Common TypeScript Configurations

Configure and customize how TypeScript behaves.

TypeScript provides over a hundred configuration options. Its options allow us to customize how TypeScript type checks and transpiles our projects. To begin exploring, we can learn how to read the documentation and highlight some of the most popular options. In addition, we'll learn how to import popular sets of configurations and how to create and share our own. Let's con*figure* this stuff out.

Reading the TypeScript Reference

TypeScript's configuration reference is divided into a set of top-level options. Each one of these options is a top-level key in a `tsconfig.json` file. Feel free to click on the provided hyperlinks to get more details and examples:

- `compilerOptions`: The bulk of options. Covers how TypeScript should type check and transpile code into JavaScript.
- `exclude`: Specifies an array of globs (like `src/**/*`) to remove from files defined in `include`.
- `extends`: Loads an external TypeScript configuration file as the base configuration. This makes sharing common configurations possible.
- `files`: Specifies specific files to include.
- `include`: Specifies an array of globs (like `src/**/*`) to include.

Common Configuration Properties

Most of the time we will be dealing with options defined in `compilerOptions`, since they determine how TypeScript behaves. We listed some common options and their links below to provide an understanding of how TypeScript is configured — you don't have to read through everything, but explore at least one option in-depth to grasp how options are implemented.

- `strict`: When `true`, requires that all code must adhere to a variety of type standards, like passing proper arguments to methods and treating `null` and `undefined` as separate types. Often projects will enable `strict`, but then override specific parts of `strict` by turning off features like the `strictFunctionTypes` option.
- `paths`: Allows re-mapping imports. If we have imports like `import MyComponent from './app/src/components/MyComponent'`, we may want to shorten the import

path to a shorthand path like `'~/components/MyComponent'`. The `paths` option allows us to define convenient short-hand conventions for paths when writing code. When TypeScript transpiles our code, it will use this mapping to insert the correct paths.

- `allowJS`: The `allowJs` compiler option allows constructs declared in JavaScript files to factor into type-checking TypeScript files. This is a great option to set to `true` when migrating a JavaScript codebase to TypeScript. `noUnusedParameters`: Ensures that every parameter defined on a function is used inside the function.
- `noUnusedLocals`: Ensures that every defined variable is used somewhere.
- `sourceMap`: Generates source map files, which help debuggers and error reporting services display the original TypeScript code when reporting errors, instead of the transpiled and minified JavaScript code.
- `resolveJsonModule`: Allows importing `*.json` files inside TypeScript files.

The options above are just the start of all of the options available. Check out TypeScript's `compilerOptions` reference to view them all.

Using Community Created Configurations

Figuring out all the options we should include for every project can add a lot of setup time when starting a project. Luckily, the TypeScript community maintains a variety of configuration *bases*. A configuration base is a shared TypeScript configuration that we can install as a dependency and then use in our `tsconfig.json`.

To install a base, we can run the following command at the root of our TypeScript project:

```
npm install --save-dev @tsconfig/recommended
```

Then, in our `tsconfig.json` file, we can add this base as the value of the `extends` option:

```
{
  "extends": "@tsconfig/recommended/tsconfig.json",
  "compilerOptions": { ... }
}
```

The `@tsconfig/recommended` base applies this configuration. While this will set up our `tsconfig.json` with a variety of options, we can override any option defined by the base configuration by defining options in our `tsconfig.json`.

A full list of bases is listed in the public tsconfig/bases repository on GitHub.

Creating a Shared Configuration

Using a base configuration created by the community is a great way to get started, but sometimes we need a configuration that fits our exact needs that we can share across projects.

To create a custom TypeScript base, we'll need to create a JSON file. For this example, we'll name it `custom-tsconfig.json`.

Inside `custom-tsconfig.json`, we can put any TypeScript configuration that we'd like to use as a base configuration. Here's an example:

```
{
  "compilerOptions": {
    "target": "esnext",
    "strict": true,
    "noUnusedParameters": true,
    "noUnusedLocals": true
  }
}
```

Then, inside `tsconfig.json`, we can add `custom-tsconfig.json` as the base configuration by adding its path to the `extends` option:

```
{
  "extends": "./custom-tsconfig.json",
  "compilerOptions": { ... }
}
```

When working on a team with a variety of projects, a common pattern is to create an npm package that includes the team's TypeScript configuration preferences. Then, every project can use it as a base configuration, which makes every codebase behave the same by default when running TypeScript's type checking and transpilation features.

Wrap Up

With our knowledge of the TypeScript configuration, its categories, and popular configuration options, we're ready to customize how TypeScript type checks and transpiles our code. In addition, we can use base configurations from the community or ones we defined ourselves to make setting up new projects a breeze. So whether working on a new project, converting an existing project to TypeScript, or jumping into an already established project, know that you have *options*.