


# Use Any Javascript Library With Vue.js

---

 [vuejsdevelopers.com/2017/04/22/vue-js-libraries-plugins](https://vuejsdevelopers.com/2017/04/22/vue-js-libraries-plugins)

Lodash, Moment, Axios, Async...these are useful Javascript libraries that you'll want to utilise in many of your Vue.js apps.

But as your project grows you'll be separating code into single file components and module files. You also may want to run your app in different environments to allow server rendering.

Unless you find an easy and robust way to include those Javascript libraries across your components and module files they're going to be a nuisance!

## How *not* to include a library in a Vue.js project

---

### Global variable

---

The naive way to add a library to your project is to make it a global variable by attaching it to the `window` object:

*entry.js*

```
window._ = require('lodash');
```

*MyComponent.vue*

```
export default {  
  created() {  
    console.log(_.isEmpty() ? 'Lodash everywhere!' : 'Uh oh..');  
  }  
}
```

The case against window variables is a long one, but, specifically to this discussion, they don't work with server rendering. When the app runs on the server the `window` object will be undefined and so attempting to access a property will end with an error.

### Importing in every file

---

Another second-rate method is to import the library into every file:

*MyComponent.vue*

```
import _ from 'lodash';

export default {
  created() {
    console.log(_.isEmpty() ? 'Lodash is available here!' : 'Uh oh..');
  }
}
```

This works, but it's not very DRY and it's basically just a pain: you have to remember to import it into every file, and remove it again if you stop using it in that file. And if you don't setup your build tool correctly you may end up with multiple copies of the same library in your build.

## A better way

---

The cleanest and most robust way to use a Javascript library in a Vue project is to proxy it to a property of the Vue prototype object. Let's do that to add the Moment date and time library to our project:

### *entry.js*

```
import moment from 'moment';
Object.defineProperty(Vue.prototype, '$moment', { value: moment });
```

Since all components inherit their methods from the Vue prototype object this will make Moment automatically available across any and all components with no global variables or anything to manually import. It can simply be accessed in any instance/component from **this.\$moment**:

### *MyNewComponent.vue*

```
export default {
  created() {
    console.log('The time is ' . this.$moment().format("HH:mm"));
  }
}
```

Let's take the time now to understand how this works.

## Object.defineProperty

---

We would normally set an object property like this:

```
Vue.prototype.$moment = moment;
```

You could do that here, but by using `Object.defineProperty` instead we are able to define our property with a descriptor. A descriptor allows us to set some low-level details such as whether or not our property is writeable and whether it shows up during enumeration in a `for` loop and more.

We don't normally bother with this in our day-to-day Javascript because 99% of the time we don't need that level of detail with a property assignment. But here it gives us a distinct advantage: properties created with a descriptor are *read-only* by default.

This means that some coffee-deprived developer (probably you) won't be able to do something silly like this in a component and break everything:

```
this.$http = 'Assign some random thing to the instance method';  
this.$http.get('/');
```

Instead, our read-only instance method protects our library, and if you attempt to overwrite it you will get "TypeError: Cannot assign to read only property".

□

## Free Vue.js Crash Course!

---

Learn what Vue is, what kind of apps you can build with it, how it compares to React & Angular, and more in this *free 30-minute video introduction*.

[Enroll For Free! See our other courses](#)

\$

---

You'll notice that we proxy our library to a property name prefixed with the dollar sign "\$". You've probably also seen other properties and methods like `$refs`, `$on`, `$mount` etc which have this prefix too.

While not required, the prefix is added to properties to remind coffee-deprived developers (you, again) that this is a public API property or method that you're welcome to use, unlike other properties of the instance that are probably just for Vue's internal use.

Being a prototype-based language, there are no (real) classes in Javascript so it doesn't have "private" and "public" variables or "static" methods. This convention is a mild substitute which I think is worthwhile to follow.

*this*

---

You'll also notice that to use the library you use `this.libraryName` which is probably not a surprise since it is now an instance method.

One consequence of this, though, is that unlike a global variable you must ensure you're in the correct scope when using your library. Inside callback methods you can't access the `this` that your library inhabits.

Fat arrow callbacks are a good solution to making sure you stay in the right scope:

```
this.$http.get('/').then(res => {  
  if (res.status !== 200) {  
    this.$http.get('/')  
  
  }  
});
```

Why not make it a plugin?

---

If you're planning to use a library across many Vue projects, or you want to share it with the world, you can build this into your own plugin!

A plugin abstracts complexity and allows you to simply do the following in a project to add your chosen library:

```
import MyLibraryPlugin from 'my-library-plugin';  
Vue.use(MyLibraryPlugin);
```

With these two lines we can use the library in any component just like we can with Vue Router, Vuex and other plugins that utilise `Vue.use`.

Writing a plugin

---

Firstly, create a file for your plugin. In this example I'll make a plugin that adds Axios to your all your Vue instances and components, so I'll call the file *axios.js*.

The main thing to understand is that a plugin must expose an **install** method which takes the Vue constructor as the first argument:

*axios.js*

```
export default {  
  install: function(Vue) {  
  
  }  
}
```

Now we can use our previos method to add the library to the prototype object:

*axios.js*

```
import axios from 'axios';  
  
export default {  
  install: function(Vue,) {  
    Object.defineProperty(Vue.prototype, '$http', { value: axios });  
  }  
}
```

The **use** instance method is all we now need to add our library to a project. For example, we can now add the Axios library as easily as this:

*entry.js*

```
import AxiosPlugin from './axios.js';  
Vue.use(AxiosPlugin);  
  
new Vue({  
  created() {  
    console.log(this.$http ? 'Axios works!' : 'Uh oh..');  
  }  
})
```

Bonus: plugin optional arguments

---

Your plugin install method can take optional arguments. Some devs

might not like calling their Axios instance method `$http` since Vue Resource is commonly given that name, so let's use an optional argument to allow them to change it to whatever they like:

*axios.js*

```
import axios from 'axios';
```

```
export default {  
  install: function(Vue, name = '$http') {  
    Object.defineProperty(Vue.prototype, name, { value: axios });  
  }  
}
```

*entry.js*

```
import AxiosPlugin from './axios.js';  
Vue.use(AxiosPlugin, '$axios');
```

```
new Vue({  
  created() {  
    console.log(this.$axios ? 'Axios works!' : 'Uh oh..');  
  }  
})
```