How to Watch Nested Data in Vue

You have an **array or an object** as a prop, and you want your app to do something whenever that data changes.

So you create a watcher for that property, but Vue doesn't seem to fire the watcher when the nested data changes.

Here's how you solve this.

You need to set `deep` to true when watching an array or object so that Vue knows that it should watch the nested data for changes.

I'll go into more detail on what this looks like in this article, plus some other useful things to know when using `watch` in Vue.

You can also check out the Vue docs on using watchers.

What we'll cover in this article

First we'll do a quick refresher on **what a watcher actually is**.

Second, we have to take a slight detour and **clarify the distinction between computed props and watchers**.

Thirdly, we'll dive into **how you can watch nested data in arrays and objects**. Feel free to skip straight here if you need — you can always come back to the first sections later on.

We'll also go through what you can do with `immediate` and `handler` fields on your watchers. This will take your watcher skills to **the next level**!

But first we have to make sure we have a good foundation.

What is a watch method?

In Vue we can watch for when a property changes, and then do something in response to that change.

For example, if the prop `colour` changes, we can decide to log something to the console:

```
export default {
  name: 'ColourChange',
  props: ['colour'],
  watch: {
    colour()
      console.log('The colour has changed!');
    }
  }
}
```

These watchers let us do all sorts of useful things.

But many times we use a watcher when all we needed was a computed prop.

Should you use watch or computed?

Watched props can often be confused with `computed` properties, because they operate in a similar way. It's even trickier to know when to use which one.

But I've come up with a good rule of thumb.

Watch is for side effects. If you need to change state you want to use a computed prop instead.

A **side effect** is anything that happens outside of your component, or anything asynchronous.

Common examples are:

- Fetching data
- Manipulating the DOM
- Using a browser API, such as local storage or audio playback

None of these things affect your component directly, so they are considered to be *side effects*.

If you aren't doing something like this, you'll probably want to use a computed prop. Computed props are really good for when you need to **update a calculation in response to something else changing**.

However, *there are* cases where you might want to use a watcher to update something in your `data`.

Sometimes it just doesn't make sense to make something a computed prop. If you have to update it from your `<template>` or from a method, it needs to go inside of your `data`. But then if you need to update it in response to a property changing, you *need* to use the watcher.

NOTE: Be careful with using a `watch` to update state. This means that both your component and the parent component are updating — directly or indirectly — the same state. **This can get very ugly very fast.**

Watching nested data — Arrays and Objects

So you've decided that you *actually* need a watcher.

But you're watching an array or an object, and it isn't working as you had expected.

What's going on here?

Let's say you set up an array with some values in it:

```
const array = [1, 2, 3, 4];
// array = [1, 2, 3, 4]
```

Now you update the array by pushing some more values into it:

```
array.push(5);
array.push(6);
array.push(7);
// array = [1, 2, 3, 4, 5, 6, 7]
```

Here's the question: has `array` changed?

Well, it's not that simple.

The *contents* of `array` have changed, but the variable `array` still points to the same Array object. The array container hasn't changed, but **what is inside of the array** has changed.

So when you watch an array or an object, Vue has no idea that **you've changed what's inside** that prop. You have to tell Vue that you want it to inspect inside of the prop when watching for changes.

You can do this by setting `deep` to `true` on your watcher and rearranging the handler function:

```
export default {
  name: 'ColourChange',
  props: {
    colours: {
      type: Array,
      required: true,
    },
  },
  watch: {
    colours: {
      // This will let Vue know to look inside the array
      deep: true,

      // We have to move our method to a handler field
      handler()
        console.log('The list of colours has changed!');
      }
    }
  }
}
```

Now Vue knows that it should also keep track of what's inside the prop when it's trying to detect changes.

What's up with the `handler` function though?

Just you wait, we'll get to that in a bit. But first let's cover something else that's important to know about Vue's watchers.

Immediate

A watcher will only fire when the prop's value changes, but we often need it to fire once on startup as well.

Let’s say we have a `MovieData` component, and it fetches data from the server based on what the `movie` prop is set to:

```
export default {
  name: 'MovieData',
  props: {
    movie: {
      type: String,
      required: true,
    }
  },
  data() {
    return {
      movieData: {},
    }
  },

  watch: {
    // Whenever the movie prop changes, fetch new data
    movie(movie) {
      // Fetch data about the movie
      fetch(`/${movie}`).then((data) => {
        this.movieData = data;
      });
    }
  }
}
```

Now, this component will work wonderfully. Whenever we change the `movie` prop, our watcher will fire, and it will fetch the new data.

Except we have one *small* problem.

Our problem here is that when the page loads, `movie` will be set to some default value. But since the prop hasn’t changed yet, the watcher isn’t fired. This means that the data isn't loaded until we select a different movie.

So how do we get our watcher to fire immediately upon page load?

We set `immediate` to true, and move our handler function:

```
watch: {
  // Whenever the movie prop changes, fetch new data
  movie: {
    // Will fire as soon as the component is created
    immediate: true,
    handler(movie) {
      // Fetch data about the movie
      fetch(`/${movie}`).then((data) => {
        this.movieData = data;
      });
    }
  }
}
```

Okay, so now you've seen this `handler` function twice now. I think it's time to cover what that is.

Handler

Watchers in Vue let you specify three different properties:

- `immediate`
- `deep`
- `handler`

We just looked at the first two, and the third isn't too difficult either. You've probably already been using it without realizing it.

The property `handler` specifies the function that will be called when the watched prop changes.

What you've probably seen before is the shorthand that Vue lets us use if we don't need to specify `immediate` or `deep`. Instead of writing:

```
watch: {
  movie: {
    handler(movie) {
      // Fetch data about the movie
      fetch(`/${movie}`).then((data) => {
        this.movieData = data;
      });
    }
  }
}
```

We can use the shorthand and just specify the function directly:

```
watch: {
  movie(movie) {
    // Fetch data about the movie
    fetch(`/${movie}`).then((data) => {
      this.movieData = data;
    });
  }
}
```

What's cool is that Vue also let's us use a `String` to name the method that handles the function. This is useful if **we want to do something when two or more props change**.

Using our movie component example, let's say we fetch data based on the `movie` prop as well as the `actor`, then this is what our methods and watchers would look like:

```
watch: {
  // Whenever the movie prop changes, fetch new data
  movie {
    handler: 'fetchData'
  },
  // Whenever the actor changes, we'll call the same method
  actor: {
    handler: 'fetchData',
  }
},

methods: {
  // Fetch data about the movie
  fetchData() {
    fetch(`/${this.movie}/${this.actor}`).then((data) => {
      this.movieData = data;
    });
  }
}
```

This makes things a little cleaner if we are watching multiple props to do the same side-effect.

If you enjoyed this article or have any comments, let me know by replying to this tweet!

26 Time Saving Tips for Vue

We all hate wasting our time.

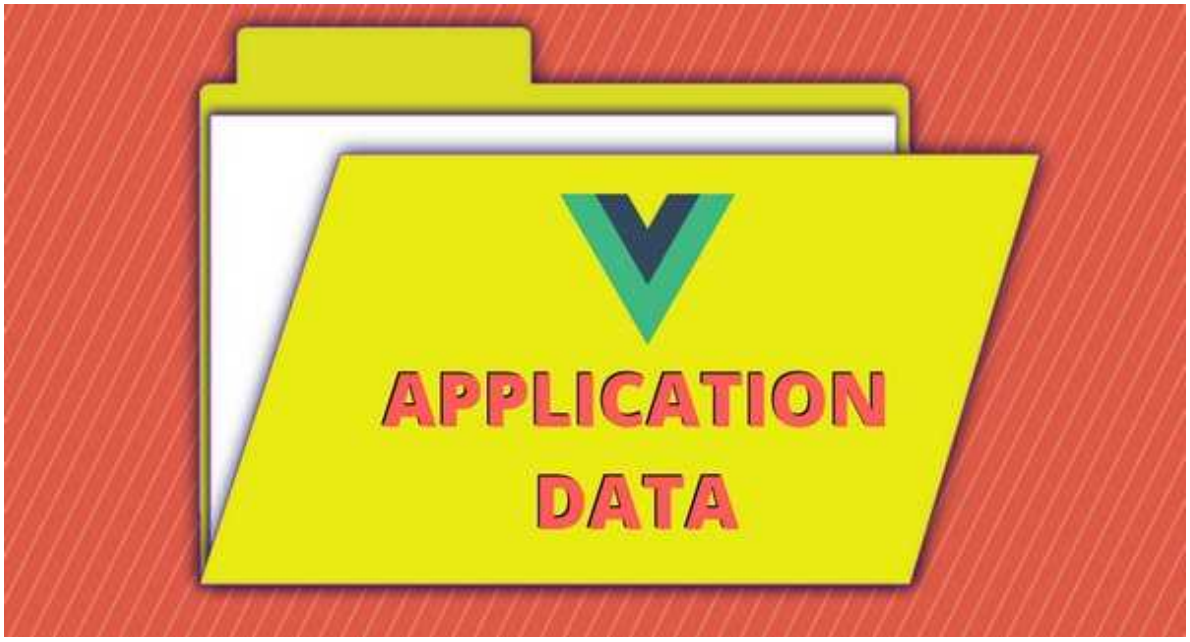
When writing Vue applications, we waste our time by doing things the *wrong* way, when we could have been doing it the *right* way from the start.

But it’s hard to know what things we should be learning.

That’s why I put together this list of **26 articles that will help you save time**, by teaching you how to avoid some common time-wasters.

And once you’ve gone through this list, you can share it with others so that you can help them save time too!

1. Use Vuex before it’s too late



If you’re building a medium to large sized app, your state — all the data you need to keep track of — can get pretty complicated.

It’s not just the amount of information you have to deal with either. The different interactions between your state and all of the different ways your state can change only add to this complexity.

Managing state is a difficult task! So many bugs and wasted time are due to very complicated state and logic.

That’s why Evan You created Vuex to go along with Vue.

It’s not necessary to use for small projects or projects with simple state. But for larger projects, it’s absolutely essential.

If you want to learn more about the problems that Vuex solves, and how to use it in your app, check out WTF is Vuex? A Beginner’s Guide To Vue’s Application Data Store by Anthony Gore.

And here’s an interesting fact.

Evan You originally intended Vuex to be pronounced “vukes” — rhyming with “pukes”. But so many people pronounced it as “view-ex” that he changed his mind .

2. Understand how Vue component instances work

Vue has a very clever design to improve performance and reduce its memory footprint.

While not necessary, understanding how this works under the hood will only help you as you build more and more Vue components.

Besides, it's really interesting!

In this brief but very informative article by Joshua Bemenderfer, learn how Vue creates component instances: Understanding Vue.js Component Instancing.

3. Force Vue to re-render — the right way

In 99% of cases where something doesn't rerender properly, it's a reactivity problem.

So if you're new to Vue, you *definitely* need to learn as much as you can about reactivity. I see it as being one of the biggest sticking points for new developers.

However, sometimes you need a sledgehammer to get things done and ship your code. Unfortunately, deadlines don't move themselves.

And *sometimes*, forcing a component to rerender is actually the best way to do it (but very very rarely).

By far my most popular article, I've written about the proper way to rerender a component.

4. Vue doesn't handle multiple root nodes — yet

Not all components make sense to have a single root node.

For example, if you're rendering a list of items, it could make way more sense to simple return the list of nodes as an array. Why unnecessarily wrap it in a `ol` or `ul` tag?

This is called a *fragment*.

Currently Vue doesn't support fragments, although there will be support for them in Vue 3.

It's something that React has had for awhile now, but it took a rewrite of the rendering system in order for them to implement this. Vue is in the same situation.

However, you can use functional components to get around this issue while we wait for Vue 3.0 to be released. You can read more about that in Can A Vue Template Have Multiple Root Nodes (Fragments)? by Anthony Gore.

5. Validate your forms the easy way — using Vuelidate

At one of my previous jobs, I'm pretty sure my official job description was:

Builds datatables and forms

It felt like all I was doing every day was building form after form after form.

But it makes sense. Forms are the main way that we get input from the user, and they are absolutely crucial to our applications working well. So we end up writing lots of them.

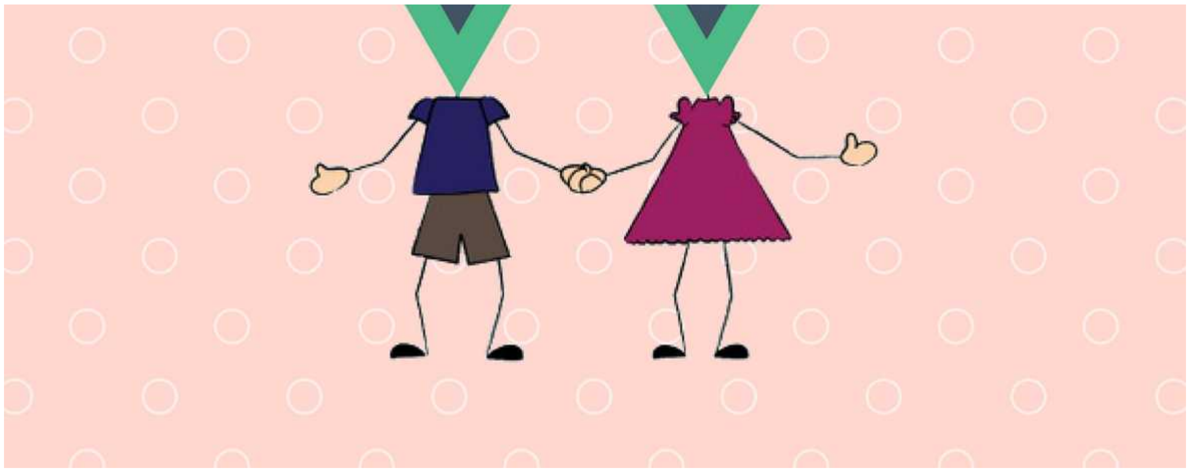
However, forms are also really tricky to build. On the surface it seems like they should be fairly straightforward to write. But as you start adding validation rules and other logic, it can quickly turn into a nightmare.

This is where Vuelidate comes in.

It’s a library that makes it super easy to add custom validation, and does all the heavy lifting for you.

Learn how to setup Vuelidate by reading Simple Vue.js Form Validation with Vuelidate by Dobromir Hristov.

6. Build components that play nicely with each other



The absolutely *worst* feeling is realizing that you built your component the wrong way, and now you have refactor it completely.

You don't want to overengineer your code, but many times things like this can avoided from the start.

Kevin Ball has written an article outlining several different things to keep in mind as you write your components to keep them playing nice with others.

Check out his article: How To Build Vue Components That Play Nice

7. Don't write one-off transitions — make them reusable

Transitions are a really cool feature in Vue. If you haven't had a chance to check them out, they're a really easy way to add nice animations into your app.

But you don't want to keep re-writing the same thing over and over again, do you?

In a great article, Cristi Jora shows us how we can write a component to make our transitions reusable. It also demonstrates some great concepts for how we can make our code more reusable, and can apply to other parts of your app as well.

Check out the article here: Creating Reusable Transitions in Vue

8. Learn how to use Axios for data fetching

Almost every app needs to fetch or post data.

The most popular library to help us do that these days is Axios. It is really configurable, and makes working with external data *so* much easier.

You can just use the browser's built-in `fetch` most of the time. But you'll probably end up writing a wrapper for it anyways, to make common cases more convenient to deal with.

Might as well just start off on the right foot and use axios from the beginning then!

Learn how to integrate `axios` into your Vue app by reading Vue.js REST API Consumption with Axios from Joshua Bemenderfer.

9. Use vue-router to handle client-side routing

If you’re doing client-side routing, hand-rolling your own solution isn’t that difficult.

It’s actually pretty simple to match routes, then swap between different components.

But just using `vue-router` is *so much easier*.

It’s also an official Vue package, so you know it will always work really well with Vue.

And once you start dealing with:

- queries
- route params
- nested routes
- dynamic route matching
- transitions

...which you probably will, writing your own solution gets to be very cumbersome.

Instead, just check out this guide from Ed Zynda on Getting Started With Vue Router

10. Create filters to reuse formatting

Formatting data to display on screen can get annoying.

If you're dealing with lots of numbers, percentages, dates, currencies, names, or anything else like that, you'll likely have functions that format that data for you.

Vue comes with this great feature called *filters*, which was inspired by Angular.

They let you easily encapsulate these formatting functions and use them in your template with a really clean syntax.

But don't just take my word for it.

Check out this very detailed article from Rachid Laasri, which has tons of examples on how to write your own filters: How to Create Filters in Vue.js with Examples

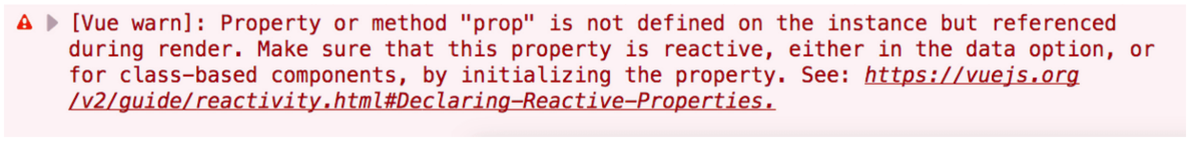
11. Make sure to avoid annoying errors and warnings

As web development has gotten more complex, so have our tools. These days we have linters, editors, type checkers, and all sorts of things that save us time by telling us our mistakes (almost) as soon as we make them.

Vue also has really good warning and error messages, but if you keep getting them it can be annoying.

Why not just avoid them altogether?

One of the most common warnings I got when I was first learning Vue was this one:



```
[Vue warn]: Property or method "prop" is not defined on the instance but referenced
during render. Make sure that this property is reactive, either in the data option, or
for class-based components, by initializing the property. See: https://vuejs.org
/v2/guide/reactivity.html#Declaring-Reactive-Properties.
```

Luckily, I wrote an entire article on what causes this, and more importantly, how to avoid it altogether!

12. Don’t be afraid of JSX — it’s extremely powerful

A lot of people are intimidated by JSX.

I get it.

It has a weird syntax, and it can be difficult to wrap your head around how it's used.

But sometimes — especially when writing higher-level reusable components — a template just doesn't cut it. You need to take advantage of the full power of the `render` method.

And JSX is one of the easiest ways of doing that.

Samuel Oloruntoba has written a great introduction to JSX and why exactly it is great that Vue has support for it: Using JSX with Vue and Why You Should Care

13. Figure out how to react to mouse hover

In CSS it's pretty easy to change things on `hover`. We just use the `:hover` psuedo-class:

```css
.item {
  background: blue;
}

.item:hover {
  background: green;
}
```

In Vue it gets a little trickier, because we don't have this functionality built in.

We have to implement most of this ourselves.

But don't worry, it's not *that* much work.

I've written an in-depth article on using hover in Vue. It covers a lot of different things you'll want to know:

- How to implement a **hover effect** in Vue
- How to **show an element** on mouseover
- How to dynamically **update classes** with a mouseover
- How to do this even on **custom Vue components**

Check out the article: How to Implement a Mouseover or Hover in Vue

14. Add v-model support to custom components

As web developers, our jobs revolve around getting data from inputs.

Vue gives us `v-model`, which is some syntactic sugar that creates a two-way data-binding for us. This is great for inputs, as it simplifies working with them a lot.

But did you know you can add `v-model` support to your own components?

Joshua Bemenderfer shows us how this can be done in Adding v-model Support to Custom Vue.js Components.

15. Fix “this is undefined” error

Perhaps one of the most common errors to run into is this one.

I used to run into this one *all* of the time. But now I know exactly what I was doing wrong. It has do with the type of function you’re using, and how you’re using it.

But I won’t get into that here.

Learn how to fix the “this is undefined” error, and get on with your life!

16. Use an off the shelf CSS framework

Getting all of your CSS *just right* can take an extraordinary amount of time.

My suggestion is to just use a CSS framework, where most of the work is already done for you.

All of the styling, colours, drop shadows, and aesthetic elements are already worked out. No need to learn graphic design! On top of that, they all come with tons of CSS styles to help you with layout, forms, and other common elements like buttons, popups, alert boxes, and so much more.

The best part is the variety.

There are tons of great ones to pick from:

- Tailwind
- Bootstrap
- Bulma
- Foundation

And don’t waste any time trying to figure out how to integrate them into your app. Dave Berning has written a great article to help get you started: Integrating and Using CSS Frameworks with Vue.js.

17. Watching nested data in Vue

Watchers are a really great feature in Vue. They make adding side-effects really clean, and they’re easy to use (like the rest of Vue).

Except when you try and use them on an array or object.

Nested data structures like arrays and objects are a little trickier to work with.

Just yesterday, I spent at least 30 minutes helping a co-worker figure out an issue with his Vue component.

Turned out to be an issue with nested data .

Because it's such a common problem, I've written an in-depth article on how to watch nested data, which also goes into some of the more advanced features that watchers give you.

18. Show loading and error states on your async components

You probably hear a lot about web performance these days — and for good reason.

The easiest way to get your application bundle smaller is to split out the code into multiple, smaller, chunks. **Vue comes with first-class support for this**, which is really cool!

But the user's experience can suffer if we don't provide a good loading state while fetching the component. And we also want to show a good error state if something goes wrong.

Luckily, integrating these isn't too difficult.

Joshua Bemenderfer shows us exactly how this can be done in Showing Loading & Error States with Vue.js Async Components.

19. Clean up your props for goodness sake!

Some components only require a few props, but others require passing many, many, props.

Eventually this can get pretty messy.

```
<v-btn
  color="primary"
  href="https://michaelnthiessen.com"
  small
  outline
  block
  ripple
>
  Hello
</v-btn>
```

But there are several different ways that we can clean this up. Not only will this make our code easier to look at, but it will also be easier to understand, and modify in the future.

Alex Jover Morales has written an excellent article outlining the different ways you can clean up your props. Check it out: Passing Multiple Properties to a Vue.js Component.

20. Don’t confuse computed props and watchers

I know that most people don’t accidentally write a computed prop when they meant to write a watcher.

That’s just silly.

But I see lots of people who use a watcher when they should instead be using a computed prop. Or using a computed prop when a watcher would be a better fit.

Although they seem like they do similar things, watchers and computed props are actually quite different.

My rule of thumb: make it a computed prop!

However, if you want to know more, I wrote an article on the differences between computed props and watchers.

21. Beware of some common pitfalls

Like any piece of technology, Vue has some areas that can catch you off guard.

I cannot tell you how many hours I wasted because I didn’t understand some of these things. But learning them didn’t take that long either.

If only I had known!

Instead of struggling through these gotchas like I did when I was learning Vue, you can avoid most of these.

Read Common Vue.js Gotchas by Joshua Bemenderfer, and save yourself a lot of frustration!

22. Learn the differences between props and data

Vue comes with two different ways of storing variables, props and data.

These can be confusing at first, since they seem like they do similar things, and it's not clear when to use one vs the other.

The answer involves reactivity, naming collisions, and the direction of data flow (spoiler: it's down).

In my article on the difference between props and data, I also go into detail on where you would use each one, and how you would use them together.

It's a really important topic to grasp, so make sure you understand it!

23. Properly call methods when the page loads

It's an extremely common pattern in web apps to perform some sort of logic as soon as the page is loaded. Often you're fetching data, or even manipulating the DOM somehow.

But there are a lot of *wrong* ways of doing this with Vue.

Lucky for us, Vue gives us lifecycle hooks that let us do this in a really clean and simple way.

You can check out this in-depth article on how to do this the proper way. The article also goes deep into what lifecycle methods are, and how we can hook into them.

24. Understand how to pass a function as a prop

Short answer: **you don't**.

But that's a hugely unsatisfying answer, so of course I'll expand on it.

This question comes up for 2 main reasons:

1. You want to communicate from the child to the parent
2. You need to abstract your component behaviour in a specific way

In React we pass functions around all the time, and that's how we would solve both of these problems. But Vue gives us two separate mechanisms for solving these two problems.

These mechanisms are events and scoped slots.

If you want to learn how to use these to solve either of these problems, as well as the differences between how React and Vue work (and more!), check out this detailed article I wrote about it: How to Pass a Function as a Prop in Vue

25. Learn why mutating props is an anti-pattern

This is an error you may have seen:

> *Avoid mutating a prop directly since the value will be overwritten whenever the parent component re-renders. Instead, use a data or computed property based on the prop's value.*

Why is mutating a prop directly not allowed in Vue, and what do you do instead?

I've written an entire article on this subject. I also go into what causes this error, and how to avoid it.

The article also touches on how to avoid getting this error when using `v-model`, as there are some specific things that can be confusing there.

26. Dynamically add CSS classes

Vue incorporates HTML and Javascript together in a really beautiful way, but we can't forget about CSS.

CSS is the thing that really makes our apps shine, and is very powerful in it's own right.

A very common pattern in web apps is to add and remove classes from elements based on the state of our application. We do this to show a button is disabled, to animate elements like loading spinners, and a ton of other things.

Vue gives us a lot of options in choosing how to dynamically add and remove CSS classes based on what's going on in our application. **Knowing what these options are gives you more tools, and you'll be able to write better code because of it**.

I wrote an article that covers all of the different ways you can dynamically add and remove classes in Vue. We go over array syntax and object syntax, using Javascript expressions to calculate the class, and adding dynamic classes to custom components (you don't need to add a custom `class` prop!). You can even generate your class names on the fly!

If you enjoyed this article, please share it with others who may enjoy it as well!

It really helps encourage me to keep writing stuff like this.

Thanks!

How to Dynamically Add a Class Name in Vue

Being able to add a dynamic class name to your component is really powerful.

It lets you write custom themes more easily, add classes based on the state of the component, and also write different variations of a component that rely on styling.

Adding a dynamic class name is as simple as adding the prop `:class="classname"` to your component. Whatever `classname` evaluates to will be the class name that is added to your component.

Of course, there is *a lot* more we can do here with dynamic classes in Vue.

We'll cover a lot of stuff in this article:

- Using static and dynamic classes in Vue
- How we can use regular Javascript expressions to calculate our class
- The array syntax for dynamic class names
- The object syntax (for more variety!)
- Generating class names on the fly
- How to use dynamic class names on custom components

Not only that, but at the end I'll show you a simple pattern for using computed props to help clean up your templates. This will come in handy once you start using dynamic class names everywhere!

Let’s dive in!

Static and Dynamic Classes

In Vue we can add both static and dynamic classes to our components.

Static classes are the boring ones that never change, and will always be present on your component. On the other hand, **dynamic** classes are the ones we can add and remove things change in our application.

If we wanted to add a static class, it’s exactly the same as doing it in regular HTML:

```
<template>
  <span class="description">
    This is how you add static classes in Vue.
  </span>
</template>
```

Dynamic classes are very similar, but we have to use Vue’s special property syntax, `v-bind`:

```
<template>
  <span v-bind:class="'description'">
    This is how you add static classes in Vue.
  </span>
</template>
```

You’ll notice we had to add extra quotes around our dynamic class name.

This is because the `v-bind` syntax takes in whatever we pass as a Javascript value. Adding the quotes makes sure that Vue will treat it as a string.

Vue also has a shorthand syntax for `v-bind`:

```
<template>
  <span :class="'description'">
    This is how you add static classes in Vue.
  </span>
</template>
```

What’s really neat is that you can even have both static and dynamic classes on the same component.

This let’s you have some static classes for the things you know won’t change, like positioning and layout, and dynamic classes for your theme:

```
<template>
  <span
    class="description"
    :class="theme"
  >
    This is how you add static classes in Vue.
  </span>
</template>
```

```
export default {
  data() {
    return {
      theme: 'blue-theme',
    };
  }
};
```

```
.blue-theme {
  color: navy;
  background: white;
}
```

In this case, `theme` is the variable that contains the classname we will apply (remember, it's treating it as Javascript there).

Using a Javascript Expression

Since `v-bind` will accept any Javascript expression, we can do some pretty cool things with it.

Guard Expressions

There is a cool trick using the logical `&&` that allows us to conditionally apply a class:

```
<template>
  <span
    class="description"
    :class="useTheme && theme"
  >
    This is how you add dynamic classes in Vue.
  </span>
</template>
```

This is known as a guard expression.

But how does it work?

Here we have the variable `useTheme` which is a boolean, and `theme` is the value of the theme class.

In Javascript, the AND operator will short-circuit if the first value is `false`.

Since both values need to be `true` in order for the expression to be `true`, if the first is `false` there is no point in checking what the second one is, since we already know the expression evaluates to `false`.

So if `useTheme` is `false`, the expression evaluates to `false` and no dynamic class name is applied.

However, if `useTheme` is true, it will also evaluate `theme`, and the expression will evaluate to the value of `theme`. This will then apply the value of `theme` as a classname.

The Many Ways I Use Ternaries

We can do a similar trick with ternaries.

If you aren't familiar, a ternary is basically a short-hand for an if-else statement.

They look like this:

```
const result = expression ? ifTrue : ifFalse;
```

Sometimes though, we'll format them like this for readability:

```
const result = expression
  ? ifTrue
  : ifFalse;
```

If `expression` evaluates to `true`, we get `ifTrue`. Otherwise we will get `ifFalse`.

Their main benefit is that they are concise, and count as only a single statement. This lets us use them inside of our templates.

Ternaries are useful if we want to decide between two different values:

```
<template>
  <span
    class="description"
    :class="darkMode ? 'dark-theme' : 'light-theme'"
  >
    This is how you add dynamic classes in Vue.
  </span>
</template>
```

If `darkMode` is `true`, we apply `dark-theme` as our class name. Otherwise we choose `light-theme`.

Now let's figure out how to add multiple dynamic class names at the same time!

Using the Array Syntax

If there are lots of different classes you want to add dynamically, you can use arrays or objects. Both are useful, but we'll cover arrays first.

Since we are just evaluating a javascript expression, you can combine the expressions we just learned with the array syntax:

```
<template>
  <span
    class="description"
    :class="[
      fontTheme,
      darkMode ? 'dark-theme' : 'light-theme',
    ]"
  >
    This is how you add dynamic classes in Vue.
  </span>
</template>
```

What's going on here?

We are using the array to set two dynamic class names on this element.

The value of `fontTheme` is a classname that will change how our fonts look.

From our previous example, we can still switch between light and dark themes using our `darkMode` variable.

Using the Object Syntax

We can even use an object to define the list of dynamic classes, which gives us some more flexibility.

For any key/value pair where the value is `true`, it will apply the key as the classname.

Let’s look at an example of the object syntax:

```
<template>
  <span
    class="description"
    :class="{
      'dark-theme': darkMode,
      'light-theme': !darkMode,
    ]"
  >
    This is how you add dynamic classes in Vue.
  </span>
</template>
```

Our object contains two keys, `dark-theme` and `light-theme`. Similar to the logic we implemented before, we want to switch between these themes based on the value of `darkMode`.

When `darkMode` is `true`, it will apply `dark-theme` as a dynamic class name to our element. But `light-theme` won’t be applied because `!darkMode` will evaluate to `false`.

The opposite happens when `darkMode` is set to false. We get `light-theme` as our dynamic classname instead of `dark-theme`.

> *It is common convention to use dashes — or hyphens — in CSS classnames. But in order to write the object keys with dashes in Javascript, we need to surround it in quotes to make it a string.*

Now we've covered the basics of dynamically adding classes to Vue components.

How do we do this with our own custom components?

Using with Custom Components

Let's say that we have a custom component that we are using in our app:

```
<template>
  <MovieList
    :movies="movies"
    :genre="genre"
  />
</template>
```

If we want to dynamically add a class that will change the theme, what would we do?

It's actually really simple.

We just add the `:class` property like before!

```
<template>
  <MovieList
    :movies="movies"
    :genre="genre"
    :class="darkMode ? 'dark-theme' : 'light-theme'"
  />
</template>
```

The reason this works is that Vue will set `class` on the root element of `MovieList` directly.

When you set props on a component, Vue will compare those props to what the component has specified in it’s `props` section. If there is a match, it will pass it along as a normal prop. Otherwise, Vue will add it to the root DOM element.

Here, because `MovieList` didn’t specify a `class` property, Vue knows that it should set it on the root element.

There are some even more advanced things we can do with dynamic class names though...

Generating Class Names on the Fly

We’ve seen a lot of different ways that we can dynamically *add* or *remove* class names.

But what about dynamically *generating* the class name itself?

Let’s say you have a `Button` component, with 20 different CSS styles for all of your different types of buttons.

That’s a lot of variation!

You probably don’t want to spend your whole day writing out every single one, along with the logic to turn it on and off. This would also be a nasty mess when it comes time to update the list!

Instead, we’ll **dynamically generate the name of the class** we want to apply.

A simple version of this you’ve already seen:

```
<template>
  <span
    class="description"
    :class="theme"
  >
    This is how you add static classes in Vue.
  </span>
</template>
```

```
export default {
  data() {
    return {
      theme: 'blue-theme',
    };
  }
};
```

```
.blue-theme {
  color: navy;
  background: white;
}
```

We can set a variable to contain the string of whatever class name we want.

If we wanted to do this for our `Button` component, we could do something simple like this:

```
<template>
  <button
    @click="$emit('click')"
    class="button"
    :class="theme"
  >
    {{ text }}
  </button>
</template>
```

```
export default {
  props: {
    theme: {
      type: String,
      default: 'default',
    }
  }
};
```

```css
.default {}

.primary {}

.danger {}
```

Now, whoever is using the `Button` component can simply set the `theme` prop to whatever theme they want to use.

If they don't set any, it will add the `.default` class.

If they set it to `primary`, it will add the `.primary` class.

Cleaning Things Up With Computed Props

Eventually the expressions in our template will get too complicated, and it will start to get very messy and hard to understand.

Luckily, we have an easy solution for that.

If we convert our expressions to computed props, we can move more of the logic *out of the template* and clean it up:

```
<template>
  <MovieList
    :movies="movies"
    :genre="genre"
    :class="class"
  />
</template>
```

```
export default {
  computed: {
    class() {
      return darkMode ? 'dark-theme' : 'light-theme';
    }
  }
};
```

Not only is this much easier to read, but it's also easier to add in new functionality and refactor in the future.

This pattern — moving things from the template into computed props — works with all sorts of things as well. It's one of the best tricks for cleaning up your Vue components!

Vue Error: Avoid Mutating a Prop Directly

You probably found this article because you've gotten this confusing error:

I'll show you a simple pattern you can use to fix this error — and never see it again.

By the end of this article you'll learn:

- A **simple pattern for fixing this issue**
- What this error means, and what causes it
- Why **mutating props is an anti-pattern**
- How to avoid this when using `v-model`

How is this caused?

Here is the error message in full:

> *Error message: Avoid mutating a prop directly since the value will be overwritten whenever the parent component re-renders. Instead, use a data or computed property based on the prop's value.*

The docs also explain what's going on here.

This error is caused by taking a prop that is passed in by the parent component, and then changing that value.

It applies equally to objects, arrays, and strings and numbers — and any other value you can use in Javascript.

Changing the value in a child component won't change it in the parent component, but it's a symptom of not having thought out your component design clearly enough.

We'll cover how to fix this using a simple pattern in the last part of the article, so hold on until we get there!

Here's what it might look like to accidentally mutate a prop:

```
export default {
  props: {
    movies: Array,
    user: Object,
    searchQuery: String,
  }
}
```

We have 3 different props defined: `movies`, `user`, and `searchQuery`, all of different types.

Before rendering the list of movies we'll sort it:

```
export default {
  props: {
    movies: Array,
    user: Object,
    searchQuery: String,
  },
  methods: {
    sortMovies() {
      this.movies = this.movies.sort();
    }
  }
```

We can also update our search query as the user is typing:

```
export default {
  props: {
    movies: Array,
    user: Object,
    searchQuery: String,
  },
  methods: {
    sortMovies() {
      this.movies = this.movies.sort();
    },
    search(query) {
      this.searchQuery = query;
    }
  }
}
```

And while we're at it, we'll want to update the list of favourite movies for the user:

```
export default {
  props: {
    movies: Array,
    user: Object,
    searchQuery: String,
  },
  methods: {
    sortMovies() {
      this.movies = this.movies.sort();
    },
    search(query) {
      this.searchQuery = query;
    },
    addToFavourites(movie) {
      this.user.favourites.push(movie);
    }
  }
}
```

All 3 of these methods mutate props!!

It's easy to accidentally do this — some of these methods don't really even *look* like they're mutating anything.

But they are, and thankfully Vue warns us.

As you can see, it's an easy mistake to make, but why is it considered bad practice in the first place?

Mutating props in Vue is an anti-pattern

Yes, in Vue, mutating props like this is considered to be an *anti-pattern*.

Meaning — please don't do it, or you'll cause a lot of headaches for yourself.

Why an anti-pattern?

In Vue, we pass data down the the component tree using props. A parent component will use props to pass data down to it's children components. Those components in turn pass data down another layer, and so on.

Then, to pass data back up the component tree, we use events.

We do this because it ensures that each component is isolated from each other. From this **we can guarantee a few things that help us in thinking about our components**:

- Only the component can change it's own state
- Only the parent of the component can change the props

If we start seeing some weird behaviour, knowing with 100% certainty where these changes are coming from makes it much easier to track down.

Keeping these rules makes our components simpler and easier to reason about.

But if we mutate props, we are breaking these rules!

Props are overwritten when re-rendering

There is another thing to keep in mind.

When Vue re-renders your component — which happens every time something changes — it will overwrite any changes you have made to your props.

This means that even if you try to mutate the prop locally, Vue will keep overwriting those changes.

Not a very good strategy, even if you don’t think that this is an anti-pattern.

Modifying value in the component

Now we get to the main reason why someone might want to mutate a prop.

There are many situations where we need to take the prop that we are passed, and then do something extra with it.

Maybe you need to take a list and sort it, or filter it.

Maybe it’s taking some numbers and summing them together, or doing some other calculation with them.

Well, we *still* don't mutate the prop.

Instead, you can use the always useful computed prop to solve the same problem.

Simple example

We'll start with a simple example, and then move on to something a little more interesting.

In our `ReversedList` component we will take in a list, reverse it, and render it to the page:

```
<template>
  <ul>
    <li v-for="item in list" />
  </ul>
</template>
```

```
export default {
  name: 'ReversedList',
  props: {
    list: {
      type: Array,
      required: true,
    }
  },
  created() {
    // Mutating the prop :(
    this.list = this.list.reverse();
  }
};
```

This is an example of a component that, although functional, isn't written very well.

In fact, it isn't completely functional either.

It will only reverse the initial list it is given. If the prop is ever updated with a new list, that won't be reversed .

But, we can use a computed prop to clean this component up!

```
<template>
  <ul>
    <li v-for="item in reversedList" />
  </ul>
</template>
```

```
export default {
  name: 'ReversedList',
  props: {
    list: {
      type: Array,
      required: true,
    }
  },
  computed: {
    reversedList() {
      return this.list.reverse();
    }
  }
};
```

We setup the computed prop `reversedList`, and then swap that out in our `li` tag.

The functionality is also fixed now, as `reversedList` will be recomputed every time that `list` is updated.

We're just getting started though. Let's move on to something a little more complicated.

The (more) complicated example

Okay, so you might have already known to use computed props like we just showed.

But what if you don't always want to use the computed value?

How do we switch between using the prop passed in from the parent, and using the computed prop?

Let’s expand our example.

Now we will have a button in our component that will toggle reversing the list. This way we will be switching between using the list as-is from the parent, and using the computed reversed list:

```
<template>
  <div>
    <!-- Add in a button that toggles our `reversed` flag -->
    <button @click="reversed = !reversed">Toggle</button>
    <ul>
      <li v-for="item in reversedList" />
    </ul>
  </div>
</template>
```

```
export default {
  name: 'ReversedList',
  props: {
    list: {
      type: Array,
      required: true,
    }
  },
  data() {
    return {
      // Define a reversed data property
      reversed: false,
    };
  },
  computed: {
    reversedList() {
      // Check if we need to reverse the list
      if (this.reversed) {
        return this.list.reverse();
      } else {
        // If not, return the plain list passed in
        return this.list;
      }
    }
  }
};
```

First, we define a variable in our reactive data called `reversed`, which keeps track of whether or not we should be showing the reversed list.

Second, we add in a button. When the button is clicked, we toggle `reversed` between `true` and `false`.

Third, we update our computed property `reversedList` to also rely on our `reversed` flag. Based on this flag we can decide to reverse the list, or just use what was passed in as a prop.

Here we see a bit more of the power and flexibility of computed props.

We don't have to tell Vue that we need to update `reversedList` when either `reversed` or `list` change, it just *knows*.

Getting tripped up by v-model

It's also a little confusing how `v-model` works with props, and many people run into issues with it. This error is not uncommon when using it.

Essentially `v-model` takes care of passing down your prop as well as listening to the change event for you. It also takes care of some edge cases, so you don't have to think too much more.

The important thing here, is that `v-model` is mutating the value that you give to it.

This means that **you can't use props with v-model**, or you'll get this error.

Example of what not to do:

```
<template>
  <input v-model="firstName" />
</template>
```

```
export default {
  props: {
    firstName: String,
  }
}
```

Instead, you need to handle the input changes yourself, or include the input in the parent component.

You can check out what the docs have to say about `v-model` for more information.

Conclusion

Now you know why mutating props isn't a good idea, as well as how to use computed props instead.

In my opinion, computed props are one of the most useful features of Vue. You can do a ton of very useful patterns with them.

If this still didn't fix your problem, or you have questions about this article, please reach out to me on Twitter.

I'm always available to help!

How to Pass a Function as a Prop in Vue

It’s a pretty common question that newer Vue developers often ask.

You can pass strings, arrays, numbers, and objects as props.

But can you pass a function as a prop?

While you can pass a function as a prop, this is almost always a bad idea. Instead, there is probably a feature of Vue that is designed exactly to solve your problem.

If you keep reading you’ll see what I mean.

In this article I’ll show you:

- How to pass a function as a prop — even though you probably shouldn’t
- Why React and Vue are different when it comes to passing methods as props
- Why events or scoped slots might be better solutions
- How you can access a parent’s scope in the child component

First, even though I probably shouldn’t, I’ll show you how to do it.

Passing a function as a prop

Taking a function or a method and passing it down to a child component as a prop is relatively straightforward. In fact, it is exactly the same as passing any other variable:

```
<template>
  <ChildComponent :function="myFunction" />
</template>
```

```
export default {
  methods: {
    myFunction() {
      // ...
    }
  }
};
```

But as I said earlier, this is something you shouldn’t ever be doing in Vue.

Why?

Well, Vue has something better...

React vs Vue

If you’re coming from React you’re used to passing down functions all of the time.

In React you'll pass a function from a parent to a child component, so the child can communicate back up to the parent. Props and data flow down, and function calls flow up.

Vue, however, has a different mechanism for achieving child -> parent communication.

We use events in Vue.

This works in the same way that the DOM works — providing a little more consistency with the browser than what React does. Elements can emit events, and these events can be listened to.

So even though it can be tempting to pass functions as props in Vue, it's considered an anti-pattern.

Using events

Events are how we communicate to parent components in Vue.

Here is a short example to illustrate how events work.

First we'll create our child component, which emits an event when it is created:

```
// ChildComponent
export default {
  created() {
    this.$emit('created');
  }
}
```

In our parent component, we will listen to that event:

```
<template>
  <ChildComponent @created="handleCreate" />
</template>
```

```
export default {
  methods: {
    handleCreate() {
      console.log('Child has been created.');
    }
  }
};
```

There is a lot more that can be done with events, and this only scratches the surface. I would highly recommend that you check out the official Vue docs to learn more about events. Definitely worth the read!

But events don't quite solve *all* of our problems.

Accessing a parent's scope from the child component

In many cases the problem you are trying to solve is accessing values from different scopes.

The parent component has one scope, and the child component another.

Often you want to access a value in the child component from the parent, or access a value in the parent component from the child.

Vue prevents us from doing this directly, which is a good thing.

It keeps our components more encapsulated and promotes their reusability. This makes your code cleaner and prevents lots of headaches in the long run.

But you may be tempted to try and pass functions as props to get around this.

Getting a value from the parent

If you want a child component to access a parent's method, it seems obvious to just pass the method straight down as a prop.

Then the child has access to it immediately, and can use it directly.

In the parent component we would do this:

```
<!-- Parent -->
<template>
  <ChildComponent :method="parentMethod" />
</template>
```

```
// Parent
export default {
  methods: {
    parentMethod() {
      // ...
    }
  }
}
```

And in our child component we would use that method:

```
// Child
export default {
  props: {
    method: { type: Function },
  },
  mounted() {
    // Use the parent function directly here
    this.method();
  }
}
```

What’s wrong with that?

Well, it’s not exactly *wrong*, but it’s much better to use events in this case.

Then, instead of the child component calling the function when it needs to, it will simply emit an event. Then the parent will receive that event, call the function, and then the props that are passed down to the child component will be updated.

This is a much better way of achieving the same effect.

Getting a value from the child

In other cases we may want to get a value from the child into the parent, and we're using functions for that.

For example, you might be doing this. The parent's function accepts the value from the child and does something with it:

```
<!-- Parent -->
<template>
  <ChildComponent :method="parentMethod" />
</template>
```

```
// Parent
export default {
  methods: {
    parentMethod(valueFromChild) {
      // Do something with the value
      console.log('From the child:', valueFromChild);
    }
  }
}
```

Where in the child component you pass the value in when calling it:

```
// Child
export default {
  props: {
    method: { type: Function },
  },
  data() {
    return { value: 'I am the child.' };
  },
  mounted() {
    // Pass a value to the parent through the function
    this.method(this.value);
  }
}
```

Again, this isn’t completely *wrong*.

It will work to do things this way.

It’s just that it isn’t the best way of doing things in Vue. Instead, events are much better suited to solving this problem.

We can achieve this exact same thing using events instead:

```
<!-- Parent -->
<template>
  <ChildComponent @send-message="handleSendMessage" />
</template>
```

```
// Parent
export default {
  methods: {
    handleSendMessage(event, value) {
      // Our event handler gets the event, as well as any
      // arguments the child passes to the event
      console.log('From the child:', value);
    }
  }
}
```

And in the child component we emit the event:

```
// Child
export default {
  props: {
    method: { type: Function },
  },
  data() {
    return { value: 'I am the child.' };
  },
  mounted() {
    // Instead of calling the method we emit an event
    this.$emit('send-message', this.value);
  }
}
```

Events are extremely useful in Vue, but they don’t solve 100% of our problems either.

There are times when we need to access a child’s scope from the parent in a different way.

For that, we have scoped slots!

Using scoped slots instead

Scoped slots are a more advanced topic, but they are also incredibly useful.

In fact, I would consider them to be one of the most powerful features that Vue offers.

They let you blur the lines between what is in the child's scope and what is in the parent's scope. But it's done in a very clean way that leaves your components as composable as ever.

If you want to learn more about how scoped slots work — and I really think that you should — check out Adam Wathan's great post on the subject.

How to Call a Vue Method on Page Load

As soon as the page loads, you want a certain function to be called.

Maybe you're fetching data, or you're initializing something.

How do we do this in Vue?

You'll want to use the `mounted` lifecycle hook so that you can run code as soon as your component is mounted to the DOM. From this lifecycle hook you can fetch data, manipulate the DOM, or do anything else you might need in order to initialize your component.

This article will explain a few things:

- What lifecycle hooks are
- How to use lifecycle hooks
- Why you should prefer using the `mounted` hook over the `created` hook

In order to call a function as soon as our Vue component has loaded, we'll first need to get familiar with Vue's lifecycle hooks.

Lifecycle Hooks

All Vue components have a series of stages — or *lifecycles* — that they go through.

As your app is run, your component will be:

- Created
- Mounted
- Updated
- Destroyed

Let's take a quick look at each of these 4 stages.

Lifecycles at a glance

First, the component is *created.*

Here, everything — including reactive `data`, computed props, and watchers — are setup and intialized.

Second, the component is *mounted* to the DOM.

This involves creating the actual DOM nodes and inserting your component into the page.

Third, your component is *updated* as reactive data changes. Vue will re-render your component many times, in order to keep everything on the page up-to-date.

Lastly, when the component is no longer needed, it is *destroyed*.

Any event listeners are cleaned up, DOM nodes are removed from the page, and any memory it was using is now released.

That’s not all.

Vue let’s us hook into these lifecycles. This lets us run code when the component is *created*, *mounted*, *updated*, or *destroyed*!

> *There is so much more to talk about when it comes to lifecycle methods. I would suggest you check out the docs as well as this awesome article about them to learn even more.*

Hooking into a lifecycle

So how do we use a lifecycle hook?

All we have to do is create a method on our component that uses the name of that lifecycle.

If we wanted to call a method right when the component is *created*, this is how we would do that:

```
export default {
  created() {
    console.log('Component has been created!');
  }
};
```

Similarly, hooking into the *destroyed* lifecycle works like this:

```
export default {
  destroyed() {
    console.log('Component has been destroyed!');
  }
};
```

But we're here to figure out **how to call a function as soon as our page loads.**

Both `created` and `mounted` hooks seem like they would work.

Which is the best one to use?

Using the Mounted Hook

In nearly all cases, the `mounted` hook is the right one for this job.

In fact, I always use this one. I only switch to a different hook if for some reason the `mounted` hook doesn't work for what I am trying to do.

The reason is this.

In the `mounted` hook, everything about the component has been initialized properly.

Props have been initialized, reactive `data` is going, computed props have been setup, and so have your watchers.

Most importantly though, Vue has setup everything in the DOM.

This means that you can safely do whatever you need to do in this lifecycle hook.

Using the Created Hook

There can often be some confusion around the differences between the `created` and `mounted` hooks.

As we saw earlier, the `created` hook is run before the `mounted` hook is run.

And in the `created` hook, nearly everything in the component has been setup. The only thing missing is the DOM.

So what is the `created` hook good for?

Well, since we don’t have access to our DOM element, we should use `created` for anything that doesn’t need the DOM element.

Oftentimes it is used to fetch data:

```
export default {
  data() {
    cars: {},
  },
  created() {
    fetch('/cars').then(cars => {
      this.cars = cars;
    });
  }
};
```

The advantage of using `created` instead of `mounted` is that `created` will be called a little sooner. This means you’ll get your data just a tiny bit faster.

Mounting the Component

Earlier I said that the `created` hook doesn’t have access to the DOM.

Let me explain why that is.

Vue first *creates* the component, and then it *mounts* the component to the DOM. We can only access the DOM *after* the component has been mounted. This is done using the `mounted` hook.

You can prove this to yourself by putting the following into your component:

```
created() {
  console.log(this.$el);
},
mounted() {
  console.log(this.$el);
}
```

What was logged out?

You should have gotten `undefined`, followed by a DOM element.

The DOM element for our component is set to `this.$el`, but it doesn't exist yet in the `created` hook.

However, it *does* exist in our `mounted` hook, because Vue has already done the work of adding it to the DOM by the time it is called.

How to Implement a Mouseover or Hover in Vue

In CSS it's pretty easy to change things on `hover`. We just use the `:hover` psuedo-class:

```
.item {
  background: blue;
}

.item:hover {
  background: green;
}
```

In Vue it gets a little trickier, because we don't have this functionality built in.

We have to implement most of this ourselves.

But don't worry, it's not *that* much work.

In this short article you'll learn:

- How to implement a **hover effect** in Vue
- How to **show an element** on mouseover
- How to dynamically **update classes** with a mouseover
- How to do this even on **custom Vue components**

Listening to the right events

So, which events do we need to listen to?

We want to know when the mouse is hovering over the element. This can be figured out by keeping track of when the mouse *enters* the element, and when the mouse *leaves* the element.

To keep track of when the mouse leaves, we'll use the mouseleave event.

Detecting when the mouse enters can be done with the corresponding mouseenter event, but we won't be using that one.

The reason is that there can be significant performance problems when using `mouseenter` on deep DOM trees. This is because `mouseenter` fires a unique event to the entered element, as well as every single ancestor element.

What will we be using instead, you ask?!?

Instead, we will use the mouseover event.

The `mouseover` event works pretty much the same as `mouseenter`. The main difference being that `mouseover` bubbles like most other DOM events. Instead of creating a ton of unique events, there is only one — making it much faster!

Let's hook things up

To hook everything up we will use Vue events to listen for when the mouse enters and leaves, and update our state accordingly.

We will also be using the shorthand of `v-on`.

Instead of writing `v-on:event`, we can just write `@event`.

Here's how we hook it up:

```
<template>
  <div
    @mouseover="hover = true"
    @mouseleave="hover = false"
  />
</template>
```

```
export default {
  data() {
    return {
      hover: false,
    };
  }
}
```

Now the reactive property `hover` will always reflect if the mouse is hovering over the element or not!

Showing an element when hovering

If you wanted to show an element based on the hover state, you can pair this with a v-if directive:

```
<template>
  <div>
    <span
      @mouseover="hover = true"
      @mouseleave="hover = false"
    >
      Hover me to show the message!
    </span>
    <span v-if="hover">This is a secret message.</span>
  </div>
</template>
```

```
export default {
  data() {
    return {
      hover: false,
    };
  }
}
```

Whenever you put your mouse over `Hover me to show the message!`, the secret message will appear!

Toggling classes when hovering

You can also do a similar thing to toggle classes:

```
<template>
  <div>
    <span
      @mouseover="hover = true"
      @mouseleave="hover = false"
      :class="{ active: hover }"
    >
      Hover me to change the background!
    </span>
  </div>
</template>
```

```
export default {
  data() {
    return {
      hover: false,
    };
  }
}
```

```
.active {
  background: green;
}
```

Although that works, it's not the best solution.

For something like this it's better to just use CSS:

```
<template>
  <span>
    Hover me to change the background!
  </span>
</template>
```

```
span:hover {
  background: green;
}
```

As you can see, even though we can do it using Vue, we don't need it to achieve this effect.

Hovering over a Vue component

If you have a Vue component that you'd like to implement this behaviour with, you'll have to make a slight modification.

You can't listen to the `mouseover` and `mouseleave` events like we were doing before.

If your Vue component doesn't emit those events, then we can't listen to them.

Instead, we can add the `.native` event modifier to listen to DOM events directly on our custom Vue component:

```
<template>
  <my-custom-component
    @mouseover.native="hover = true"
    @mouseleave.native="hover = false"
  />
</template>
```

```
export default {
  data() {
    return {
      hover: false,
    };
  }
}
```

Using `.native`, we listen for native DOM events instead of the ones that are emitted from the Vue component.

If this part is a little confusing, check out the docs. They do a really great job of explaining how this works.

Conclusion

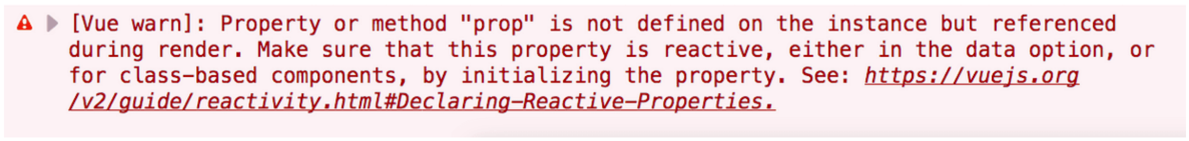
There you have it!

Achieving a mouseover effect in Vue JS requires only a little bit of code.

Now you can go and do all sorts of things when someone hovers over your Vue app.

Warn: Property or Method is Not Defined

Chances are if you've been developing with Vue for any amount of time, you've gotten this error:



```
[Vue warn]: Property or method "prop" is not defined on the instance but referenced
during render. Make sure that this property is reactive, either in the data option, or
for class-based components, by initializing the property. See: https://vuejs.org
/v2/guide/reactivity.html#Declaring-Reactive-Properties.
```

Most of the time this error is because you misspelled a variable name somewhere.

But there are other causes as well.

I've gotten this one so many times, because it's a fairly easy mistake to make. Luckily, **fixing it is pretty easy** too, and I'm no longer stumped by it like I was when I first encountered it.

What does the warning mean?

The full message is this:

> *[Vue warn]: Property or method "prop" is not defined on the instance but referenced during render. Make sure that this property is reactive, either in the data option, or for class-based components, by initializing the property. See: https://vuejs.org/v2/guide/reactivity.html#Declaring-Reactive-Properties.*

This gist of the error is this.

Vue is trying to render your component, and your component wants to use the property `prop`, but Vue can't find `prop` anywhere.

You don't need to worry about the reactive properties part of the error for now. As long as you define things in `data` you'll be okay (but it is good to understand how reactivity works).

So, let's take a look at the 2 main problems that can cause this warning!

1. You misspelled a variable name

This is the one that always gets me.

I was typing too fast, or wasn't paying enough attention. I misspelled a variable name, and now *Vue is complaining* because it doesn't know what I'm trying to do.

It's a pretty easy mistake to make:

```
<template>
  <div>
    {{ messag }}
  </div>
</template>
```

```
export default {
  data() {
    return {
      message: "Hello, world!"
    };
  }
};
```

Whenever I see this warning I make sure to closely inspect my code and ensure I haven't made any weird typos.

But if there are no typos, the problem lies somewhere else!

2. The value is defined on a different component

This is another common mistake that is easy to make.

Components are scoped, meaning that something defined in one component isn't available in another. You have to use props and events to move things between components.

But just like making a typo, it's pretty easy to forget that a method or property is on one component, **when it's actually on a different one.**

How does this happen?

Well, sometimes you'll want to define multiple components in one file. It might look something like this:

```
<!-- Template for the Page component -->
<template>
  <ul>
    <link-item url="google.com" text="Google" />
    <link-item url="yahoo.com" text="Yahoo" />
    <link-item url="facebook.com" text="Facebook" />
  </ul>
</template>
```

```
// Clean up some code by using another component
const LinkItem = {
  props: ['url', 'text'],
  template: `
    <li>
      <a
        :href="url"
        target="_blank"
      >
        {{ text }}
      </a>
    </li>
  `
};

// Define the Page component
export default {
  name: 'Page',
  components: { LinkItem },
};
```

Instead of rewriting the `<li>` tag each time, we just encapsulated it inside of the `LinkItem` component.

But let's say we have a method on our `Page` component that forces us to always use HTTPS instead of HTTP:

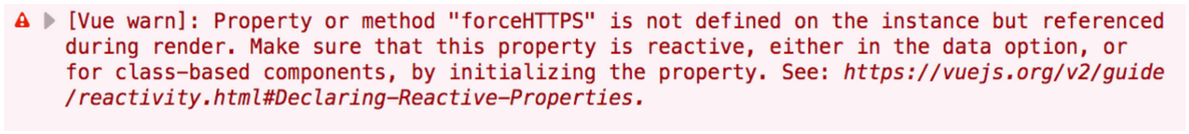
```
methods: {
  forceHTTPS(url) {
    // ...
  }
}
```

What `forceHTTPS` actually does is unimportant here.

But let's use it on our URLs so that we can make sure our app only links to safe website. We'll update `LinkItem` to use this method:

```
const LinkItem = {
  props: ['url', 'text'],
  template: `
    <li>
      <a
-       :href="url"
+       :href="forceHTTPS(url)"
        target="_blank"
      >
        {{ text }}
      </a>
    </li>
  `
};
```

We save it, our page refreshes, and...



```
[Vue warn]: Property or method "forceHTTPS" is not defined on the instance but referenced
during render. Make sure that this property is reactive, either in the data option, or
for class-based components, by initializing the property. See: https://vuejs.org/v2/guide
/reactivity.html#Declaring-Reactive-Properties.
```

Whoops.

It turns out that we forgot that `forceHTTPS` isn’t defined on the `LinkItem` component, but instead it’s defined on the `Page` component!

Because the two components were in the same file, **it was easy for us to get mixed up with what was and wasn’t in scope**. This is one reason why it can be a good idea to separate things out into their own files, even if it’s only a few lines long.

Let’s fix that then:

```
const LinkItem = {
  props: ['url', 'text'],
+ methods: {
+   forceHTTPS(url) {
+     // Do some stuff...
+   }
+ },
  template: `
    <li>
      <a
        :href="forceHTTPS(url)"
        target="_blank"
      >
        {{ text }}
      </a>
    </li>
  `
};
```

Now things should be dandy.

As you can see, it’s **pretty easy to mess this up** and forget that a value is on a different component. Thankfully Vue has this handy warning!

Conclusion

If you have this warning all of the time, don’t worry about it!

It’s really common, and even though I have a lot of experience with Vue, **I still get this one all of the time.**

To recap, the two main causes of this are:

1. You have a typo somewhere when naming a property or method
2. The property or method exists on a different component

If you’re still having trouble with this warning, the problem may lie elsewhere. Ping me on Twitter and I’d love to help you out with it!