Using Vue with TypeScript

A type system like TypeScript can detect many common errors via static analysis at build time. This reduces the chance of runtime errors in production, and also allows us to more confidently refactor code in large-scale applications. TypeScript also improves developer ergonomics via type-based auto-completion in IDEs.

Vue is written in TypeScript itself and provides first-class TypeScript support. All official Vue packages come with bundled type declarations that should work out-of-the-box.

Project Setup

`create-vue`, the official project scaffolding tool, offers the options to scaffold a Vite-powered, TypeScript-ready Vue project.

Overview

With a Vite-based setup, the dev server and the bundler are transpilation-only and do not perform any type-checking. This ensures the Vite dev server stays blazing fast even when using TypeScript.

- During development, we recommend relying on a good IDE setup for instant feedback on type errors.
- If using SFCs, use the `vue-tsc` utility for command line type checking and type declaration generation. `vue-tsc` is a wrapper around `tsc`, TypeScript's own command line interface. It works largely the same as `tsc` except that it supports Vue SFCs in addition to TypeScript files. You can run `vue-tsc` in watch mode in parallel to the Vite dev server, or use a Vite plugin like vite-plugin-checker which runs the checks in a separate worker thread.
- Vue CLI also provides TypeScript support, but is no longer recommended. See notes below.

- Visual Studio Code (VS Code) is strongly recommended for its great out-of-the-box support for TypeScript.
  - Vue - Official (previously Volar) is the official VS Code extension that provides TypeScript support inside Vue SFCs, along with many other great features.

    > ⓘ TIP
    >
    > Vue - Official extension replaces Vetur, our previous official VS Code extension for Vue 2. If you have Vetur currently installed, make sure to disable it in Vue 3 projects.

- WebStorm also provides out-of-the-box support for both TypeScript and Vue. Other JetBrains IDEs support them too, either out of the box or via a free plugin. As of version 2023.2, WebStorm and the Vue Plugin come with built-in support for the Vue Language Server. You can set the Vue service to use Volar integration on all TypeScript versions, under Settings > Languages & Frameworks > TypeScript > Vue. By default, Volar will be used for TypeScript versions 5.0 and higher.

Configuring `tsconfig.json`

Projects scaffolded via `create-vue` include pre-configured `tsconfig.json`. The base config is abstracted in the `@vue/tsconfig` package. Inside the project, we use Project References to ensure correct types for code running in different environments (e.g. app code and test code should have different global variables).

When configuring `tsconfig.json` manually, some notable options include:

- `compilerOptions.isolatedModules` is set to `true` because Vite uses esbuild for transpiling TypeScript and is subject to single-file transpile limitations. `compilerOptions.verbatimModuleSyntax` is a superset of `isolatedModules` and is a good choice, too - it's what `@vue/tsconfig` uses.
- If you're using Options API, you need to set `compilerOptions.strict` to `true` (or at least enable `compilerOptions.noImplicitThis`, which is a part of the `strict` flag) to leverage type checking of `this` in component options. Otherwise `this` will be treated as `any`.
- If you have configured resolver aliases in your build tool, for example the `@/*` alias configured by default in a `create-vue` project, you need to also configure it for TypeScript via `compilerOptions.paths`.
- If you intend to use TSX with Vue, set `compilerOptions.jsx` to `"preserve"`, and set `compilerOptions.jsxImportSource` to `"vue"`.

- Official TypeScript compiler options docs
- esbuild TypeScript compilation caveats

Note on Vue CLI and `ts-loader`

In webpack-based setups such as Vue CLI, it is common to perform type checking as part of the module transform pipeline, for example with `ts-loader`. This, however, isn't a clean solution because the type system needs knowledge of the entire module graph to perform type checks. Individual module's transform step simply is not the right place for the task. It leads to the following problems:

- `ts-loader` can only type check post-transform code. This doesn't align with the errors we see in IDEs or from `vue-tsc`, which map directly back to the source code.
- Type checking can be slow. When it is performed in the same thread / process with code transformations, it significantly affects the build speed of the entire application.
- We already have type checking running right in our IDE in a separate process, so the cost of dev experience slow down simply isn't a good trade-off.

If you are currently using Vue 3 + TypeScript via Vue CLI, we strongly recommend migrating over to Vite. We are also working on CLI options to enable transpile-only TS support, so that you can switch to `vue-tsc` for type checking.

General Usage Notes

`defineComponent()`

To let TypeScript properly infer types inside component options, we need to define components with `defineComponent()`:

```ts
import { defineComponent } from 'vue'

export default defineComponent({
  // type inference enabled
  props: {
    name: String,
    msg: { type: String, required: true }
  },
  data() {
    return {
      count: 1
    }
  },
    this.msg // type: string
    this.count // type: number
  }
})
```

`defineComponent()` also supports inferring the props passed to `setup()` when using Composition API without `<script setup>`:

```ts
import { defineComponent } from 'vue'

export default defineComponent({
  // type inference enabled
  props: {
    message: String
  },
  setup(props) {
    props.message // type: string | undefined
  }
})
```

See also:

- Note on webpack Treeshaking
- type tests for `defineComponent`

> ⓘ TIP
>
> `defineComponent()` also enables type inference for components defined in plain JavaScript.

Usage in Single-File Components

To use TypeScript in SFCs, add the `lang="ts"` attribute to `<script>` tags. When `lang="ts"` is present, all template expressions also enjoy stricter type checking.

```vue
<script lang="ts">
import { defineComponent } from 'vue'

export default defineComponent({
  data() {
    return {
      count: 1
    }
  }
})
</script>
```

```
  {{ count.toFixed(2) }}
</template>
```

`lang="ts"` can also be used with `<script setup>`:

```vue
<script setup lang="ts">
// TypeScript enabled
import { ref } from 'vue'

const count = ref(1)
</script>

<template>
  <!-- type checking and auto-completion enabled -->
  {{ count.toFixed(2) }}
</template>
```

TypeScript in Templates

The `<template>` also supports TypeScript in binding expressions when `<script lang="ts">` or `<script setup lang="ts">` is used. This is useful in cases where you need to perform type casting in template expressions.

Here's a contrived example:

```vue
<script setup lang="ts">
let x: string | number = 1
</script>

<template>
  <!-- error because x could be a string -->
  {{ x.toFixed(2) }}
</template>
```

This can be worked around with an inline type cast:

```vue
<script setup lang="ts">
let x: string | number = 1
</script>

<template>
  {{ (x as number).toFixed(2) }}
</template>
```

ⓘ TIP

Usage with TSX

Vue also supports authoring components with JSX / TSX. Details are covered in the Render Function & JSX guide.

Generic Components

Generic components are supported in two cases:

- In SFCs: `<script setup>` with the `generic` attribute
- Render function / JSX components: `defineComponent()`'s function signature

API-Specific Recipes

- TS with Composition API
- TS with Options API

Edit this page on GitHub

Previous
Security

Next
TS with Composition API