The correct way to 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.