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 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.