the other APIs you get when running your JavaScript in the browser:

```
{
  "compilerOptions": {
  "lib": ["es2015", "es2016.array.include", "dom"]
  }
}
```

For a full list of supported libs run `tsc --help`.

Enabling Source Maps

Source maps are a way to link your transpiled code back to the source code it was generated from. Most developer tools (like Chrome DevTools), error reporting and logging frameworks, and build tools know about source maps. Since a typical build pipeline can produce code that's very different from the code you started with (for example, your pipeline might compile TypeScript to ES5 JavaScript, tree-shake it with Rollup, preevaluate it with Prepack, then minify it with Uglify), using source maps throughout your build pipeline can make it a lot easier to debug the resulting JavaScript.

It's generally a good idea to use source maps in development, and ship source maps to production in both browser and server environments. There's one caveat, though: if you rely on some level of security through obscurity for your browser code, don't ship source maps to browsers in production.

Project References

As your application grows, it will take longer and longer for TSC to typecheck and compile your code. This time grows roughly linearly with the size of your codebase. When developing locally, slow incremental compile times can seriously slow down your development, and make working with TypeScript painful.

To address this, TSC comes with a feature called *project references* that speeds up compilation times dramatically, including incremental compile times. For any project with a few hundred files or more, project references are a must-have.

Use them like this:

1. Split your TypeScript project into multiple projects. A project is simply a folder that contains a *tsconfig.json* and some TypeScript code. Try to split your code in such a way that code that tends to be updated together lives in the same folder.
2. In each project folder, create a *tsconfig.json* that includes at least:

```
{
  "compilerOptions": {
    "composite": true,
    "declaration": true,
    "declarationMap": true,
    "rootDir": "."
  },
  "include": [
    "./**/*.ts"
  ],
  "references": [
    {
      "path": "../myReferencedProject",
      "prepend": true
    }
  ],
}
```

The keys here are:

1. `composite` , which tells TSC that this folder is a subproject of a larger TypeScript project.

2. `declaration`, which tells TSC to emit *.d.ts* declaration files for this project. The way project references work, projects have access to each other's declaration files and emitted JavaScript, but not their source TypeScript files. This creates a boundary beyond which TSC won't try to retypecheck or recompile your code: if you update a line of code in your subproject *A*, TSC doesn't have to retypecheck your other subproject *B*; all TSC needs to check for a type error is *B*'s type declarations. This is the core behavior that makes project references so efficient at rebuilding big projects.
3. `declarationMap`, which tells TSC to build source maps for generated type declarations.
4. `references`, which is an array of subprojects that your subproject depends on. Each reference's `path` should point either to a folder that contains a *tsconfig.json*, or directly to a TSC configuration file (if your configuration file isn't named *tsconfig.json*). `prepend` will concatenate the JavaScript and source maps generated by the subproject you're referencing to the JavaScript and source maps generated by your subproject. Note that `prepend` is only useful when you're using `outFile`—if you don't use `outFile`, you can ditch the `prepend`.
5. `rootDir`, which explicitly specifies that this subproject should be compiled relative to the root project (`.`). Alternatively, you can specify an `outDir` that's a subfolder of the root project's `outDir`.

3. Create a root *tsconfig.json* that references any subprojects that aren't yet referenced by another subproject:

```
{
  "files": [],
  "references": [
    {"path": "./myProject"},
    {"path": "./mySecondProject"}
  ]
}
```

4. Now when you compile your project with TSC, use the `build` flag to tell TSC to take project references into account:

```
tsc --build # Or, tsc -b for short
```

WARNING

At the time of writing, project references are a new TypeScript feature with some rough edges. When using them, be careful to:

- Rebuild the entire project (with `tsc -b`) after cloning or refetching it, in order to regenerate any missing or outdated *.d.ts* files. Alternatively, check in your generated *d.ts* files.
- Not use `noEmitOnError: false` with project references—TSC will always hardcode the option to `true`.
- Manually make sure that a given subproject isn't prepended by more than one other subproject. Otherwise, the doubly prepended subproject will show up twice in your compiled output. Note that if you're just referencing and not prepending, you're good to go.

USING EXTENDS TO REDUCE TSCONFIG.JSON BOILERPLATE

Because you probably want all of your subprojects to share the same compiler options, it's convenient to create a "base" *tsconfig.json* in your root directory that subprojects' *tsconfig.json*s can extend:

```
{
  "compilerOptions": {
    "composite": true,
    "declaration": true,
    "declarationMap": true,
    "lib": ["es2015", "es2016.array.include"],
    "rootDir": ".",
    "sourceMap": true,
    "strict": true,
    "target": "es5",
  }
}
```

Then, update your subprojects to extend it using *tsconfig.json*'s `extends` option:

```
{
  "extends": "../tsconfig.base",
  "include": [
    "./**/*.ts"
  ],
  "references": [
    {
      "path": "../myReferencedProject",
      "prepend": true
    }
  ],
}
```

Error Monitoring

TypeScript warns you about errors at compile time, but you also need a way to find out about exceptions that your users experience at runtime, so that you can try to prevent them at compile time (or at least fix the bug that caused the

runtime error). Use an error monitoring tool like Sentry or Bugsnag to report and collate your runtime exceptions.

Running TypeScript on the Server

To run your TypeScript code in a NodeJS environment, just compile your code to ES2015 JavaScript (or ES5, if you're targeting a legacy NodeJS version) with your *tsconfig.json*'s module flag set to `commonjs`:

```
{
  "compilerOptions": {
    "target": "es2015",
    "module": "commonjs"
  }
}
```

That will compile your ES2015 `import` and `export` calls to `require` and `module.exports`, respectively, so your code will run on NodeJS with no further bundling needed.

If you're using source maps (you should be!), you'll need to feed your source maps into your NodeJS process. Just grab the `source-map-support` package from NPM, and follow the package's setup instructions. Most process monitoring, logging, and error reporting tools like PM2, Winston, and Sentry have built-in support for source maps.

Running TypeScript in the Browser

Compiling TypeScript to run in the browser involves a little more work than running TypeScript on the server.

First, pick a module system to compile to. A good rule of thumb is to stick to `umd` when publishing a library for others to use (e.g., on NPM) in order to maximize compatibility with various module bundlers that people might use in their projects.

If you just plan to use your code yourself without publishing it to NPM, which format you compile to depends on the module bundler you're using. Check

your bundler's documentation—for example, Webpack and Rollup work best with ES2015 modules, while Browserify requires CommonJS modules. Here are a few guidelines:

- If you're using the SystemJS module loader, set `module` to `systemjs`.
- If you're running your code through an ES2015-aware module bundler like Webpack or Rollup, set `module` to `es2015` or higher.
- If you're using an ES2015-aware module bundler and your code uses dynamic imports (see "Dynamic Imports"), set `module` to `esnext`.
- If you're building a library for other projects to use, and aren't running your code through any additional build steps after `tsc`, maximize compatibility with different loaders that people use by setting `module` to `umd`.
- If you're bundling your module with a CommonJS bundler like Browserify, set `module` to `commonjs`.
- If you're planning to load your code with RequireJS or another AMD module loader, set `module` to `amd`.
- If you want your top-level exports to be globally available on the `window` object (as you might if you're Mussolini's great-nephew), set `module` to `none`. Note that TSC will try to curb your enthusiasm for inflicting pain on other software engineers by compiling to `commonjs` anyway if your code is in module mode (see "Module Mode Versus Script Mode").

Next, configure your build pipeline to compile all your TypeScript to a single JavaScript file (usually called a "bundle") or a set of JavaScript files. While TSC can do this for you for small projects with the `outFile` TSC flag, the flag is limited to generating SystemJS and AMD bundles. And since TSC doesn't support build plugins and intelligent code splitting the same way that a dedicated build tool like Webpack does, you'll soon find yourself wanting a more powerful bundler.

That's why for frontend projects, you should use a more powerful build tool from the beginning. There are TypeScript plugins for whatever build tool you might be using, such as:

- `ts-loader` for Webpack
- `tsify` for Browserify
- `@babel/preset-typescript` for Babel
- `gulp-typescript` for Gulp

- `grunt-ts` for Grunt

While a full discussion of optimizing your JavaScript bundle for fast loading is outside the scope of this book, some brief advice—not specific to TypeScript—is:

- Keep your code modular, and avoid implicit dependencies in your code (these can happen when you assign things to the `window` global, or to other globals), so that your build tool can more accurately analyze your project's dependency graph.
- Use dynamic imports to lazy-load code that you don't need for your initial page load, so you don't unnecessarily block your page from rendering.
- Take advantage of your build tool's automatic code splitting functionality, so that you avoid loading too much JavaScript and slowing page load unnecessarily.
- Have a strategy for measuring page load time, either synthetically or, ideally, with real user data. As your app grows the initial load time can get slower and slower; you can only optimize that load time if you have a way to measure it. Tools like New Relic and Datadog are invaluable here.
- Keep your production build as similar as possible to your development build. The more the two diverge, the more hard-to-fix bugs you'll have that only show up in production.
- Finally, when shipping TypeScript to run in the browser, have a strategy for polyfilling missing browser features. This might be a standard set of polyfills you ship as part of every bundle, or it might be a dynamic set of polyfills based on what features the user's browser supports.

Publishing Your TypeScript Code to NPM

It's easy to compile your TypeScript code so that other TypeScript and JavaScript projects can use it. There are a few best practices to keep in mind when compiling to JavaScript for external use:

- Generate source maps, so you can debug your own code.
- Compile to ES5, so that others can easily build and run your code.
- Be mindful about which module format you compile to (UMD, CommonJS, ES2015, etc.).

- Generate type declarations, so that other TypeScript users have types for your code.

Start by compiling your TypeScript to JavaScript with `tsc`, and generate corresponding type declarations. Be sure to configure your *tsconfig.json* to maximize compatibility with popular JavaScript environments and build systems (more on that in "Building Your TypeScript Project"):

```
{
"compilerOptions": {
  "declaration": true,
  "module": "umd",
  "sourceMaps": true,
  "target": "es5"
  }
}
```

Next, blacklist your TypeScript source code from getting published to NPM in your *.npmignore*, to avoid bloating the size of your package. And in your *.gitignore*, exclude generated artifacts from your Git repository to avoid polluting it:

```
# .npmignore

*.ts # Ignore .ts files
!*.d.ts # Allow .d.ts files


# .gitignore

*.d.ts # Ignore .d.ts files
*.js # Ignore .js files
```

NOTE

If you stuck with the recommended project layout and kept your source files in *src/* and your generated files in *dist/*, your *.ignore* files will be even simpler:

```
# .npmignore

src/ # Ignore source files


# .gitignore

dist/ # Ignore generated files
```

Finally, add a `"types"` field to your project's *package.json* to indicate that it comes with type declarations (note that this isn't mandatory, but it is a helpful hint to TSC for any consumers that use TypeScript), and add a script to build your package before publishing it, to make sure that your package's JavaScript, type declarations, and source maps are always up to date and in sync with the TypeScript you compiled them from:

```
{
  "name": "my-awesome-typescript-project",
  "version": "1.0.0",
  "main": "dist/index.js",
  "types": "dist/index.d.ts",
  "scripts": {
    "prepublishOnly": "tsc -d"
  }
}
```

That's it! Now when you `npm publish` your package to NPM, NPM will automatically compile your TypeScript to a format usable by both people that use TypeScript (with full type safety) and people that use JavaScript (with some type safety, if their code editor supports it).

Triple-Slash Directives

TypeScript comes with a little-known, rarely used, and mostly outdated feature called *triple-slash directives*. These directives are specially formatted

TypeScript comments that serve as instructions to TSC.

They come in a few flavors, and in this section, we'll cover just two of them: `types`, for eliding type-only full-module imports, and `amd-module`, for naming generated AMD modules. For a full reference, see Appendix E.

The types Directive

When you import something from a module, depending on what you imported, TypeScript won't always need to generate an `import` or `require` call when you compile your code to JavaScript. If you have an `import` statement whose export is only used in a type position in your module (i.e., you just imported a type from a module), TypeScript won't generate any JavaScript code for that `import`—think of it as only existing at the type level. This feature is called *import elision*.

The exception to the rule is imports used for side effects: if you import an entire module (without importing a specific export or a wildcard from that module), that import will generate JavaScript code when you compile your TypeScript. You might do this, for instance, if you want to make sure that an ambient type defined in a script-mode module is available in your program (like we did in "Safely Extending the Prototype"). For example:

```
// global.ts
type MyGlobal = number

// app.ts
import './global'
```

After compiling *app.ts* to JavaScript with `tsc app.ts`, you'll notice that that the `./global` import wasn't elided:

```
// app.js
import './global'
```

If you find yourself writing imports like this, you may want to start by making sure that your import really needs to use side effects, and that there isn't some other way to rewrite your code to make it more explicit which value or type you're importing (e.g., `import {MyType} from './global'`—which TypeScript will elide for you—instead of `import './global'`). Or, see if

you can include your ambient type in your *tsconfig.json*'s `types`, `files`, or `include` field and avoid the import altogether.

If neither of those works for your use case, and you want to continue to use a full-module import but avoid generating a JavaScript `import` or `require` call for that import, use the `types` triple-slash directive. A triple-slash directive is three slashes `///` followed by one of a few possible XML tags, each with its own set of required attributes. For the `types` directive, it looks like this:

- Declare a dependency on an ambient type declaration:

```
/// <reference types="./global" />
```

- Declare a dependency on *@types/jasmine/index.d.ts*:

```
/// <reference types="jasmine" />
```

You probably won't find yourself using this directive often. And if you do, you may want to rethink how you're using types in your project, and consider if there's a way to rely less on ambient types.

The amd-module Directive

When compiling your TypeScript code to the AMD module format (indicated with `{"module": "amd"}` in your *tsconfig.json*), TypeScript will generate anonymous AMD modules by default. You can use the AMD triple-slash directive to give your emitted modules names.

Let's say you have the following code:

```
export let LogService = {
  log() {
    // ...
  }
}
```

Compiling to the `amd` module format, TSC generates the following JavaScript code:

```
define(['require', 'exports'], function(require, export
  exports.__esModule = true
  exports.LogService = {
    log() {
      // ...
    }
  }
})
```

If you're familiar with the AMD module format, you might have noticed that this is an anonymous AMD module. To give your AMD module a name, use the `amd-module` triple-slash directive in your code:

```
/// <amd-module name="LogService" /> ❶

export let LogService = { ❷

  log() {
    // ...
  }
}
```

❶ We use the `amd-module` directive, and set a `name` attribute on it.

❷ The rest of our code is unchanged.

Recompiling to the AMD module format with TSC, we now get the following JavaScript:

```
/// <amd-module name='LogService' />
define('LogService', ['require', 'exports'], function(r
  exports.__esModule = true
  exports.LogService = {
    log() {
      // ...
    }
  }
})
```

When compiling to AMD modules, use the `amd-module` directive to make your code easier to bundle and debug (or, switch to a more modern module format like ES2015 modules if you can).

Summary

In this chapter we covered everything you need to know to build and run your TypeScript application in production, either in the browser or on the server. We discussed how to choose a JavaScript version to compile to, which libraries to mark as available in your environment (and how to polyfill libraries when they're missing), and how to build and ship source maps with your application to make it easier to debug in production and develop locally. We then explored how to modularize your TypeScript project to keep compilation times fast. Finally, we finished up with how to run your TypeScript application on the server and in the browser, how to publish your TypeScript code to NPM for others to use, how import elision works, and—for AMD users—how to use triple-slash directives to name your modules.

1 If you use a language feature that TSC doesn't transpile and your target environment doesn't support it either, you can usually find a Babel plugin to transpile it for you. To find the most up-to-date plugin, search for "babel plugin <feature name>" in your favorite search engine.