HOW TO

Solve Common Problems

IN VUE

Hi there!

I wanted to thank you for downloading my book. I've put a lot of time and energy into it, and I hope you get a lot out of it.

Unlike most books, there is no progression from beginning to end.

Instead, each chapter stands on it's own. A specific solution to a specific problem.

This means that this book isn't meant to be read from cover to cover.

You can dip in and out however you want.

So find a chapter that sounds interesting, and check it out. Then read another (but only if you want).

It doesn't matter if you read from back to front, or if you never read the whole thing.

What's most important is that afterwards, you're able to write Vue apps faster, and with much less frustration.

Enjoy!

- Michael Thiessen

Table of Contents

Reactivity

Errors

How To

Part 1

Reactivity

Chapter 1

Force Vue to Re-render a Component

Sometimes Vue's reactivity system isn't enough, and you just need to re-render a component.

Or maybe you just want to blow away the current DOM and start over.

So how do you get Vue to re-render a component **the right way?**

The best way to force Vue to re-render a component is to set a `:key` on the component. When you need the component to be re-rendered, you just change the value of the key and Vue will re-render the component.

It's a pretty simple solution, right?

You'll be happy to know that there are lots of others ways to do it:

- The horrible way: **reloading** the entire page
- The terrible way: using the **v-if hack**
- The better way: using Vue's built-in **forceUpdate** method
- The best way: **key-changing** on your component

Except here is where I'm going to ruin it for you.

If you need to force a reload or force an update, there's probably a better way.

It's likely that you're misunderstanding one of the following tricky things:

1. Vue's **reactivity**
2. Computed props
3. Watched props (sometimes)
4. Not using a `:key` attribute with `v-for`

Now, there **are** valid use cases for forcing an update. Most of these will be **solved using the key-changing technique** that's at the bottom of this article.

Horrible way: reload the entire page

This is one is equivalent to **restarting your computer every time you want to close an app**.

I **guess** it would work some of the time, but it's a pretty bad solution.

There isn't really much more to say about this. Don't do it.

Let's look for a better way.

Terrible way: the v-if hack

Vue comes with the `v-if` directive that will only render the component when it's true. If it's false, the component will not exist at all in the DOM.

Here's how we set it up for the `v-if` hack to work.

In your `template` you'll add the `v-if` directive:

```
<template>
  <my-component v-if="renderComponent" />
</template>
```

In your `script` you'll add in this method that uses `nextTick`:

```
<script>
  export default {
    data() {
      return {
        renderComponent: true,
      };
    },
    methods: {
      forceRerender() {
        // Remove my-component from the DOM
        this.renderComponent = false;

        this.$nextTick(() => {
          // Add the component back in
          this.renderComponent = true;
        });
      }
    }
  };
</script>
```

This is what's going on here:

1. Initially `renderComponent` is set to `true`, so `my-component` is rendered
2. When we call `forceRerender` we immediately set `renderComponent` to `false`
3. We stop rendering `my-component` because the `v-if` directive now evaluates to `false`
4. On the next tick `renderComponent` is set back to `true`

5. Now the `v-if` directive evaluates to `true`, so we start rendering `my-component` again

There are two pieces that are important in understanding how this works.

First, we have to **wait until the next tick** or we won't see any changes.

In Vue, a tick is a single DOM update cycle. Vue will collect all updates made in the same tick, and at the end of a tick it will update what is rendered into the DOM based on these updates. If we don't wait until the next tick, our updates to `renderComponent` will just cancel themselves out, and nothing will change.

Second, **Vue will create an entirely new component** when we render the second time. Vue will destroy the first one and create a new one. This means that our new `my-component` will go through all of its lifecycles as normal — `created`, `mounted`, and so on.

On a side note, you can use `nextTick` with promises if you prefer that:

```
forceRerender() {
  // Remove my-component from the DOM
  this.renderComponent = false;

  // If you like promises better you can
  // also use nextTick this way
  this.$nextTick().then(() => {
    // Add the component back in
    this.renderComponent = true;
  });
}
```

Still, this isn't a great solution. I call it a hack because we're hacking around what Vue wants us to do.

So instead, **let's do what Vue wants us to do**!

Better way: You can use forceUpdate

This is one of the two best ways to solve this problem, both of which are **officially supported by Vue**.

Normally, Vue will react to changes in dependencies by updating the view. However, when you call `forceUpdate`, you can force that update to occur, even if none of the dependencies has actually changed.

Here is where most people make the **biggest mistakes** with this method.

If Vue automatically updates when things change, **why should we need to force an update**?

The reason is that sometimes Vue's reactivity system can be confusing, and we **think** that Vue will react to changes to a certain property or variable, but it doesn't actually. There are also certain cases where Vue's reactivity system won't detect any changes at all.

So just like the last methods, if you need this to re-render your component, **there's probably a better way**.

There are two different ways that you can call `forceUpdate`, on the component instance itself as well as globally:

```
// Globally
import Vue from 'vue';
Vue.forceUpdate();

// Using the component instance
export default {
  methods: {
    methodThatForcesUpdate() {
      // ...
      this.$forceUpdate();  // Notice we have to use a $ here
      // ...
    }
  }
}
```

Important: This will not update any computed properties you have. Calling `forceUpdate` will only force the view to re-render.

The best way: key-changing

There are many cases where you will have a legitimate need to re-render a component.

To do this the proper way, we will supply a `key` attribute so Vue knows that a specific component is tied to a specific piece of data. If the key stays the same, it won't change the component, but if the key changes, Vue knows that it should **get rid of the old component and create a new one**.

Exactly what we need!

But first we'll need to take a **very short** detour to understand why we use `key` in Vue.

Why do we need to use key in Vue?

Once you understand this, it's a pretty small step to understanding how to force re-renders the proper way.

Let's say you're rendering a list of components that has one or more of the following:

- It's own **local state**
- Some sort of **initialization process**, typically in `created` or `mounted` hooks
- Non-reactive **DOM manipulation**, through jQuery or vanilla APIs

If you sort that list, or update it in any other way, you'll need to re-render parts of the list. But you won't want to re-render **everything** in the list, just the things that have changed.

To help Vue keep track of what has changed and what hasn't, we supply a `key` attribute. **Using the index of an array is not helpful here**, since the index is not tied to specific objects in our list.

Here is an example list that we have:

```
const people = [
  { name: 'Evan', age: 34 },
  { name: 'Sarah', age: 98 },
  { name: 'James', age: 45 },
];
```

If we render it out using indexes we will get this:

```
<ul>
  <li v-for="(person, index) in people" :key="index">
    {{ person.name }} - {{ index }}
  </li>
</ul>

// Outputs
Evan - 0
Sarah - 1
James - 2
```

If we remove Sarah, we will get:

```
Evan - 0
James - 1
```

The index associated with James is changed, even though James is still James. James will be re-rendered, even if we don't want him to be.

So here **we want to use some sort of unique id**, however we end up generating it.

```
const people = [
  { id: 'this-is-an-id', name: 'Evan', age: 34 },
  { id: 'unique-id', name: 'Sarah', age: 98 },
  { id: 'another-unique-id', name: 'James', age: 45 },
];

<ul>
  <li v-for="person in people" :key="person.id">
    {{ person.name }} - {{ person.id }}
  </li>
</ul>
```

Before when we removed Sarah from our list, Vue deleted the components for Sarah and James, and then created a new component for James. Now, Vue knows that it can keep the two components for Evan and James, and all it has to do is delete Sarah's.

If we add a person to the list, it also knows that it can keep all of the existing components, and it only has to create a single new component and insert it into the correct place. This is really useful, and helps us a lot when we have **more complex components that have their own state, have initialization logic, or do any sort of DOM manipulation**.

Maybe that detour wasn't so short. But it was necessary to explain how keys in Vue work.

Anyways, let's get on with the best method of forcing re-renders!

Key-changing to force re-renders of a component

Finally, here is the **very best way** (in my opinion) to force Vue to re-render a component.

You take this strategy of assigning keys to children, but whenever you want to re-render a component, you just update the key.

Here is a very basic way of doing it:

```
<template>
  <component-to-re-render :key="componentKey" />
</template>
```

```
export default {
  data() {
    return {
      componentKey: 0,
    };
  },
  methods: {
    forceRerender() {
      this.componentKey += 1;
    }
  }
}
```

Every time that `forceRerender` is called, our prop `componentKey` will change. When this happens, Vue will know that it has to destroy the component and create a new one.

What you get is a child component that will re-initialize itself and "reset" it's state.

A simple and elegant way to solve our problem!

Just remember, if you find yourself needing to force Vue to re-render a component, maybe you aren't doing something the best way.

If, however, you do need to re-render something, choose the **key-changing** method over anything else.

Chapter 2

Props vs 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.

So what's the difference between props and data?

`Data` is the private memory of each component where you can store any variables you need. Props are how you pass this data from a parent component down to a child component.

In this article you'll learn:

- What props are, and why **this data only flows down**, not up
- What the `data` option is used for
- What **reactivity** is
- How to avoid **naming collisions** between props and `data`
- How to use **props and data together** for fun and profit

What are props?

In Vue, props (or properties), are the way that we pass data from a parent component down to it's child components.

When we build our applications out of components, we end up building a data structure called a tree. Similar to a family tree, you have:

- parents
- children
- ancestors
- and descendants

Data flows down this tree from the root component, the one at the very top. Sort of like how genetics are passed down from one generation to the next, **parent components pass props down to their children**.

In Vue we add props to components in the `<template>` section of our code:

```
<template>
  <my-component cool-prop="hello world"></my-component>
</template>
```

In this example, we are passing the prop `cool-prop` a value of `"hello world"`. We will be able to access this value from inside of `my-component`.

However, when we access props from inside of a component, we don't own them, **so we can't change them** (just like you can't change the genes your parents gave you).

> **Note: While it's possible to change properties in a component, it's a really bad idea. You end up changing the value that the parent is using as well, which can cause lots of confusion.**

But if we can't change variables, we're kind of stuck.

This is where `data` comes in!

What is data?

Data is the **memory** of each component. This is where you would store data (hence the name), and any other variables you want to track.

If we were building a counter app, we would need to keep track of the count, so we would add a `count` to our `data`:

```
<template>
  <div>
    {{ count }}
    <button @click="increment">+</button>
    <button @click="decrement">-</button>
  </div>
</template>
```

```
export default {
  name: 'Counter',
  data() {
    return {
      // Initialized to zero to begin
      count: 0,
    }
  },
  methods: {
    increment() {
      this.count += 1;
    },
    decrement() {
      this.count -= 1;
    }
  }
}
```

This data is private, and **only for the component itself** to use. Other components do not have access to it.

> **Note: Again, it is possible for other components to access this data, but for the same reasons, it's a really bad idea to do this!**

If you need to pass data to a component, you can use props to pass data down the tree (to child components), or events to pass data up the tree (to parent components).

Props and data are both reactive

With Vue you don't need to think all that much about when the component will update itself and render new changes to the screen.

This is because Vue is reactive.

Instead of calling `setState` every time you want to change something, you just change the thing! As long as you're updating a **reactive property** (props, computed props, and anything in `data`), Vue knows to watch for when it changes.

Going back to our counter app, let's take a closer look at our methods:

```
methods: {
  increment() {
    this.count += 1;
  },
  decrement() {
    this.count -= 1;
  }
}
```

All we have to do is update `count`, and Vue detects this change. It then re-renders our app with the new value!

Vue's reactivity system has a lot more nuance to it, and I believe it's really important to understand it well if you want to be highly productive with Vue. Here are some more things to learn about Vue's reactivity system if you want to dive deeper.

Avoiding naming collisions

There is another great thing that Vue does that makes developing **just a little bit nicer**.

Let's define some props and data on a component:

```
export default {
  props: ['propA', 'propB'],
  data() {
    return {
      dataA: 'hello',
      dataB: 'world',
    };
  },
};
```

If we wanted to access them inside of a method, we don't have to do `this.props.propA` or `this.data.dataA`. Vue let's us omit `props` and `data` completely, leaving us with cleaner code.

We can access them using `this.propA` or `this.dataA`:

```
methods: {
  coolMethod() {
    // Access a prop
    console.log(this.propA);

    // Access our data
    console.log(this.dataA);
  }
}
```

Because of this, if we accidentally use the same name in both our `props` and our `data`, we can run into issues.

Vue will give you a warning if this happens, because it doesn't know which one you wanted to access!

```
export default {
  props: ['secret'],
  data() {
    return {
      secret: '1234',
    };
  },
  methods: {
    printSecret() {
      // Which one do we want?
      console.log(this.secret);
    }
  }
};
```

The real magic of using Vue happens when you start using props and `data` together.

Using props and data together

Now that we've seen how props and data are different, let's see why **we need both of them**, by building a basic app.

Let's say we are building a social network and we're working on the profile page. We've built out a few things already, but now we have to add the contact info of the user.

We'll display this info using a component called `ContactInfo`:

```
// ContactInfo
<template>
  <div class="container">
    <div class="row">
      Email: {{ emailAddress }}
      Twitter: {{ twitterHandle }}
      Instagram: {{ instagram }}
    </div>
  </div>
</template>
```

```
export default {
  name: 'ContactInfo',
  props: ['emailAddress', 'twitterHandle', 'instagram'],
};
```

The `ContactInfo` component takes the props `emailAddress`, `twitterHandle`, and `instagram`, and displays them on the page.

Our profile page component, `ProfilePage`, looks like this:

```
// ProfilePage
<template>
  <div class="profile-page">
    <div class="avatar">
      <img src="user.profilePicture" />
      {{ user.name }}
    </div>
  </div>
</template>
```

```
export default {
  name: 'ProfilePage',
  data() {
    return {
      // In a real app we would get this data from a server
      user: {
        name: 'John Smith',
        profilePicture: './profile-pic.jpg',
        emailAddress: 'john@smith.com',
        twitterHandle: 'johnsmith',
        instagram: 'johnsmith345',
      },
    }
  }
};
```

Our `ProfilePage` component currently displays the users profile picture along with their name. It also has the user data object.

How do we get that data from the parent component (`ProfilePage`) down into our child component (`ContactInfo`)?

We have to pass down this data using props.

First we need to import our `ContactInfo` component into the `ProfilePage` component:

```
// Import the component
import ContactInfo from './ContactInfo.vue';

export default {
  name: 'ProfilePage',

  // Add it as a dependency
  components: {
    ContactInfo,
  },

  data() {
    return {
      user: {
        name: 'John Smith',
        profilePicture: './profile-pic.jpg',
        emailAddress: 'john@smith.com',
        twitterHandle: 'johnsmith',
        instagram: 'johnsmith345',
      },
    }
  }
};
```

Second, we have to add in the component to our `<template>` section:

```
// ProfilePage
<template>
  <div class="profile-page">
    <div class="avatar">
      <img src="user.profilePicture" />
      {{ user.name }}
    </div>

    <!-- Add component in with props -->
    <contact-info
      :email-address="emailAddress"
      :twitter-handle="twitterHandle"
      :instagram="instagram"
    />

  </div>
</template>
```

Now all the user data that `ContactInfo` needs will flow down the component tree and into `ContactInfo` from the `ProfilePage`!

The reason we keep the data in `ProfilePage` and not `ContactInfo` is that other parts of the profile page need access to the user object.

Since **data only flows down**, this means we have to put our data high enough in the component tree so that it can flow down to all of the places it needs to go.

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

Chapter 3

How to Watch Nested Data

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 chapter, 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 chapter

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.

Chapter 4

Computed Props vs Watchers

Computed properties and watchers are two of the most fundamental concepts in Vue.

But I often see people mixing them up — using one when they should be using the other.

They have a lot of similarities, so it can be hard knowing which one is best for what you're trying to accomplish.

In this chapter you'll learn:

- What a watcher is, and what a computed prop is
- Common use cases for both
- The differences and similarities between them
- How to know which to use

What is a watcher?

When building components in Vue, we often need to respond to changes in our props.

A watcher — or watched prop — let's us **track a property** on our component and **run a function whenever it changes**.

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!');
    }
  }
}
```

You can put a watcher on any reactive property. This includes computed props, **props**, as well as data that is specified inside of `data()` on your Vue component.

They're really useful for **creating side effects** — things that don't update your application's state immediately.

If you need to fetch data or do some other **ansychronous action**, watched props are really good for that. Or maybe you need to interact with an imperative browser API, such as `localstorage`.

Because watchers aren't expected to be pure functions, we can do all sorts of things like this with them!

Watched props are really powerful, but many times I accidentally use a watcher when all that I needed was a computed property.

What is a computed prop?

Computed props let us compose new data from other data.

Let's say we have a component that takes a person's `firstName` and `lastName`. We can create `fullName` as a computed prop:

```
export default {
  name: 'FullName',
  props: ['firstName', 'lastName'],
  computed: {
    fullName() {
      return this.firstName + ' ' + this.lastName;
    }
  }
}
```

Because computed props are **reactive**, whenever `firstName` or `lastName` are changed, `fullName` will be recomputed and will always be up to date.

Already we can see an advantage of computed props over watchers.

Composing this data together **can** be done with watchers, but it's much cleaner and **more declarative** using computed props:

```
export default {
  name: 'FullName',
  props: ['firstName', 'lastName'],
  data() {
    return {
      fullName: this.firstName + ' ' + this.lastName,
    };
  },
  watched: {
    firstName() {
      this.fullName = this.firstName + ' ' + this.lastName;
    },
    lastName() {
      this.fullName = this.firstName + ' ' + this.lastName;
    }
  }
}
```

This is a fairly simple example — computed props can depend on as many other properties as you need them to. You can even watch other computed props, as well as reactive data in `data()`!

Computed properties are required to be **pure functions**.

This means that they return a new value, but aren't allowed to change anything, and they must be synchronous.

Once we've created our computed prop, we can access it like we would any other prop. This is because computed props are reactive properties, along with regular props and data.

Now that we've covered how to use watchers and computed props, **where do we use them?**

Common use cases for a watcher

The most common use case for a watched prop is in **creating side effects.**

A **side effect** is anything that affects state outside of the component, or affects state in an asynchronous way.

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**.

We'll look at an example of fetching data.

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 us the new data.

You can also use watchers to recompute data in certain cirumstances.

While computed props are generally the better way of doing this (as we'll see in a moment), there are cases where it doesn't make sense to use a computed prop.

> **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.**

Common use cases for computed props

The main use case for computed props, as discussed before, is in composing data from other data.

This use case is pretty straightforward, but extremely useful. In my Vue code I use computed props far more than I use watchers, and almost always they are used to compose new data together.

I'll say this again because it's so important.

Using computed props to compose new data is one of the most useful patterns to learn when building Vue applications.

I use them literally everywhere.

Another one is allowing us **easier access to nested data**.

If we had some deeply nested data in our component that we needed access to, like `data.nested.really.deeply.importantValue`, we can simplify this a whole lot:

```
computed() {
  importantValue() {
    return this.data.nested.really.deeply.importantValue;
  },
}
```

And now we only need to write `this.importantValue` to get ahold of it!

Differences and Similarities

Let's see how they are different:

- Computed props are **more declarative** than watched properties
- Computed props should be **pure**: return a value, synchronous, and have no side-effects

- Computed props create new **reactive properties**, watched props only call functions
- Computed props can react to changes in **multiple props**, whereas watched props can only watch one at a time
- Computed props are **cached**, so they only recalculate when things change. Watched props are executed every time
- Computed props are **evaluated lazily**, meaning they are only executed when they are needed to be used. Watched props are executed whenever a prop changes

But they also have some similarities.

They aren't exactly twins, but they both:

- **react to changes** in properties
- can be used to **compute new data**

But how do you know **which one to use** in a given situation?

Which one do you use?

Watched properties 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.

Most of the time you'll want a computed prop, so **try to use that first**. If it doesn't work, or results in weird code that makes you feel like taking a shower, then you should switch to using a watched prop.

Learn more

If you want to learn more about this, Sarah Drasner has a great article comparing computed props, watchers, and methods.

You can also check out the official documentation, which addresses this issue specifically.

Part 2

Errors

Chapter 5

How to Fix "this is undefined"

You're happily coding along, loving how awesome Vue is, when it strikes.

Your VueJS app doesn't work, and you get an error that says:

```
this is undefined
```

Don't worry, you're not alone — I've run into this issue countless times, and I'll show you just how to solve it.

The likely cause of this is that you're mixing up your usage of regular functions and arrow functions. My guess is that you're using an arrow function. If you replace the arrow function with a regular function it will probably fix the issue for you.

But let's go a little further and try to understand **why this works**.

After all, knowledge is power, and if you know what caused your problem, you'll be able to avoid a lot of frustration and wasted time in the future.

There are also a few other places where you can get tripped up with this error:

- Fetching data using `fetch` or `axios`
- Using libraries like `lodash` or `underscore`

I'll cover these as well, and how to do them properly.

Understanding the two main types of functions

In Javascript we get two different kinds of functions. They operate in almost identical ways, except they differ in how they treat the variable `this`.

This causes a ton of confusion for new and old Javascript devs alike — but by the time we're through you won't be caught by this one anymore.

Regular functions

A regular function can be defined in a few different ways.

The first way is less common in Vue components, because it's a bit longer to write out:

```
methods: {
  regularFunction: function() {
    // Do some stuff
  }
}
```

The second way is the shorthand function, which is probably more familiar to you:

```
methods: {
  shorthandFunction() {
    // Do some stuff
  }
}
```

In a regular function like this one, `this` will refer to the "owner" of the function. Since we're defining it on the Vue component, `this` **refers to your Vue component**. I'll explain how this scoping works in more detail later on.

In most cases you should use a regular function with Vue, especially when creating:

- methods
- computed props
- watched props

While regular functions are usually what you need, arrow functions come in very handy as well.

Arrow functions

Arrow functions can be even shorter and quicker to write, and have gained lots of popularity recently because of this. However, they aren't too different when defining a method on an object like we are doing when writing Vue components.

This is what they look like on a Vue component:

```
methods: {
  arrowFunction: () => {
    // Do some stuff
  }
}
```

The real differences start to come in to play when dealing with how `this` works.

In an arrow function, `this` **does not** refer to the owner of the function.

An arrow function uses what is called **lexical scoping**. We'll get into this more in a bit, but it basically means that the arrow function takes `this` from it's context.

If you try to access `this` from inside of an arrow function that's on a Vue component, you'll get an error because `this` doesn't exist!

```
data() {
  return {
    text: 'This is a message',
  };
},
methods: {
  arrowFunction: () => {
    console.log(this.text);  // ERROR! this is undefined
  }
}
```

So in short, try to avoid using arrow functions on Vue components. It will save you a lot of headaches and confusion.

There **are** times when it's nice to use a short arrow function. But this only works if you aren't referencing `this`:

```
computed: {
  location: () => window.location,
}
```

Now that we know the two main types of functions, how do we use them in the correct way?

Anonymous Functions

Anonymous functions are great for when you just need to create a quick function and don't need to call it from anywhere else. They're called **anonymous** because they aren't given a name, and aren't tied to a variable.

Here are some scenarios where you'd use an anonymous function:

- Fetching data using `fetch` or `axios`
- Functional methods like `filter`, `map`, and `reduce`
- Anywhere else **inside of Vue methods**

I'll show you what I mean:

```
// Fetching data
fetch('/getSomeData').then((data) => {
  this.data = data;
});

// Functional methods
const array = [1, 2, 3, 4, 5];
const filtered = array.filter(number => number > 3);
const mapped = array.map(number => number * 2);
const reduced = array.reduce((prev, next) => prev + next);
```

As you can see from the examples, most of the time when people create anonymous functions they use arrow functions. We typically use arrow

functions for several reasons:

- **Shorter and more condensed syntax**
- Improved readability
- `this` is taken from the **surrounding context**

Arrow functions also work great as anonymous functions inside of Vue methods.

But wait, didn't we just figure out that **arrow functions don't work** when we try to access `this`?

Ah, but here is where the distinction is.

When we use arrow functions **inside a regular or shorthand function**, the regular function sets `this` to be our Vue component, and the arrow function uses that `this` (say that 5 times fast!).

Here's an example:

```
data() {
  return {
    match: 'This is a message',
  };
},
computed: {
  filteredMessages(messages) {
    console.log(this); // Our Vue component

    const filteredMessages = messages.filter(
      // References our Vue Component
      (message) => message.includes(this.match)
    );

    return filteredMessages;
  }
}
```

Our filter can access `this.match` because the arrow function uses the same context that the method `filteredMessages` uses. This method, **because it is a regular function** (and not an arrow function), sets it's own context to be the Vue instance.

Let's expand further on how you would this to fetch data using `axios` or `fetch`.

Using the right function when fetching data

If you're fetching async data using `fetch` or `axios`, you're also using promises. Promises **love** anonymous arrow functions, and they also make working with `this` a lot easier.

If you're fetching some data and want to set it on your component, this is how you'd do that properly:

```
export default {
  data() {
    return {
      dataFromServer: undefined,
    };
  },
  methods: {
    fetchData() {
      fetch('/dataEndpoint')
        .then(data => {
          this.dataFromServer = data;
        })
        .catch(err => console.error(err));
    }
  }
};
```

Notice how we're using a regular function as the method on the Vue component, and then using anonymous arrow functions inside of the promise:

```
.then(data => {
  this.dataFromServer = data;
})
```

Inside of the scope for `fetchData()`, we have that `this` is set to our Vue component because it is a regular function. Since arrow functions use the outer scope as their own scope, the arrow functions also set `this` to be our Vue component.

This allows us to access our Vue component through `this` and update `dataFromServer`.

But what if you need to pass functions to a helper library, like `lodash` or `underscore`?

Using with Lodash or Underscore

Let's say that you have a method on your Vue component that **you want to debounce using Lodash or Underscore**. How do you prevent those pesky `this is undefined` errors here?

If you come from the React world, you've probably seen something similar to this.

Here is how we would do it in Vue:

```
created() {
  this.methodToDebounce = _.debounce(this.methodToDebounce, 500);
},
methods: {
  methodToDebounce() {
    // Do some things here
  }
}
```

That's it!

All we're doing is taking our function, wrapping it in the debounce function, and returning a new one that has the debouncing built in. Now when we call `this.methodToDebounce()` on our Vue component we will be calling the debounced version!

What is lexical scoping?

So I promised that I would explain this more clearly, so here it is.

As I mentioned before, the main reason that there is a difference between regular functions and arrow functions has to do with **lexical scoping**.

Let's break down what it means. We'll work in reverse order.

First, **a scope is any area of the program where a variable exists**. In Javascript, the `window` variable has global scope — it's available everywhere. Most variables though are limited to the function they are defined in, the class they are a part of, or limited to a module.

Second, **the word "lexical" just means that the scope is determined by how you write the code**. Some programming languages will determine what is in scope only once the program is running. This can get really confusing, so most languages just stick with lexical scoping.

Arrow functions use lexical scoping, but regular and shorthand functions do not.

The trickiest part here is how lexical scoping affects `this` in your functions. For arrow functions, `this` is bound to the same `this` of the outer scope. Regular functions have some weirdness to how they bind `this`, which is why arrow functions were introduced, and why most people try and use arrow functions as much as possible.

Examples of how scope works in functions

Here are some examples to illustrate how scope works differently between the two function types:

```
// This variable is in the window's scope
window.value = 'Bound to the window';

const object = {
  // This variable is in the object's scope
  value: 'Bound to the object',
  arrowFunction: () => {
    // The arrow function uses the window's scope for `this`
    console.log(this.value); // 'Bound to the window'
  },
  regularFunction() {
    // The regular function uses the object's scope for `this`
    console.log(this.value);  // 'Bound to the object'
  }
};
```

Now you know a bit about how scope is bound to functions in Javascript!

But there is a way to override this default behaviour and do it all yourself...

Binding scope to a function

Did you know that you can actually **override how `this` is bound** and provide your own `this` to a function?

You need to use the bind method on the function:

```
const boundFunction = unboundFunction.bind(this);
```

This gives you much greater flexibility in writing Vue components, and let's you reuse your methods more easily.

It's a bit more advanced in it's usage, so you should try to avoid using it too often.

Chapter 6

Avoid Mutating Props Directly

So you've gotten this confusing error:

> **Avoid mutating a prop directly since the value will be overwritten whenever the parent component re-renders**

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

By the end of this chapter 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`

What causes this?

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 chapter, 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.

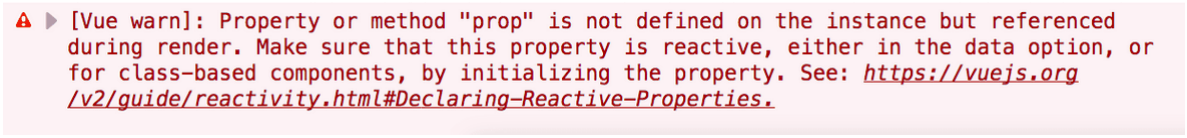
If this still didn't fix your problem, or you have questions about this chapter, please reach out to me on Twitter.

I'm always available to help!

Chapter 7

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:



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!

Part 3

How To

Chapter 8

Implement a Mouseover or Hover

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 chapter 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.

Chapter 9

Pass a Function as a Prop

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 chapter 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.

Chapter 10

Dynamically Add Classnames

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 chapter:

- 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',
    }
  }
};
```

```
.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!

Chapter 11

Call a 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 chapter 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.