# 5 Extremely Easy ways to drastically improve your VueJS app's speed

dev.to/veebuv/5-extremely-easy-ways-to-drastically-improve-your-vuejs-app-s-speed-5k0

## The number 1

What we have here is a "functional" template, in that it has no declared state, and only deals with props. This can be easily created into a Vue based functional component with the use of the render method https://vuejs.org/v2/guide/render-function.html

```
<template>
  <div>
    Hi {{ name }}
  </div>
</template>

<script>
export default {
  props: {
    name: {
      type: String,
      default: '',
    },
  },
};
</script>
```

If you read into it, you see props being passed called `functional: true`

So an easy fix to this solution is below

```
<template functional>
  <div>
    Hi {{ name }}
  </div>
</template>

<script>
export default {
  props: {
    name: {
      type: String,
      default: '',
    },
  },
};
</script>
```

As simple as that, you don't need to worry about changing your template syntax, you can stick to it and still enjoy the luxury of Vue syntax.

QUICK EDIT: Because it's a functional component, the context of this is non-existent, so to access props, you need to do props.name — Thanks to Dinesh Madhanlal for the mention

## Second Easy Tip

Using keep-alive for dynamically loaded components.

Sometimes we load components on the fly using `is` prop provided by Vue, and we might switch between components that are dynamically loaded. Inorder for us to maintain state and prevent re-loading of data whenever components are toggled, DOMless wrapper acts as a good solution to speed things up

```
<template>
  <component :is="currentTabComponent" />
</template>
```

```
<template>
  <keep-alive>
    <component :is="currentTabComponent" />
  </keep-alive>
</template>
```

## The Third Easy Tip

This one would be a little more obvious to most, given how Vue's vDOM system operates. The goal of the vDOM is to act as an intermediary update medium and track (very efficiently) isolated changes to the UI of the project and triggered isolated rerenders to those targeted components vs actually repainting the entire screen.

Often times, we might create a component that has a wrapper which re-renders a lot and the some other part of the same template has to do ALOT of work whenever the other part of the template re-renders, an easy fix is to simply componentise the file. Unless the child depends on the parent wrt to data, it should operate fine.

```
<template>
  <div :style="{ length: loaded/100 }">
    <div>{{ makeApiCall() }}</div>
  </div>
</template>
```

```
<template>
  <div :style="{ length: loaded/100 }">
    <ChildComponent />
  </div>
</template>
```

## The Fourth Easy Tip

Using anonymous functions in CTA events. Whenever an anonymous function is passed to the “onClick” of a button, Ive seen this pattern amongst devs who come from React because that’s one of the ways in React to pass custom data to a function, its better to not pass an anonymous function. Reason being such.

Take this example below

```
<template>
  <div>
    <div :style="{ length: loaded/100 }" />
    <button
      v-for="num in [1,2,3]"
      :key="num"
      @click="() => sendData(num)"
    >
      Click Me
    </button>
  </div>
</template>
```

Whats happening here is every single time the div grows in length (a simple progress bar) all the buttons will be re-rendered as well.
They technically shouldn’t be, because nothing in them is changing right ? No props update or no data update etc.

Thats the quick catch, JS interacts with anonymous functions in memory, i.e each time a re-render happens, a new instance of the anonymous function is created, and the diffing algo picks it up as a new prop and therefore re-renders the buttons even if it’s not needed.

Luckily Vue is so amazing, it’s smart enough to understand any self invoked function should not be called until the event it’s attached to is triggered, so even if its a IIF, Vue makes it a thunk, delaying the execution.

```
<template>
  <div>
    <div :style="{ length: loaded/100 }" />
    <button
      v-for="num in [1,2,3]"
      :key="num"
      @click="sendData(num)"
    >
      Click Me
    </button>
  </div>
</template>
```

Of if you want to be safer, always worth creating a closure that returns another function, therefore the wrapper function only ever has one instance and won't cause a re-render.

## The Magic 5th Easy Tip

This is also a simple one, there are grey area's for this and isn't a blanket solution. Use this method only for times where there's a lot of collateral on the page and toggling the display of a component happens rapidly,

Yes I'm talking about the use of v-if or v-show. There's a massive difference between the two. V-if = false never renders the component its a directive of. So if this component is being toggled multiple times in a short span, it will affect performance, so using v-show in such situations works out really well.

However, the catch is this, in a situation you add v-show to a component and that component needs to do a heavy operation on the `first time` its rendered, then that operation will be executed regardless of v-show being true or false, is worth delaying it by using a v-if until that component is actually needed. Remember v-show only sets the CSS display value of a component to display: none, the component is still "rendered".

However even if this component has a heavy initial workload, if its continuously toggled, and that method needs to be executed each time, its better to do a v-show. It all comes down to the user requirements.