How to write a Vue.js app completely in TypeScript

(source : https://blog.logrocket.com/how-to-write-a-vue-js-app-completely-in-typescript/)

January 20, 2020 7 min read

Vue is an amazing, lightweight, and progressive frontend framework. Since Vue is flexible, users are not forced to use TypeScript. And unlike Angular, older versions of Vue did not have proper support for TypeScript. For this reason, most Vue applications have historically been written in JavaScript.

Now with official support for TypeScript, it's possible to create TypeScript projects from scratch using Vue CLI. However, we still need some third-party packages with custom decorators and features to create a true, complete TypeScript application, and the official documentation does not include all the information you need to get started.

To help paint a fuller picture, we'll demonstrate how to build a new Vue + TypeScript application using Vue CLI.

Getting started

Start with this line of code:

```
vue create typescript-app
```

Choose `manually select features` and configure it as shown below.

```
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, TS, Router, Vuex, CSS Pre-processors, Linter, Unit
? Use class-style component syntax? Yes
? Use Babel alongside TypeScript (required for modern mode, auto-detected polyfills, transpiling JSX)? Yes
? Use history mode for router? (Requires proper server setup for index fallback in production) Yes
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are supported by default): Sass/SCSS (with
? Pick a linter / formatter config: Prettier
? Pick additional lint features: (Press <space> to select, <a> to toggle all, <i> to invert selection)Lint on
? Pick a unit testing solution: Jest
? Where do you prefer placing config for Babel, ESLint, etc.? In dedicated config files
? Save this as a preset for future projects? No
```

After the project setup, we'll run the project to test it once.

```
cd typescript-app
```

```
npm run serve
```

Open `localhost:8080` (or the URL your console shows after starting the project), and we can see it running successfully.

As we move through this tutorial, we'll review the following and show how to write them using TypeScript.

1. Class-based components
2. Data, props, computed properties, methods, watchers, and emit
3. Lifecycle hooks
4. Mixins
5. Vuex

Open the `HelloWorld.vue` file from the components directory, and you'll see a structure like below.

Note: For each instance, I'll show both the TypeScript and JavaScript-equivalent code so you can easily compare the two. Let's get started!

1. Class-based components

```
//Typescript code
<script lang="ts">
import { Component, Vue } from 'vue-property-decorator'
@Component
export default class HelloWorld extends Vue {
}
</script>
```

The JavaScript-equivalent code would be:

```
<script>
export default {
name: 'HelloWorld'
}
</script>
```

To use TypeScript, we need to first set the `lang` attribute in the `<script>` tag to `ts`.

`vue-property-decorator` is a third-party package that uses the official `vue-class-component` package and adds more decorators on top of that. We could also explicitly use the `name` attribute to name the component, but using it as a class name will suffice.

```
@component({
  name: 'HelloWorld'
})
```

Importing a component

The code to register components inside the other components is written inside the `@Component` decorator, like below.

```
<template>
  <div class="main">
    <project />
  </div>
</template>
<script lang="ts">
import { Component, Vue } from 'vue-property-decorator'
import Project from '@/components/Project.vue'
@Component({
  components: {
    project
  }
})
export default class HelloWorld extends Vue {
}
</script>
```

The JavaScript-equivalent code would be:

```
<template>
  <div class="main">
    <project />
  </div>
</template>
<script>
import Project from '@/components/Project.vue'
export default {
  name: 'HelloWorld',
  components: {
    project
  }
})
</script>
```

2. Data, props, computed properties, methods, watchers, and emit

Using data

To use data properties, we can simply declare them as class variables.

```
@Component
export default class HelloWorld extends Vue {
  private msg: string = "welcome to my app"
  private list: Array<object> = [
```

```
    {
      name: 'Preetish',
      age: '26'
    },
    {
      name: 'John',
      age: '30'
    }
  ]
}
```

The JavaScript-equivalent code would look like this:

```
export default {
  data() {
    return {
      msg: "welcome to my app",
      list: [
        {
          name: 'Preetish',
          age: '26'
        },
        {
          name: 'John',
          age: '30'
        }
      ]
    }
}
```

Using props

We can use the `@Prop` decorator to use props in our Vue component. In Vue, we can give additional details for props, such as `required`, `default`, and `type`. We first import the `Prop` decorator from `vue-property-decorator` and write it as shown below. We could also use `readonly` to avoid manipulating the props.

```
import { Component, Prop, Vue } from 'vue-property-decorator'
@Component
export default class HelloWorld extends Vue {
  @Prop() readonly msg!: string
  @Prop({default: 'John doe'}) readonly name: string
  @Prop({required: true}) readonly age: number
  @Prop(String) readonly address: string
  @Prop({required: false, type: String, default: 'Developer'})
readonly job: string
}
</script>
```

The JavaScript-equivalent code would be as follows.

```
export default {
  props: {
    msg,
    name: {
      default: 'John doe'
    },
    age: {
      required: true,
    },
    address: {
      type: String
    },
    job: {
      required: false,
      type: string,
      default: 'Developer'
    }
  }
}
```

Computed properties

A computed property is used to write simple template logic, such as manipulating, appending, or concatenating data. In TypeScript, a normal computed property is also prefixed with the `get` keyword.

```
export default class HelloWorld extends Vue {
  get fullName(): string {
    return this.first+ ' '+ this.last
  }
}
```

Here is the JavaScript equivalent:

```
export default {
  fullName() {
    return this.first + ' ' + this.last
  }
}
```

We can write complex computed properties, which has both `getter` and `setter`, in TypeScript as follows.

```
export default class HelloWorld extends Vue {
```

```
  get fullName(): string {
    return this.first+ ' '+ this.last
  }
  set fullName(newValue: string) {
    let names = newValue.split(' ')
    this.first = names[0]
    this.last = names[names.length - 1]
  }
}
```

The JavaScript-equivalent code would be:

```
fullName: {
  get: function () {
    return this.first + ' ' + this.last
  },
  set: function (newValue) {
    let names = newValue.split(' ')
    this.first = names[0]
    this.last = names[names.length - 1]
  }
}
```

Methods

Methods in TypeScript, like normal class methods, have an optional access modifier.

```
export default class HelloWorld extends Vue {
  public clickMe(): void {
    console.log('clicked')
    console.log(this.addNum(4, 2))
  }
  public addNum(num1: number, num2: number): number {
    return num1 + num2
  }
}
```

The JavaScript-equivalent code is as follows.

```
export default {
  methods: {
    clickMe() {
      console.log('clicked')
      console.log(this.addNum(4, 2))
    }
    addNum(num1, num2) {
      return num1 + num2
    }
  }
```

```
}
```

Watchers

Watcher are written differently from how we usually write in JavaScript. The most-used syntax for a watcher in JavaScript is:

```
watch: {
  name: function(newval) {
    //do something
  }
}
```

We don't tend to use handler syntax often.

```
watch: {
  name: {
    handler: 'nameChanged'
  }
}
methods: {
  nameChanged (newVal) {
    // do something
  }
}
```

However, the TypeScript syntax is similar to the second method. In TypeScript, we use the `@Watch` decorator and pass the name of the variable we need to watch.

```
@Watch('name')
nameChanged(newVal: string) {
  this.name = newVal
}
```

We can also set the `immediate` and `deep` watchers.

```
@Watch('project', {
  immediate: true, deep: true
})
projectChanged(newVal: Person, oldVal: Person) {
  // do something
}
```

Here is the JS-equivalent code:

```
watch: {
  person: {
      handler: 'projectChanged',
      immediate: true,
      deep: true
```

```
    }
}
methods: {
  projectChanged(newVal, oldVal) {
    // do something
  }
}
```

Emit

To emit a method from a child component to a parent component, we'll use the `@Emit` decorator in TypeScript.

```
@Emit()
addToCount(n: number) {
  this.count += n
}
@Emit('resetData')
resetCount() {
  this.count = 0
}
```

In the first example, function name `addToCount` is converted to `kebab-case`, very similarly to how the Vue emit works.

In the second example, we pass the explicit name `resetData` for the method, and that name is used instead. Since `addData` is in `CamelCase`, it is converted to `kebab-case` again.

```
<some-component add-to-count="someMethod" />
<some-component reset-data="someMethod" />
```

```
//Javascript Equivalent
 methods: {
    addToCount(n) {
      this.count += n
      this.$emit('add-to-count', n)
    },
    resetCount() {
      this.count = 0
      this.$emit('resetData')
    }
}
```

3. Lifecycle hooks

A Vue component has eight lifecycle hooks, including `created`, `mounted`, etc., and the same TypeScript syntax is used for each hook. These are declared as normal class methods. Since

lifecycle hooks are automatically called, they neither take an argument nor return any data. So we don't need access modifiers, typing arguments, or return types.

```
export default class HelloWorld extends Vue {
  mounted() {
    //do something
  }
  beforeUpdate() {
    // do something
  }
}
```

The JavaScript-equivalent code is shown below.

```
export default {
  mounted() {
    //do something
  }
  beforeUpdate() {
    // do something
  }
}
```

4. Mixins

To create mixins in TypeScript, we must first create our mixin file, which contains the data we share with other components.

Create a file called `ProjectMixin.ts` inside the mixins directory and add the following mixin, which shares the project name and a method to update the project name.

```
import { Component, Vue } from 'vue-property-decorator'
@Component
class ProjectMixin extends Vue {
  public projName: string = 'My project'
  public setProjectName(newVal: string): void {
    this.projName = newVal
  }
}
export default ProjectMixin
```

In JavaScript, we'd write this code as follows.

```
export default {
  data() {
    return {
      projName: 'My project'
    }
  },
  methods: {
```

```
    setProjectName(newVal) {
      this.projName = newVal
    }
  }
}
```

To use the above mixin in our Vue component, we need to import `Mixins` from `vue-property-decorator` and our mixin file itself and write it as follows.

```
//Projects.vue
<template>
  <div class="project-detail">
    {{ projectDetail }}
  </div>
</template>
<script lang="ts">
import { Component, Vue, Mixins } from 'vue-property-decorator'
import ProjectMixin from '@/mixins/ProjectMixin'
@Component
export default class Project extends Mixins(ProjectMixin) {
  get projectDetail(): string {
    return this.projName + ' ' + 'Preetish HS'
  }
}
</script>
```

The JavaScript-equivalent code would be:

```
<template>
  <div class="project-detail">
    {{ projectDetail }}
  </div>
</template>
<script>
import ProjectMixin from '@/mixins/ProjectMixin'
export default {
  mixins: [ ProjectMixin ],
  computed: {
    projectDetail() {
      return this.projName + ' ' + 'Preetish HS'
    }
  }
}
</script>
```

5. Vuex

Vuex is the official state management library used in most Vue.js applications. It's a good practice to split the store into namespaced modules. We'll demonstrate how to write that in TypeScript.

First, we need to install two popular third-party packages:

```
npm install vuex-module-decorators -D
```

```
npm install vuex-class -D
```

In the `store` folder, let's create a `module` folder to place each of our namespaced store modules.

Create a file called `user.ts` to have the user state.

```
// store/modules/user.ts
import { VuexModule, Module, Mutation, Action } from 'vuex-module-decorators'
@Module({ namespaced: true, name: 'test' })
class User extends VuexModule {
  public name: string = ''
  @Mutation
  public setName(newName: string): void {
    this.name = newName
  }
  @Action
  public updateName(newName: string): void {
    this.context.commit('setName', newName)
  }
}
export default User
```

The `vuex-module-decorators` library provides decorators for `Module`, `Mutation`, and `Action`. The state variables are declared directly, like class variables. This is a simple module that stores a user's name and has a mutation and an action to update the user name.

We don't need to have `state` as our first parameter in `Mutations` and `context` in case of `Actions` — the library takes care of that. It's already injected into those methods.

Below is the JavaScript-equivalent code.

```
export default {
  namespaced: true,
  state: {
    name: ''
  },
  mutations: {
    setName(state, newName) {
      state.name = newName
    }
  },
```

```
  actions: {
    updateName(context, newName) {
      context.commit('setName', newName)
    }
  }
}
```

In the store folder, we need to create an `index.ts` file to initialize `vuex` and register this `module`:

```
import Vue from 'vue'
import Vuex from 'vuex'
import User from '@/store/modules/user'
Vue.use(Vuex)
const store = new Vuex.Store({
  modules: {
    User
  }
})
export default store
```

Using Vuex in components

To use Vuex, we can leverage a library called `vuex-class`. This library provides decorators to bind `State`, `Getter`, `Mutation`, and `Action` in our Vue component.

Since we are using namespaced Vuex modules, we first import `namespace` from `vuex-class` and then pass the name of the module to get access to that module.

```
<template>
  <div class="details">
    <div class="username">User: {{ nameUpperCase }}</div>
    <input :value="name" @keydown="updateName($event.target.value)"
/>
  </div>
</template>
<script lang="ts">
import { Component, Vue } from 'vue-property-decorator'
import { namespace } from 'vuex-class'
const user = namespace('user')
@Component
export default class User extends Vue {
  @user.State
  public name!: string

  @user.Getter
  public nameUpperCase!: string

  @user.Action
  public updateName!: (newName: string) => void
```

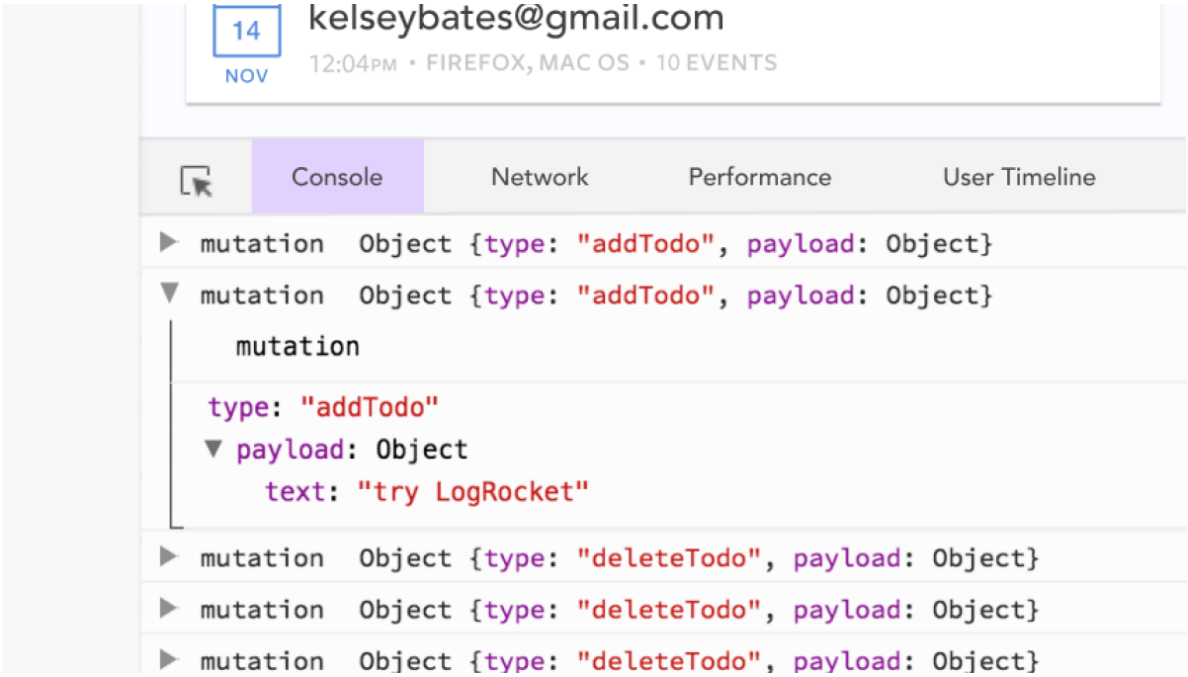
```
}
</script>
```

The JavaScript-equivalent code would be:

```
<template>
  <div class="details">
    <div class="username">User: {{ nameUpperCase }}</div>
    <input :value="name" @keydown="updateName($event.target.value)"
/>
  </div>
</template>
<script>
import { mapState, mapGetters, mapActions} from 'vuex'
export default  {
  computed: {
    ...mapState('user', ['name']),
    ...mapGetters('user', ['nameUpperCase'])
  }
  methods: {
    ...mapActions('user', ['updateName'])
  }
}
</script>
```

The next step: use LogRocket to see a replay of Vuex mutations for every user in your app

Debugging Vue.js applications can be difficult, especially when there are dozens, if not hundreds of mutations during a user session. If you're interested in monitoring bugs and performance in your production Vue apps, try LogRocket.



https://logrocket.com/signup/

LogRocket is like a DVR for web apps, recording literally everything that happens in your Vue apps including network requests, JavaScript errors, performance problems, and much more. Instead of guessing why problems happen, you can aggregate and report on what state your application was in when an issue occurred.

The LogRocket Vuex plugin logs Vuex mutations to the LogRocket console, giving you context around what led to an error, and what state the application was in when an issue occurred.

Modernize how you debug your Vue apps – Start monitoring for free.

Conclusion

Now you have all the basic information you need to create a Vue.js application completely in TypeScript using a few official and third-party libraries to fully leverage the typing and custom decorator features. Vue 3.0 will have better support for TypeScript out of the box, and the whole Vue.js code was rewritten in TypeScript to improve maintainability.

Using TypeScript might seem a bit overwhelming at first, but when you get used to it, you'll have far fewer bugs in your code and smooth code collaboration between other developers who work on the same code base.

Jan 1, 2021 8 min read

24 Replies to “How to write a Vue.js app completely in TypeScript”

1.  **Nikita Sobolev** Says:

   January 23, 2020 at 9:39 am

   Great article!

   You can use https://github.com/wemake-services/wemake-vue-template to get the similar setup in seconds!
   It comes with full TypeScript support and Nuxt.

   Check it out!

2.  **Joshua** Says:

   February 22, 2020 at 10:41 am

   Nice! Filling in the typescript gaps between vuex and vue components is very important. Is there a way of gracefully getting this to work with action and getters in the root store as well?

   const store = new Vuex.Store({
   actions: {}, // Can’t make use of decorators
   getters: {}, // Can’t make use of decorators
   modules: {
   User
   }
   })

3.  **Preetish HS** Says:

   February 24, 2020 at 2:10 am

   Thank you!
   Great I’ll check it out. Thanks for sharing.

4.  **Preetish HS** Says:

   February 24, 2020 at 2:41 am

   If you are using actions and getters in your root store, you will be able to access it directly like this

“`
import {
Getter,
Action,
} from ‘vuex-class’

@Component
export class MyComp extends Vue {

@Getter(‘foo’) getterFoo
@Action(‘foo’) actionFoo

created () {
this.getterFoo
this.actionFoo()

}
}
“`

https://github.com/ktsn/vuex-class

5.  **Robert Wildling** Says:

March 4, 2020 at 9:24 am

Thank you for this article! I am new, and with this topic I find the offical docu not so good as its reputation would make me expect.

However, I do get lost, when you import “Project” and later set it as a component as “project” (lowercase). That doesn’t work in my case. Shouldn’t the component be written in uppercase, too?

6.  **Robert Wildling** Says:

March 4, 2020 at 9:51 am

And I get a ton off errors when defining the @Props:

ERROR in /Volumes/_II_/_SITES/vue_chord_player/src/components/Keyboard.vue(15,43):
15:43 Property ‘name’ has no initializer and is not definitely assigned in the constructor.
13 | export default class Keyboard extends Vue {
14 | @Prop() readonly msg!: string
> 15 | @Prop({ default: ‘John Doe’ }) readonly name: string
| ^

This is weird, because I do follow your setup completely… what is wrong here?

7.  **Preetish H. Shantharaja** Says:

   March 12, 2020 at 7:47 am

   Hi Robert,
   Thanks for pointing it out, you are right, it should have been uppercase “Project”.
   And coming to props. It should actually work, Here is a working github repo which I created for Nuxt + Typescript. It Has similar @Prop declarations. You can check it out here. It should be there same for Vue as well.

   https://github.com/preetishhs/nuxt-typescript-example

8.  **Greg** Says:

   April 6, 2020 at 7:39 pm

   Hi, I applaud anybody who is spreading information about how to get typescript and vue working. Until Vue 3.0 comes out, it is not an easy thing to do. I have been working on my learning of it for the past 2 -3 months. My top two requirements were to use typescript (preferably classes) and unit testing.

   I early on came to the conclusion that the proper solution at this point in time is to use the composition API and not the options API or the class API for Vue components. The composition API is the way forward for typescript in Vue and we can start using it today with the composition api plugin. And it makes your code very easy to separate the logic from the presentation so both can easily be unit tested. I have been defining all my logic of a vue in a class that gets instantiated in the vue component’s setup function and the class’ variables are defined as Refs for reactivity. It works great and unit testing it is so easy. it also allows you to separate different parts of the component’s logic into different classes so those items can be unit tested independently.

   For global state, I tried vuex-module-decorators and gave up. They seem to work at first, but once you start trying to do error handling of actions, both in production code and unit tests, you start to get strange errors from the decorator code. There is an SO thread an issue on github about it. This issue was the thing that finally made me abandon vuex. It just isn’t architected well to work with a typed language like typescript and I don’t want to give up the type safety and tooling support of typescript by casting everything everywhere. So now that I don’t use vuex, I have started to create typescript classes that are singletons as my global state and I import these modules wherever I need them. It works perfectly and I get typesafety because they are just classes. And they are incredibly easy to unit test. I don’t have to do anything special.

   Now that i have done all that, Vue and typescript work really well together.

   My biggest recommendation would be to ditch vuex. I would say that 80% of my time learning vue was trying all the different approaches to typescript and vuex. It all felt like banging my head against the wall

9.  **BENEDICT MWANGA** Says:

   April 24, 2020 at 4:49 am

   This is great! please keep this article alive for as long as possible. please note this is in my bookmarks as i refer to it every time i am stuck

10.  **Preetish H. Shantharaja** Says:

    April 24, 2020 at 9:31 am

    Hi Benedict. Thanks. This article will always be live.
    I forgot to add an example app link in the article. If you want to take a look at the code of a simple Vue.js app written fully in typescript. Please take a look at this.

    https://github.com/preetishhs/vue-typescript-example

11.  **BENEDICT MWANGA** Says:

    May 15, 2020 at 1:39 pm

    I can not thank you enough bro, the provided link(GitHub repo) has helped me a-lot to spot some of thinks that did not sink well while following your tutorial. Thanks man

12.  **Daniel Andrade Groh** Says:

    May 17, 2020 at 7:22 am

    Hi first of all thanks for the nice article.

    Unfortunately I didn’t manage to run your example with Vuex. The action updateName doesn’t get triggered. I remmember when I tried Vuex for the first time with plain Javascript, I had no problems, but somehow I thing with typescript it makes things 100 times more difficult.

13.  **Daniel Groh** Says:

    May 17, 2020 at 7:24 am

    I completely agree. I could do this with typescript so easily and now things got much worse in my opnion. Maybe I just have to get used to it.

14.  **Fabian** Says:

May 18, 2020 at 1:05 pm

Can you share a repo for how it works without Vuex? The repos shared here are my only reference. Now newly thinking up a new way of doing things makes my brain explode^^

15.  **Preetish H. Shhantharaja** Says:

May 28, 2020 at 3:22 pm

Hi Daniel,
You are using Vuex-class helpers like the tutorial right?
Please take a look at the github example code. It might help you fix your issue.

https://github.com/preetishhs/vue-typescript-example

Well, The newer version of Vuex will also come with typescript support, It would be much easier then!

16.  **Krishal Shah** Says:

June 23, 2020 at 1:34 am

Thank you so much! It’s really helpful.

17.  **Farhad Mehryari** Says:

July 2, 2020 at 3:10 am

Thanks for this great Article.

18.  **Elena** Says:

August 12, 2020 at 9:24 am

Thank you, Preetish!
Your tutorial was very helpful for me

19.  **Nikhil Das Nomula** Says:

August 12, 2020 at 12:57 pm

Preetish! This is a great article, thanks!

20.  **Bing** Says:

    September 26, 2020 at 8:38 am

    You missed “!” after “name”. As you have the “!” after “msg”, that property doesn’t have the problem. “!” is the “definite assignment assertion”.
    See https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-7.html#definite-assignment-assertions

21.  **Michael Coombes** Says:

    November 6, 2020 at 10:01 pm

    I’m hitting issues now with this, when I use an action I’m getting:

    ERR_ACTION_ACCESS_UNDEFINED: Are you trying to access this.someMutation() or this.someGetter inside an @Action?
    That works only in dynamic modules.
    If not dynamic use this.context.commit(“mutationName”, payload) and this.context.getters[“getterName”]

    In even the simplest examples like the above that are definitely using this.context.commit, anyone got any ideas?

22.  **Preetish H. Shantharaja** Says:

    November 7, 2020 at 3:10 pm

    Hi Michael,
    Unfortunately the library `vuex-module-decorators` is no longer maintained by the author.
    There is a similar issue already opened on github:
    https://github.com/championswimmer/vuex-module-decorators/issues/321

    For a working Vue typescript example please check this repo:
    https://github.com/preetishhs/Vue-typescript-example

    For Nuxt Typescript example:
    https://github.com/preetishhs/nuxt-typescript-example

23.  **Noor** Says:

    November 22, 2020 at 4:00 am

    Great article! very simple and clear example. Thank you for the article. I wish more articles on typescript

24.  **Jerome Villiseck** Says:

December 15, 2020 at 11:19 am

NOTE: Be careful, getters in vuex are not reactive. Loaded just once when the component is created.
To access a data reactively, use this syntax
syntax: this.$store.state.{CLASSNAME}.{MEMBER_OF_CLASS}
example: this.$store.state.User.name
to put in the component which you want access at the data.