Modules: TypeScript

▶ History

Stability: 1 .1 - Active development

Enabling

There are two ways to enable runtime TypeScript support in Node.js:

1. For full support of all of TypeScript's syntax and features, including using any version of TypeScript, use a third-party package.
2. For lightweight support, you can use the built-in support for type stripping.

Full TypeScript support

To use TypeScript with full support for all TypeScript features, including `tsconfig.json`, you can use a third-party package. These instructions use `tsx` as an example but there are many other similar libraries available.

1. Install the package as a development dependency using whatever package manager you're using for your project. For example, with `npm`:

```
npm install --save-dev tsx
```
COPY

2. Then you can run your TypeScript code via:

```
npx tsx your-file.ts
```
COPY

Or alternatively, you can run with `node` via:

```
node --import=tsx your-file.ts
```
COPY

Type stripping

Added in: v22.6.0

Stability: 1 .1 - Active development

The flag `--experimental-strip-types` enables Node.js to run TypeScript files. By default Node.js will execute only files that contain no TypeScript features that require transformation, such as enums or namespaces. Node.js will replace inline type annotations with whitespace, and no type checking is performed. To enable the transformation of such features use the flag `--experimental-transform-types`. TypeScript features that depend on settings within `tsconfig.json`, such as paths or converting newer JavaScript syntax to older standards, are intentionally unsupported. To get full TypeScript support, see Full TypeScript support.

The type stripping feature is designed to be lightweight. By intentionally not supporting syntaxes that require JavaScript code generation, and by replacing inline types with whitespace, Node.js can run TypeScript code without the need for source maps.

Type stripping works with most versions of TypeScript but we recommend version 5.7 or newer with the following `tsconfig.json` settings:

```
{
  "compilerOptions": {
    "target": "esnext",
    "module": "nodenext",
    "allowImportingTsExtensions": true,
    "rewriteRelativeImportExtensions": true,
    "verbatimModuleSyntax": true
  }
}
```
COPY

Determining module system

Node.js supports both CommonJS and ES Modules syntax in TypeScript files. Node.js will not convert from one module system to another; if you want your code to run as an ES module, you must use `import` and `export` syntax, and if you want your code to run as CommonJS you must use `require` and `module.exports`.

- `.ts` files will have their module system determined the same way as `.js` files. To use `import` and `export` syntax, add `"type": "module"` to the nearest parent `package.json`.
- `.mts` files will always be run as ES modules, similar to `.mjs` files.
- `.cts` files will always be run as CommonJS modules, similar to `.cjs` files.
- `.tsx` files are unsupported.

As in JavaScript files, file extensions are mandatory in `import` statements and `import()` expressions: `import './file.ts'`, not `import './file'`. Because of backward compatibility, file extensions are also mandatory in `require()` calls: `require('./file.ts')`, not `require('./file')`, similar to how the `.cjs` extension is mandatory in `require` calls in CommonJS files.

The `tsconfig.json` option `allowImportingTsExtensions` will allow the TypeScript compiler `tsc` to type-check files with `import` specifiers that include the `.ts` extension.

TypeScript features

Since Node.js is only removing inline types, any TypeScript features that involve *replacing* TypeScript syntax with new JavaScript syntax will error, unless the flag `--experimental-transform-types` is passed.

The most prominent features that require transformation are:

- `Enum`
- `namespaces`
- `legacy module`
- parameter properties

Since Decorators are currently a TC39 Stage 3 proposal and will soon be supported by the JavaScript engine, they are not transformed and will result in a parser error. This is a temporary limitation and will be resolved in the future.

In addition, Node.js does not read `tsconfig.json` files and does not support features that depend on settings within `tsconfig.json`, such as paths or converting newer JavaScript syntax into older standards.

Importing types without `type` keyword

Due to the nature of type stripping, the `type` keyword is necessary to correctly strip type imports. Without the `type` keyword, Node.js will treat the import as a value import, which will result in a runtime error. The tsconfig option `verbatimModuleSyntax` can be used to match this behavior.

This example will work correctly:

```
import type { Type1, Type2 } from './module.ts';
import { fn, type FnParams } from './fn.ts';
```
COPY

This will result in a runtime error:

```
import { Type1, Type2 } from './module.ts';
import { fn, FnParams } from './fn.ts';
```
COPY

Non-file forms of input

Type stripping can be enabled for `--eval`. The module system will be determined by `--input-type`, as it is for JavaScript.

TypeScript syntax is unsupported in the REPL, STDIN input, `--print`, `--check`, and `inspect`.

Source maps

Since inline types are replaced by whitespace, source maps are unnecessary for correct line numbers in stack traces; and Node.js does not generate them. When `--experimental-transform-types` is enabled, source-maps are enabled by default.

Type stripping in dependencies

To discourage package authors from publishing packages written in TypeScript, Node.js will by default refuse to handle TypeScript files inside folders under a `node_modules` path.

Paths aliases

`tsconfig "paths"` won't be transformed and therefore produce an error. The closest feature available is subpath imports with the limitation that they need to start with `#`.