> ## 🛑 *Read carefully before you jump to conclusions on this page!*

There are easy ways to configure TypeScript to ensure faster compilations and editing experiences. The earlier that these practices can be adopted, the better. Beyond best-practices, there are some common techniques for investigating slow compilations/editing experiences, some common fixes, and some common ways of helping the TypeScript team investigate the issues as a last resort.

Writing Easy-to-Compile Code

Preferring Interfaces Over Intersections

Much of the time, a simple type alias to an object type acts very similarly to an interface.

```
interface Foo { prop: string }

type Bar = { prop: string };
```

However, and as soon as you need to compose two or more types, you have the option of extending those types with an interface, or intersecting them in a type alias, and that's when the differences start to matter.

Interfaces create a single flat object type that detects property conflicts, which are usually important to resolve! Intersections on the other hand just recursively merge properties, and in some cases produce `never`. Interfaces also display consistently better, whereas type aliases to intersections can't be displayed in part of other intersections. Type relationships between interfaces are also cached, as opposed to intersection types as a whole. A final noteworthy difference is that when checking against a target intersection type, every constituent is checked before checking against the "effective"/"flattened" type.

For this reason, extending types with `interface`s/`extends` is suggested over creating intersection types.

```diff
- type Foo = Bar & Baz & {
-     someProp: string;
- }
+ interface Foo extends Bar, Baz {
+     someProp: string;
+ }
```

Using Type Annotations

Adding type annotations, especially return types, can save the compiler a lot of work. In part, this is because named types tend to be more compact than anonymous types (which the compiler might infer), which reduces the amount of time spent reading and writing declaration files (e.g. for incremental builds). Type inference is very convenient, so there's no need to do this universally - however, it can be a useful thing to try if you've identified a slow section of your code.

```diff
- import { otherFunc } from "other";
+ import { otherFunc, otherType } from "other";

- export function func() {
+ export function func(): otherType {
      return otherFunc();
  }
```

Preferring Base Types Over Unions

Union types are great - they let you express the range of possible values for a type.

```
interface WeekdaySchedule {
  day: "Monday" | "Tuesday" | "Wednesday" | "Thursday" | "Friday";
  wake: Time;
  startWork: Time;
  endWork: Time;
  sleep: Time;
}

interface WeekendSchedule {
  day: "Saturday" | "Sunday";
  wake: Time;
  familyMeal: Time;
  sleep: Time;
}

declare function printSchedule(schedule: WeekdaySchedule | WeekendSchedule);
```

However, they also come with a cost. Every time an argument is passed to `printSchedule`, it has to be compared to each element of the union. For a two-element union, this is trivial and inexpensive. However, if your union has more than a dozen elements, it can cause real problems in compilation speed. For instance, to eliminate redundant members from a union, the elements have to be compared pairwise, which is quadratic. This sort of check might occur when intersecting large unions, where intersecting over each union member can result in enormous types that then need to be reduced. One way to avoid this is to use subtypes, rather than unions.

```
interface Schedule {
  day: "Monday" | "Tuesday" | "Wednesday" | "Thursday" | "Friday" | "Saturday" |
"Sunday";
  wake: Time;
  sleep: Time;
}

interface WeekdaySchedule extends Schedule {
  day: "Monday" | "Tuesday" | "Wednesday" | "Thursday" | "Friday";
  startWork: Time;
  endWork: Time;
}

interface WeekendSchedule extends Schedule {
  day: "Saturday" | "Sunday";
  familyMeal: Time;
}

declare function printSchedule(schedule: Schedule);
```

A more realistic example of this might come up when trying to model every built-in DOM element type. In this case, it would be preferable to create a base `HtmlElement` type with common members, which `DivElement`, `ImgElement`, etc. extend, rather than to create the exhaustive union `DivElement | /*...*/ | ImgElement | /*...*/`.

Using Project References

New Code

When building up any codebase of a non-trivial size with TypeScript, it is helpful to organize the codebase into several independent *projects*. Each project has its own `tsconfig.json` that has dependencies on other projects. This can be helpful to avoid loading too many files in a single compilation, and also makes certain codebase layout strategies easier to put together.

There are some very basic ways of organizing a codebase into projects. As an example, one might be a program with a project for the client, a project for the server, and a project that's shared between the two.

```
              ------------
              |          |
              |  Shared  |
              ^----------^
             /            \
            /              \
------------                ------------
|          |                |          |
|  Client  |                |  Server  |
-----^------                ------^-----
```

Tests can also be broken into their own project.

```
              ------------
              |          |
              |  Shared  |
              ^-----^----^
             /      |     \
            /       |      \
------------  ------------  ------------
|          |  |  Shared  |  |          |
|  Client  |  |  Tests   |  |  Server  |
-----^------  ------------  ------^-----
     |                            |
     |                            |
------------                ------------
|  Client  |                |  Server  |
|  Tests   |                |  Tests   |
------------                ------------
```

You can read up more about project references here.

Existing Code

When a workspace becomes so large that it's hard for the editor to handle (and you've used performance tracing to confirm that there are no hotspots, making scale the most likely culprit), it can be helpful to break it down into a collection of projects that reference each other. If you're working in a monorepo, this can be as simple as creating a project for each package and mirroring the package dependency graph in project references. Otherwise the process is more ad hoc - you may be able to follow the directory structure or you may have to use carefully chosen `include` and `exclude` globs. Some things to keep in mind:

- Aim for evenly-sized projects - avoid having a single humongous project with lots of tiny satellites
- Try to group together files that will be edited together - this will limit the number of projects the editor needs to load
- Separating out test code can help prevent product code from accidentally depending on it

Performance Considerations

As with any encapsulation mechanism, projects come with a cost. For example, if all projects depend on the same packages (e.g. a popular UI framework), some parts of that package's types will be checked repeatedly - once for each project consuming them. Empirically, it seems that (for a workspace with more than one project) 5-20 projects is an appropriate range - fewer may result in editor slowdowns and more may result in excessive overhead. Some good reasons to split out a project:

- It has a different output location (e.g. because it's a package in a monorepo)
- It requires different settings (e.g. `lib` or `moduleResolution`)
- It contains global declarations that you want to scope (either for encapsulation or to limit expensive global rebuilds)
- The editor's language service runs out of memory when trying to process the code as a single project
  - In this case, you will want to set `"disableReferencedProjectLoad": true` and `"disableSolutionSearching": true` to limit project loading while editing

Configuring `tsconfig.json` or `jsconfig.json`

TypeScript and JavaScript users can always configure their compilations with a `tsconfig.json` file. `jsconfig.json` files can also be used to configure the editing experience for JavaScript users.

Specifying Files

You should always make sure that your configuration files aren't including too many files at once.

Within a `tsconfig.json` , there are two ways to specify files in a project.

- the `files` list
- the `include` and `exclude` lists

The primary difference between the two is that `files` expects a list of file paths to source files, and `include` / `exclude` use globbing patterns to match against files.

While specifying `files` will allow TypeScript to quickly load up files up directly, it can be cumbersome if you have many files in your project without just a few top-level entry-points. Additionally, it's easy to forget to add new files to your `tsconfig.json` , which means that you might end up with strange editor behavior where those new files are incorrectly analyzed. All this can be cumbersome.

`include` / `exclude` help avoid needing to specify these files, but at a cost: files must be discovered by walking through included directories. When running through a *lot* of folders, this can slow compilations down. Additionally, sometimes a compilation will include lots of unnecessary `.d.ts` files and test files, which can increase compilation time and memory overhead. Finally, while `exclude` has some reasonable defaults, certain configurations like mono-repos mean that a "heavy" folders like `node_modules` can still end up being included.

For best practices, we recommend the following:

- Specify only input folders in your project (i.e. folders whose source code you want to include for compilation/analysis).
- Don't mix source files from other projects in the same folder.
- If keeping tests in the same folder as other source files, give them a distinct name so they can easily be excluded.
- Avoid large build artifacts and dependency folders like `node_modules` in source directories.

Note: without an `exclude` list, `node_modules` is excluded by default; as soon as one is added, it's important to explicitly add `node_modules` to the list.

Here is a reasonable `tsconfig.json` that demonstrates this in action.

```
{
    "compilerOptions": {
        // ...
    },
    "include": ["src"],
    "exclude": ["**/node_modules", "**/.*/"],
}
```

Controlling `@types` Inclusion

By default, TypeScript automatically includes every `@types` package that it finds in your `node_modules` folder, regardless of whether you import it. This is meant to make certain things "just work" when using Node.js, Jasmine, Mocha, Chai, etc. since these tools/packages aren't imported - they're just loaded into the global environment.

Sometimes this logic can slow down program construction time in both compilation and editing scenarios, and it can even cause issues with multiple global packages with conflicting declarations, causing errors like

```
Duplicate identifier 'IteratorResult'.
Duplicate identifier 'it'.
Duplicate identifier 'define'.
Duplicate identifier 'require'.
```

In cases where no global package is required, the fix is as easy as specifying an empty field for the `"types"` option in a `tsconfig.json` / `jsconfig.json`

```
// src/tsconfig.json
{
   "compilerOptions": {
       // ...

       // Don't automatically include anything.
       // Only include `@types` packages that we need to import.
       "types" : []
   },
   "files": ["foo.ts"]
}
```

If you still need a few global packages, add them to the `types` field.

```
// tests/tsconfig.json
{
   "compilerOptions": {
       // ...

       // Only include `@types/node` and `@types/mocha`.
       "types" : ["node", "mocha"]
   },
   "files": ["foo.test.ts"]
}
```

Incremental Project Emit

The `--incremental` flag allows TypeScript to save state from the last compilation to a `.tsbuildinfo` file. This file is used to figure out the smallest set of files that might to be re-checked/re-emitted since it last ran, much like how TypeScript's `--watch` mode works.

Incremental compiles are enabled by default when using the `composite` flag for project references, but can bring the same speed-ups for any project that opts in.

Skipping `.d.ts` Checking

By default, TypeScript performs a full re-check of all `.d.ts` files in a project to find issues and inconsistencies; however, this is typically unnecessary. Most of the time, the `.d.ts` files are known to already work - the way that types extend each other was already verified once, and declarations that matter will be checked anyway.

TypeScript provides the option to skip type-checking of the `.d.ts` files that it ships with (e.g. `lib.d.ts`) using the `skipDefaultLibCheck` flag.

Alternatively, you can also enable the `skipLibCheck` flag to skip checking *all* `.d.ts` files in a compilation.

These two options can often hide misconfiguration and conflicts in `.d.ts` files, so we suggest using them *only* for faster builds.

Using Faster Variance Checks

Is a list of dogs a list of animals? That is, is `List<Dog>` assignable to `List<Animals>`? The straightforward way to find out is to do a structural comparison of the types, member by member. Unfortunately, this can be very expensive. However, if we know enough about `List<T>`, we can reduce this assignability check to determining whether `Dog` is assignable to `Animal` (i.e. without considering each member of `List<T>`). (In particular, we need to know the variance of the type parameter `T`.) The compiler can only take full advantage of this potential speedup if the `strictFunctionTypes` flag is enabled (otherwise, it uses the slower, but more lenient, structural check). For this reason, we recommend building with `--strictFunctionTypes` (which is enabled by default under `--strict`).

Configuring Other Build Tools

TypeScript compilation is often performed with other build tools in mind - especially when writing web apps that might involve a bundler. While we can only make suggestions for a few build tools, ideally these techniques can be generalized.

Make sure that in addition to reading this section, you read up about performance in your choice of build tool - for example:

- ts-loader's section on *Faster Builds*
- awesome-typescript-loader's section on *Performance Issues*

Concurrent Type-Checking

Type-checking typically requires information from other files, and can be relatively expensive compared to other steps like transforming/emitting code. Because type-checking can take a little bit longer, it can impact the inner development loop - in other words, you might experience a longer edit/compile/run cycle, and this might be frustrating.

For this reason, some build tools can run type-checking in a separate process without blocking emit. While this means that invalid code can run before TypeScript reports an error in your build tool, you'll often see errors in your editor first, and you won't be blocked for as long from running working code.

An example of this in action is the `fork-ts-checker-webpack-plugin` plugin for Webpack, or awesome-typescript-loader which also sometimes does this.

Isolated File Emit

By default, TypeScript's emit requires semantic information that might not be local to a file. This is to understand how to emit features like `const enum` s and `namespace` s. But needing to check *other* files to generate the output for an arbitrary file can make emit slower.

The need for features that need non-local information is somewhat rare - regular `enum` s can be used in place of `const enum` s, and modules can be used instead of `namespace` s. For that reason, TypeScript provides the `isolatedModules` flag to error on features powered by non-local information. Enabling `isolatedModules` means that your codebase is safe for tools that use TypeScript APIs like `transpileModule` or alternative compilers like Babel.

As an example, the following code won't properly work at runtime with isolated file transforms because `const enum` values are expected to be inlined; but luckily, `isolatedModules` will tell us that early on.

```
// ./src/fileA.ts

export declare const enum E {
    A = 0,
    B = 1,
}

// ./src/fileB.ts

import { E } from "./fileA";

console.log(E.A);
//          ~
// error: Cannot access ambient const enums when the '--isolatedModules' flag is
provided.
```

> ***Remember:*** `isolatedModules` *doesn't automatically make code generation faster - it just tells you when you're about to use a feature that might not be supported. The thing you're looking for is isolated module emit in different build tools and APIs.*

Isolated file emit can be leveraged by using the following tools:

- ts-loader provides a `transpileOnly` flag which performs isolated file emit by using `transpileModule`.
- awesome-typescript-loader provides a `transpileOnly` flag which performs isolated file emit by using `transpileModule`.
- TypeScript's `transpileModule` API can be used directly.
- awesome-typescript-loader provides the `useBabel` flag.
- babel-loader compiles files in an isolated manner (but does not provide type-checking on its own).
- gulp-typescript enables isolated file emit when `isolatedModules` is enabled.
- rollup-plugin-typescript ***only*** performs isolated file compilation.
- ts-jest can use be configured with the [`isolatedModules` flag set to `true`]isolatedModules: true(.
- ts-node can detect the `"transpileOnly"` option in the `"ts-node"` field of a `tsconfig.json`, and also has a `--transpile-only` flag.

Investigating Issues

There are certain ways to get hints of what might be going wrong.

Disabling Editor Plugins

Editor experiences can be impacted by plugins. Try disabling plugins (especially JavaScript/TypeScript-related plugins) to see if that fixes any issues in performance and responsiveness.

Certain editors also have their own troubleshooting guides for performance, so consider reading up on them. For example, Visual Studio Code has its own page for Performance Issues as well.

`extendedDiagnostics`

You can run TypeScript with `--extendedDiagnostics` to get a printout of where the compiler is spending its time.

```
Files:                         6
Lines:                     24906
Nodes:                    112200
Identifiers:               41097
Symbols:                   27972
Types:                      8298
Memory used:              77984K
Assignability cache size:  33123
Identity cache size:           2
Subtype cache size:            0
I/O Read time:             0.01s
```

```
Parse time:                0.44s
Program time:              0.45s
Bind time:                 0.21s
Check time:                1.07s
transformTime time:        0.01s
commentTime time:          0.00s
I/O Write time:            0.00s
printTime time:            0.01s
Emit time:                 0.01s
Total time:                1.75s
```

> *Note that `Total time` won't be the sum of all times preceding it, since there is some overlap and some work is not instrumented.*

The most relevant information for most users is:

Field	Meaning
`Files`	the number of files that the program is including (use `--listFiles` to see what they are).
`I/O Read time`	time spent reading from the file system - this includes traversing `include`'d folders.
`Parse time`	time spent scanning and parsing the program
`Program time`	combined time spent performing reading from the file system, scanning and parsing the program, and other calculation of the program graph. These steps are intermingled and combined here because files need to be resolved and loaded once they're included via `import`s and `export`s.
`Bind time`	Time spent building up various semantic information that is local to a single file.
`Check time`	Time spent type-checking the program.
`transformTime time`	Time spent rewriting TypeScript ASTs (trees that represent source files) into forms that work in older runtimes.
`commentTime`	Time spent calculating comments in output files.
`I/O Write time`	Time spent writing/updating files on disk.
`printTime`	Time spent calculating the string representation of an output file and emitting it to disk.

Things that you might want to ask given this input:

- Does the number of files/number of lines of code roughly correspond to the number of files in your project? Try running `--listFiles` if not.
- Does `Program time` or `I/O Read time` seem fairly high? Ensure your `include` / `exclude` settings are configured correctly.
- Do other times seem off? You might want to file an issue! Things you can do to help diagnose it might be
  - Running with `emitDeclarationOnly` if `printTime` is high.
  - Read up instructions on Reporting Compiler Performance Issues.

`showConfig`

It's not always obvious what settings a compilation is being run with when running `tsc`, especially given that `tsconfig.json`s can extend other configuration files. `showConfig` can explain what `tsc` will calculate for an invocation.

```
tsc --showConfig

# or to select a specific config file...

tsc --showConfig -p tsconfig.json
```

`traceResolution`

Running with `traceResolution` can help explain *why* a file was included in a compilation. The emit is somewhat verbose, so you might want to redirect output to a file.

```
tsc --traceResolution > resolution.txt
```

If you find a file that shouldn't be present, you may need to look into fixing up your `include`/`exclude` lists in your `tsconfig.json`, or alternatively, you might need to adjust other settings like `types`, `typeRoots`, or `paths`.

Running `tsc` Alone

Much of the time, users run into slow performance using 3rd party build tools like Gulp, Rollup, Webpack, etc. Running with `tsc --extendedDiagnostics` to find major discrepancies between using TypeScript and the tool can indicate external misconfiguration or inefficiencies.

Some questions to keep in mind:

- Is there a major difference in build times between `tsc` and the build tool with TypeScript integration?
- If the build tool provides diagnostics, is there a difference between TypeScript's resolution and the build tool's?
- Does the build tool have *its own configuration* that could be the cause?
- Does the build tool have configuration *for its TypeScript integration* that could be the cause? (e.g. options for ts-loader?)

Upgrading Dependencies

Sometimes TypeScript's type-checking can be impacted by computationally intensive `.d.ts` files. This is rare, but can happen. Upgrading to a newer version of TypeScript (which can be more efficient) or to a newer version of an `@types` package (which may have reverted a regression) can often solve the issue.

Performance Tracing

In some cases, the approaches above might not give you enough insight to understand why TypeScript feels slow. TypeScript 4.1 and higher provides a `--generateTrace` option that can give you a sense of the work the compiler is spending time on. `--generateTrace` will create an output file that can be analyzed within Edge or Chrome.

Ideally, TypeScript will be able to compile your project without any errors, though it's not a strict requirement for tracing. Once you're ready to get a trace, you can run TypeScript with the `--generateTrace` flag.

```
tsc -p ./some/project/src/tsconfig.json --generateTrace tracing_output_folder
```

To quickly list performance hot-spots, you can install and run @typescript/analyze-trace from npm.

Alternatively, you can review the details manually:

1. Visit `about://tracing` on Edge/Chrome,
2. Click on the `Load` button at the top left,
3. Open the generated JSON file (`trace.*.json`) in your output directory.

Note that, even if your build doesn't directly invoke tsc (e.g. because you use a bundler) or the slowdown you're seeing is in the editor, collecting and interpreting a trace from tsc will generally be much easier than trying to investigate your slowdown directly and will still produce representative results.

You can read more about performance tracing in more detail here.

⚠ Warning: A performance trace may include information from your workspace, including file paths and source code. If you have any concerns about posting this publicly on GitHub, let us know and you can share the details privately.

⚠ Warning: The format of performance trace files is not stable, and may change from version to version.

Common Issues

Once you've trouble-shooted, you might want to explore some fixes to common issues. If the following solutions don't work, it may be worth filing an issue.

Misconfigured `include` and `exclude`

As mentioned above, the `include` / `exclude` options can be misused in several ways.

Problem	Cause	Fix
`node_modules` was accidentally included from deeper folder	*`exclude` was not set*	`"exclude": ["**/node_modules", "**/.*/"]`
`node_modules` was accidentally included from deeper folder	`"exclude": ["node_modules"]`	`"exclude": ["**/node_modules", "**/.*/"]`
Hidden dot files (e.g. `.git`) were accidentally included	`"exclude": ["**/node_modules"]`	`"exclude": ["**/node_modules", "**/.*/"]`
Unexpected files are being included.	*`include` was not set*	`"include": ["src"]`

Filing an Issue

If your project is already properly and optimally configured, you may want to file an issue.

The best reports of performance issues contain *easily obtainable* and *minimal* reproductions of the problem. In other words, a codebase that can easily be cloned over git that contains only a few files. They require either no external integration with build tools - they can either be invoked via `tsc` or use isolated code which consumes the TypeScript API. Codebases that require complex invocations and setups cannot be prioritized.

We understand that this is not always easy to achieve - specifically, because it is hard to isolate the source of a problem within a codebase, and because sharing intellectual property may be an issue. In some cases, the team will be willing to send a non-disclosure agreement (NDA) if we believe the issue is highly impactful.

Regardless of whether a reproduction is possible, following these directions when filing issues will help us provide you with performance fixes.

Reporting Compiler Performance Issues

Sometimes you'll witness performance issues in both build times as well as editing scenarios. In these cases, it's best to focus on the TypeScript compiler.

First, a nightly version of TypeScript should be used to ensure you're not hitting a resolved issue:

```
npm install --save-dev typescript@next

# or

yarn add typescript@next --dev
```

A compiler perf issue should include

- The version of TypeScript that was installed (i.e. `npx tsc -v` or `yarn tsc -v`)
- The version of Node on which TypeScript ran (i.e. `node -v`)
- The output of running with `extendedDiagnostics` (`tsc --extendedDiagnostics -p tsconfig.json`)
- Ideally, a project that demonstrates the issues being encountered.
- Output logs from profiling the compiler (`isolate-*-*-*.log` and `*.cpuprofile` files)

Providing Performance Traces

Performance traces are meant to help teams figure out build performance issues in their own codebases; however, they can also be useful for the TypeScript team in diagnosing and fixing issues. See the above section on performance traces and continue reading more on our dedicated performance tracing page.

Profiling the Compiler

It is important to provide the team with diagnostic traces by running Node.js v10+ with the `--trace-ic` flag alongside TypeScript with the `--generateCpuProfile` flag:

```
node --trace-ic ./node_modules/typescript/lib/tsc.js --generateCpuProfile
profile.cpuprofile -p tsconfig.json
```

Here `./node_modules/typescript/lib/tsc.js` can be replaced with any path to where your version of the TypeScript compiler is installed, and `tsconfig.json` can be any TypeScript configuration file. `profile.cpuprofile` is an output file of your choice.

This will generate two files:

- `--trace-ic` will emit to a file of the `isolate-*-*-*.log` (e.g. `isolate-00000176DB2DF130-17676-v8.log`).
- `--generateCpuProfile` will emit to a file with the name of your choice. In the above example, it will be a file named `profile.cpuprofile`.

> ⚠ *Warning: These files may include information from your workspace, including file paths and source code. Both of these files are readable as plain-text, and you can modify them before attaching them as part of a GitHub issue. (e.g. to scrub them of file paths that may expose internal-only information).*
>
> *However, if you have any concerns about posting these publicly on GitHub, let us know and you can share the details privately.*

Reporting Editing Performance Issues

Perceived editing performance is frequently impacted by a number of things, and the only thing within the TypeScript team's control is the performance of the JavaScript/TypeScript language service, as well as the integration between that language service and certain editors (i.e. Visual Studio, Visual Studio Code, Visual Studio for Mac, and Sublime Text). Ensure that all 3rd-party plugins are turned off in your editor to determine whether there is an issue with TypeScript itself.

Editing performance issues are slightly more involved, but the same ideas apply: clone-able minimal repro codebases are ideal, and though in some cases the team will be able to sign an NDA to investigate and isolate issues.

Including the output from `tsc --extendedDiagnostics` is always good context, but taking a TSServer trace is the most helpful.

Taking a TSServer Log

Collecting a TSServer Log in Visual Studio Code

1. Open up your command palette and either
   - open your global settings by entering `Preferences: Open User Settings`
   - open your local project by entering `Preferences: Open Workspace Settings`
2. Set the option `"typescript.tsserver.log": "verbose",`
3. Restart VS Code and reproduce the problem
4. In VS Code, run the `TypeScript: Open TS Server log` command
5. This should open the `tsserver.log` file.

⚠ Warning: A TSServer log may include information from your workspace, including file paths and source code. If you have any concerns about posting this publicly on GitHub, let us know and you can share the details privately.