RETHINKING

Reusability

IN VUE

Alex Vipond

Contents

Chapter 2: Refactoring compound components

Chapter 3: Authoring the function ref pattern

- Understand the function ref core concepts
- Start with a Vue component
- Organize code by logical concern
- Within logical concerns, write code in a consistent order
- Understand effect timing in Vue 3
- Abstract your function ref creation
- Abstract your ID generation
- For rendered lists, track index-based positions, not raw data
- Understand the limitations of ref objects
- Store elements in custom data structures
- Identify and abstract common logic
- Split up your event handlers
- Combine event listeners for rendered lists
- Take note of performance improvements
- Expose reactive references and methods in the return value
- Compare and contrast composables with directives
- Explore patterns for extending functionality
- Keep Tailwind markup readable
- Follow tactics to get world-class TypeScript skills
- Test in a real browser
- Recognize that accessibility leaves room for creativity
- Expand the web platform
- Get inspiration from other open-source projects
- Is Vue still simple?

About

Changelog

1.1.0

- Updated listbox code
  - In compound `Listbox` group, showed how to make `provide` and `inject` type-safe using Vue's `InjectionKey` type
  - Updated `useListbox` composable to use the latest version of Baleada Features, which includes API changes + better and simpler type safety for `bind` and `on` functions
  - Renamed `active` to `focused` to clarify purpose and have better parity with my production-grade listbox composable
- Converted all code examples to `script setup`
- Small revisions for clarity throughout all chapters
- Added lots of new content to Chapter 3, Authoring the function ref pattern
  - Understand the limitations of ref objects
  - Store elements in custom data structures
  - Combine event listeners for rendered lists
  - Take note of performance improvements
  - Compare and contrast composables with directives
  - Explore patterns for extending functionality
  - Keep Tailwind markup readable
  - Follow tactics to get world-class TypeScript skills
  - Expand the web platform
  - Is Vue still simple?

1.1.0 (continued)

- Added Changelog
- Added About
- Recorded a bunch of videos about Vue and other tools, all available on my YouTube channel

1.0.2

- Added table of contents and cover page

1.0.1

- Fixed broken links

1.0.0

- Published!

Introduction

Chapter summary

Reusability separates good Vue code from great Vue code, and the Composition API opens up new frontiers in reusability. Let's explore that concept.

Reusable code. It's the solution to bloated data stores, all-powerful "God" components, state & functions littering the global scope, prop drilling, naming collisions, and uncertainty & unpredictability.

I started reaping the benefits of reusable code after I took Adam Wathan's "Advanced Vue Component Design" course, and learned how to write highly reusable, Tailwind-friendly renderless and compound components.

When you're using a highly reusable Vue component, you feel like you're writing superpowered HTML, with tons of functionality and accessibility that just *works*, without requiring you to write custom, complex JavaScript, sift through documentation on component props, fight CSS specificity wars, or do anything that distracts you from designing and building a great UX.

The DX of reusable components is awesome! But in my opinion, the authoring experience was difficult...until we got the Vue Composition API.

Authoring reusable components is difficult

When you're authoring highly reusable components, like renderless or compound components, you need to have complete mastery of scoped slots, as well as `provide` and `inject` and render functions.

These specialized Vue APIs are complex on their own, but complexity *really* balloons when you add in your own equally specialized needs in accessibility, design, reactivity, avoiding memory leaks, integrating third party libraries, testing, bundling, browser compatibility, and everything else that goes into building a powerful, reusable feature in a Vue app.

Stitching together niche Vue APIs to meet your complex requirements is difficult!

Enter the Vue Composition API

The Vue Composition API lets us write `composables`—plain JavaScript functions that can create reactive state, perform reactive side effects, and hook into the component lifecycle.

We don't have to write these functions inside Vue components—we can write them in their own files or even in separate repositories. We can publish them as packages, and version them separately from the rest of our app, or our component libraries.

I've been exploring the possibilities of the Vue Composition API, and I've worked out the kinks in `the function ref pattern`, a composable pattern that improves the experience of authoring feature-rich, accessible user interfaces.

The function ref pattern is super useful for anyone who wants to keep their codebases readable and maintainable, and for anyone who believes that mastery of niche Vue APIs (scoped slots, `provide` and `inject`, and render functions) `shouldn't be a prerequisite` for a great authoring experience in Vue.

Rethinking Reusability in Vue

Reusability is what separates `good Vue code` from `great Vue code`.

To help you write great Vue code, I organized what I've learned about the function ref pattern into three chapters:

Chapter 1, `Reusable component pros and cons`, is a deep dive into the pros and cons of the renderless and compound component patterns—arguably the most powerful reusable component patterns that arose from Vue 2. It also hints at the improvements composables bring to the table.

Chapter 2, `Refactoring compound components`, introduces the function ref pattern for Vue 3 composables, and refactors a compound component into a function ref composable. The main goals are to reduce boilerplate, drastically simplify component communication, and cleanly collocate tightly coupled logic.

Chapter 3, `Authoring the function ref pattern`, is an unstructured deep dive into the function ref pattern, with detailed guidance on:

- Organizing code in the best way possible
- Replacing directives with function ref composables
- Using Tailwind CSS efficiently and effectively with function ref composables
- Different ways of thinking about UI problems and their solutions
- Much, much more

Don't forget the ripple effects 🎯

When we talk about reusability in the Vue ecosystem, we usually identify the core benefits:

- You no longer have to copy/paste large chunks of code multiple times
- When it's time to change your code, you only make the change in one place, and the rest of the app gets updated automatically
- After you abstract component logic into something more reusable, you don't have to wade through its complexity when you're polishing your app—changing a few styles, adding a bit of UI based on customer feedback, etc.
- You can more easily style reusable components to align with your brand or preferences.
- To achieve proper HTML semantics, reusable components either make sensible choices for you, or they let you make those decisions yourself.

Those benefits are great, but I get even more motivated to work on reusability when I look at the ripple effects.

You'll decrease your cognitive load

When UI logic is abstracted away into reusable composables, you no longer have to constantly confront that complexity. Your cognitive load goes down, and you can stay more focused on the task at hand.

This is *so* important when we're polishing our apps, or doing long term app maintenance.

The path to accessibility is much clearer

To create accessible user experiences, we need lots of UI logic to manage additional state, respond to keyboard events, trap focus, reactively update ARIA attributes, etc.

As we get better at extracting this logic into flexible, reusable patterns, we make it much easier to build bespoke accessibility features into our apps.

Vue newcomers are empowered

Building custom, logic-heavy features, like an accessible tablist or an autocomplete experience, has simply been out of reach for Vue newcomers in the past.

A new wave of reusable composables that don't rely on niche Vue APIs will make it easier for newcomers to build something they're proud of (or something they'll get paid for!).

Prerequisites

This book is written for Vue developers who are comfortable in Vue's Options API, and have at least played around with the Vue 3 Composition API a bit.

You `don't` need to understand niche Vue APIs like scoped slots, provide & inject, and render functions. That knowledge definitely helps, but in fact, I highly recommend this book for people who don't deeply know those APIs, since I'll be showing you how to make UI logic more reusable *without* using them.

As Vue 3 experience goes, you `don't` have to be an expert. Familiarity with `ref`, `computed`, `watch` & `watchEffect`, and the `setup` function is more than good enough, and if you want to feel really confident, I suggest a course like Ben Hong's "Launching with the Composition API".

If you're familiar with "template refs" in Vue 3, great! If you're even familiar with "function refs", even better! But neither of those is required——Chapter 2 covers that info in depth.

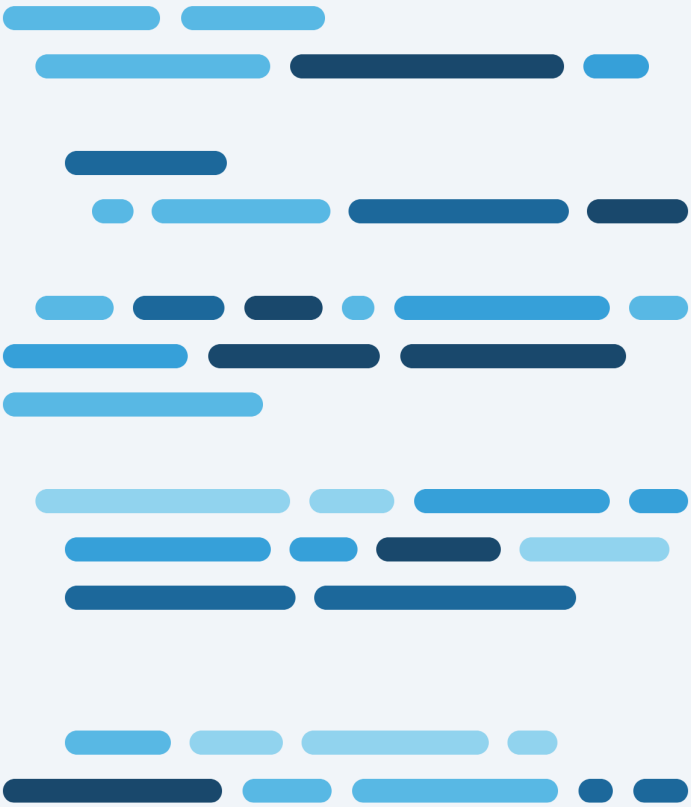
You `do` want to be comfortable reading Vue Single File Component syntax before you dive in.

Being comfortable with modern JavaScript, including basic destructuring, is always a plus, but not strictly required.

A handful of code snippets are written with very minimal TypeScript type annotations. If you don't write TypeScript, the annotations are minimal enough that you can ignore them.

150+ pages of writing, code, and data viz

Buy the book



Reusable component pros and cons

Chapter summary

Reusable components, e.g. renderless and compound components, are great user-facing APIs...but the authoring experience is tough.

Two reusable component patterns that spread from the React community into Vue 2 are "renderless components" and "compound components".

A renderless component is one that renders *none* of its own markup. Instead, it renders a slot or a scoped slot (known in React as "render props").

Adam Wathan's renderless tags input is a good example, and Michael Thiessen's renderless component experiments are definitely worth checking out too.

A compound component is part of a group of components that use `provide` and `inject` to share reactive state in their component tree. In a compound group of components, there is one parent component that manages most or all of that state, and there are descendant components that slot into the parent. Internally, those descendants modify and watch the parent's state, ready to react to modifications made by other descendants.

Also, the descendants can't be used outside of the parent component. Without the parent's reactive state (shared through `provide` and `inject`), the descendants break.

The Headless UI library is full of great examples compound components. Also check out Lachlan Miller's videos about render functions, which feature a reusable tabs component built with the compound component pattern.

In this chapter and the next, we're going to constructively criticize a `renderless, compound listbox`, whose source code can be found on GitHub in the repository accompanying the book.

Advanced components are *really* nice APIs

A nicely built renderless or compound component makes you feel like you're writing HTML with superpowers. Complex, accessible widgets like listboxes, grids, and modal dialogs become just another set of HTML tags to learn.

For example, take the `renderless tags input` mentioned above. Internally, it implements a bunch of cool features:

- It doesn't let you add duplicate tags
- It doesn't let you add empty tags
- It trims whitespace from tags
- Tags are added when the user presses `enter` on their keyboard
- Tags are removed when the user clicks the × icon

To benefit from all those features, you can write a component like the one shown on the next page.

```vue
<!-- CustomTagsInput.vue -->
<script setup>
import { ref } from 'vue'
import TagsInput from 'path/to/TagsInput'

const tags = ref([])
</script>

<template>
  <TagsInput
    v-model="tags"
    v-slot="{ removeTag, inputBindings }"
  >
    <span v-for="tag in tags">
      <span>{{ tag }}</span>
      <button
        type="button"
        @click="removeTag(tag)"
      >
        ×
      </button>
    </span>

    <!--
      Multiple attributes and event listeners get bound to
      the input element. They're all contained inside the
      inputBindings object, so that you can v-bind
      them more easily 👍
    -->
    <input
      placeholder="Add tag..."
      v-bind="inputBindings"
    />
  </TagsInput>
</template>
```

Note that, with the exception of `v-slot`, every line of code in the template is either plain HTML, or it's basic Vue syntax.

If you have that body of knowledge, you won't just be copy/pasting a code snippet like this and hoping it works—you'll actually be able to read and understand every single line of code you're writing.

On top of that, you'll have complete control over the semantic HTML you're using *and* any and all styling that gets applied—in other words, everything that you most often need to customize when reusing components within and across your projects.

And as for that `v-slot`—if you needed to deeply understand what it is and how it works, you could visit the docs on scoped slots. More often than not, though, a renderless component's documentation will tell you all you need to know about how to use `v-slot` with that component.

Compound components have a really similar feel to them. The compound listbox component I wrote for this book follows the WAI-ARIA listbox accessibility guidelines and implements these features:

For end users:

- End users can click an option to select it, or mouse over an option to focus it.
- End users can tab into the listbox to focus the selected option. From there, they can use up and down arrow keys to navigate the listbox, transferring focus to different options and causing assistive tech to read the focused option's text. Mac users can hold down Command while pressing arrow keys to quickly navigate to the top or bottom of the list of options.
- End users can hit `enter` or their spacebar to select the focused option.
- Assistive tech properly informs end users of listbox state, because accessibility attributes (namely `role`, `tabindex`, `aria-selected`, `aria-activedescendant`, and `aria-orientation`) are all managed automatically.

For developers:

- Developers can use `v-model` on the root `Listbox` component to control the value of the selected option.
- The root `Listbox` component renders a scoped slot, which has access to data describing listbox state, and methods to programmatically focus or select different options.
- The child `ListboxOption` components can be rendered with `v-for`. Each `ListboxOption` renders a scoped slot, which has access to methods that retrieve the focused or selected state of that specific option, and methods to easily and programmatically select the next or previous option.
- Since `Listbox` and `ListboxOption` components *only* render scoped slots, developers have full control over all markup and styles.

You can find a full example styled with Tailwind on GitHub, but for now, let's flip to the next page to see the markup and Vue template you would write to wire up the compound listbox.

```vue
<template>
  <Listbox
    :options="options"
    v-model="myOption"
    v-slot="{
      bindings,
      focused, focus, focusFirst, focusLast,
      selected, select
    }"
  >
    <ul v-bind="bindings">
      <ListboxOption
        v-for="option in options"
        :key="option"
        :option="option"
        v-slot="{
          bindings,
          isFocused, isSelected,
          focusPrevious, focusNext
        }"
      >
        <li v-bind="bindings">
          <span>{{ option }}</span>
          <CheckIcon v-show="isSelected()" />
        </li>
      </ListboxOption>
    </ul>
  </Listbox>
</template>

<script setup>
import { ref } from 'vue'
import { CheckIcon } from '@heroicons/vue/solid'
import { Listbox, ListboxOption } from './Listbox'
import { options } from 'path/to/options'

const myOption = ref(options[0])
</script>
```

This component happens to be both compound *and* renderless. Not all compound components are renderless—some, like the Headless UI Listbox, render minimal, customizable markup.

Regardless, there are always lots of similarities between renderless and compound components.

The basic developer experience is similar: you're writing plain HTML, sprinkled with custom HTML tags, basic Vue syntax like `v-model` and `v-for`, and the occasional use of `v-slot` to access useful things provided by the components in the compound group.

To read and write this code effectively, you'll need to know how the components in a compound group are actually supposed to fit together. For example, our `ListboxOption` has to nest inside our root `Listbox`.

Conceptually, though, this is exactly the same as nesting an HTML `option` inside of a `select`. Alone or in the wrong order, the child elements break, but nested together in the correct order, they take on additional meaning and functionality.

> A nicely built renderless or compound component makes you feel like you're writing HTML with superpowers.

Authoring reusable components is tough 👎

Usually, writing Vue components is a great experience! Vue's custom Single File Component format makes it easy to keep related markup, styles, and UI logic all in the same file.

In my experience, renderless and compound components are the exception to the rule.

Admittedly, renderless components aren't as tricky in Vue 3 as they were in Vue 2. In Vue 3, we can render a single empty slot as the only element in our Vue template, so the renderless component boilerplate looks like this:

```
<!-- MyRenderlessComponent.vue -->
<template>
  <!--
    If you bind data to this slot, it becomes a
    scoped slot.
  -->
  <slot />
</template>

<script setup>
// Normal Vue setup code goes here
</script>
```

This is a big step up from Vue 2, where our only option was to write a render function, using the `this.$slots` or `this.$scopedSlots` API to render the slot.

But compound components, even in Vue 3, are still complex and verbose.

Authoring compound components is *really* tough.

Complexity and verbosity run rampant in compound components.

At a basic level, you need to author multiple Vue components when building a compound group, so you need to repeat essential component boilerplate multiple times.

If you're writing components in separate files, this means repeating the renderless component boilerplate in each one.

On the other hand, if you're writing all components in a single `.ts` or `.js` file, you can't use Vue templates, so you'll have to repeat some render function boilerplate as well:

```
export const Root = {
  setup: (props, { slots }) {
    ...
    return () => slots.default({ ... })
  }
}

export const Child = {
  setup: (props, { slots }) {
    ...
    return () => slots.default({ ... })
  }
}

export const AnotherChild = {
  setup: (props, { slots }) {
    ...
    return () => slots.default({ ... })
  }
}
```

All this is to say: as your compound group of components grows, you'll repeat more lines of code that simply scaffold Vue components, and don't directly add functionality.

Is this boilerplate unbearable or horrendous? No. But it's `verbose and not optimal`, in my opinion.

Compound component communication is complex.

Something that *does* feel more unbearable is a common complexity found in compound components: frequent use of `provide` and `inject`.

`provide` and `inject`! They're really useful when you need a child component to read or edit some data in a potentially distant parent component. Inside the parent component, you'll tell the component to "provide" some data, and inside the child component, you'll tell the child to "inject" that data.

When the child component gets created, it will walk up the component tree, looking for the nearest parent that's providing the data the child wants to inject.

The `Listbox` compound component we're studying in this book has a perfect example of where this feature comes in handy. To explore why and how, let's examine a small slice of logic this component needs to implement.

First, per WAI-ARIA guidelines, each option element (rendered by the `ListboxOption` components) must have an ID, and the root element (rendered by `Listbox`) must have an `aria-activedescendant` attribute whose value is the ID of the currently focused option.

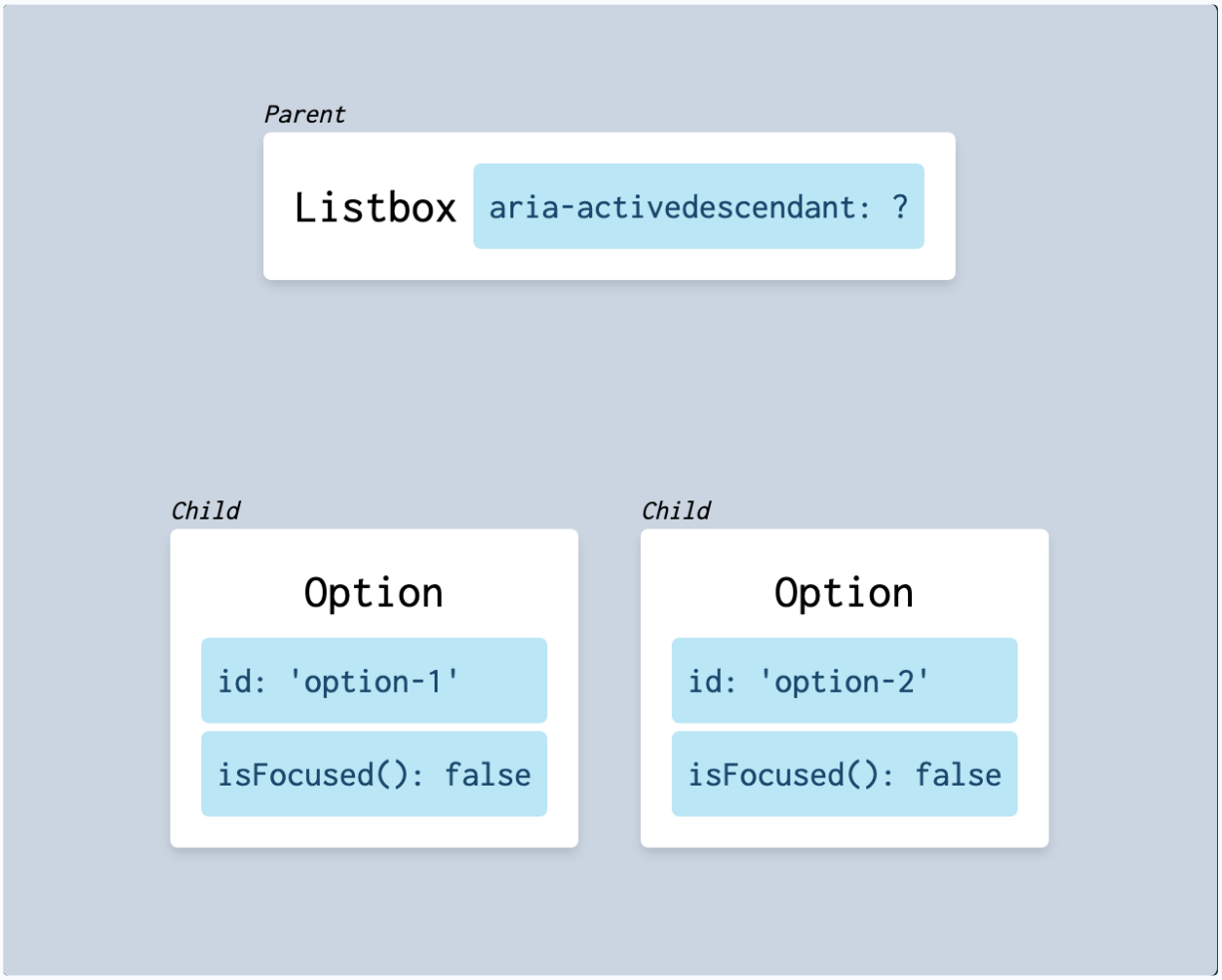
Also, to make it easier for developers to apply styles and classes to the focused option, the `ListboxOption` component should provide an `isFocused` function to its scoped slot, which returns `true` for the focused option.

Finally, when an option element detects a `mouseenter` event, it should become the focused item. It should notify the root element to update its `aria-activedescendant`. Its `isFocused` function should return `true`, and the `isFocused` function of every other `ListboxOption` should return `false`.

So, inside of this component, we've encountered a situation where a parent component—the `Listbox`—needs to be aware of reactively changing data: the ID of the focused descendant.

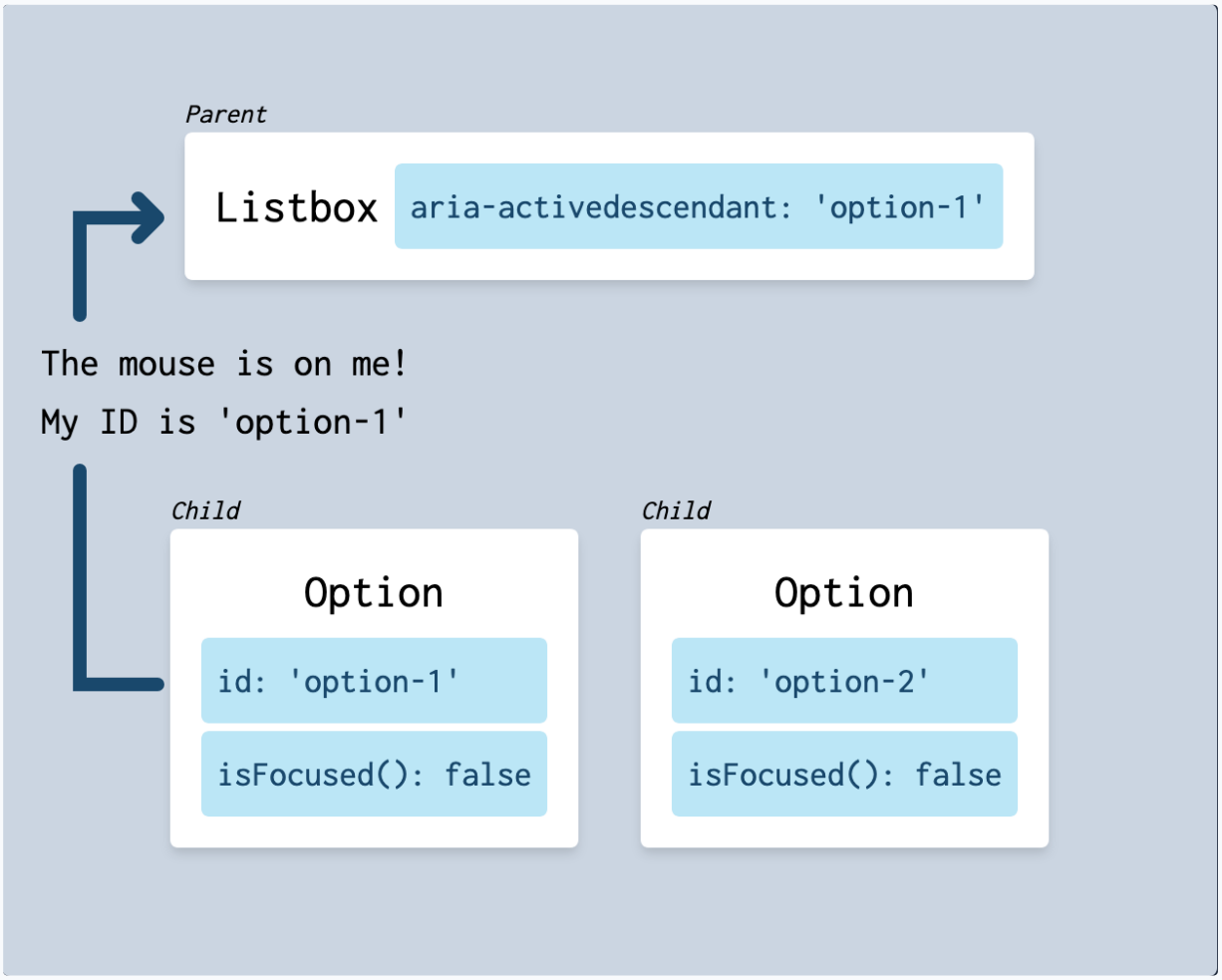
Each child of that parent—the `ListboxOption` components—also need to be aware of which option is currently focused, and they need a way to notify the `Listbox` when their `mouseenter` event happens.

Here's a diagram of the basic component tree structure, and the reactive data we're working with:



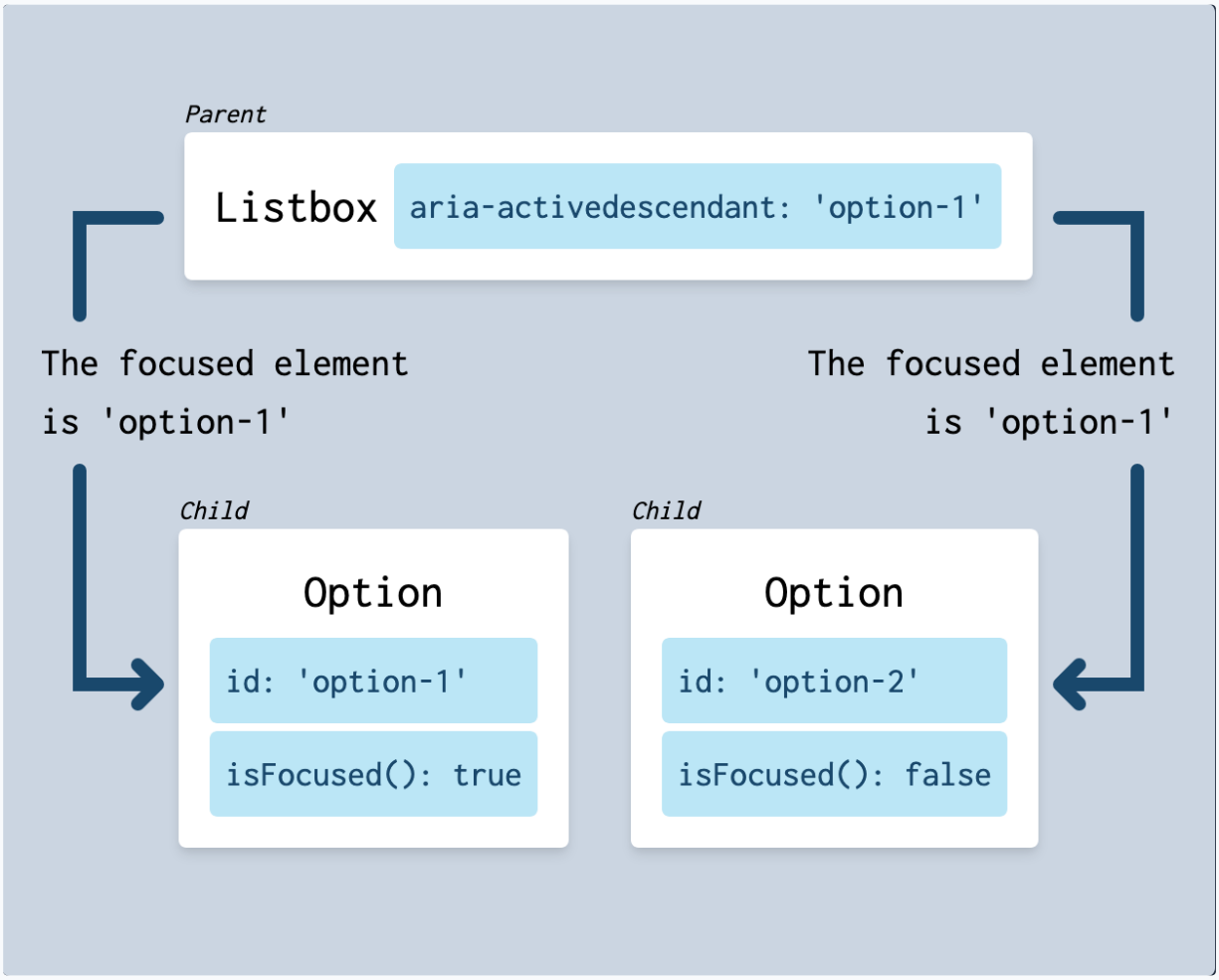
When the end user's mouse enters a `ListboxOption`, that descendant component needs to tell the `Listbox` parent component, "Hey, the mouse is on me! Here's my ID."

Note in this diagram that the `Listbox` receives the ID `option-1` and uses it to update the `aria-activedescendant` attribute.



That information, after flowing up to the `Listbox`, needs to trickle back down through the component tree. Each `ListboxOption` needs to be notified that the active descendant has changed, and they need to know if they are that item. If so, their `isFocused` function returns `true`.

Note in this diagram that `option-1` now has `isFocused()` returning `true`, while it still returns `false` for `option-2`:



In some ways, this feels like basic component communication. You might instinctively think the `ListboxOption` can use Vue's `emit` feature to emit an event when the mouse enters, and `Listbox` can listen for that event.

To move reactive data in the other direction, back down the component tree, you might think that each `ListboxOption` can accept a prop to keep track of the active descendant.

Props and emit work in many cases, but they get verbose and difficult to maintain for large or more deeply nested component trees, with lots of reactive data to keep track of.

Moreover, props and emit are not at all viable for our compound listbox. To see why, let's take another quick look at the API we're trying to expose to the developers who would actually use these components:

```
<template>
  <Listbox
    :options="options"
    v-model="myOption"
    v-slot="{
      bindings,
      focused, focus, focusFirst, focusLast,
      selected, select
    }"
  >
    <ul v-bind="bindings">
      <ListboxOption
        v-for="option in options"
        :key="option"
        :option="option"
        v-slot="{
          bindings,
          isFocused, isSelected,
          focusPrevious, focusNext
        }"
      >
        <li v-bind="bindings">
          <span>{{ option }}</span>
          <CheckIcon v-show="isSelected()" />
        </li>
      </ListboxOption>
    </ul>
  </Listbox>
</template>
```

In that sleek template, do you see any `activeDescendant` prop being passed down to the `ListboxOptions`? Do you see any indication that the `ListboxOption`s are passing the `mouseenter` event up to the `Listbox`? The answer in both cases is a resounding `no`.

And that's a really great thing—it means that the developers using these compound components don't have to think about the implementation details of component communication for this accessibility feature, let alone all the other `Listbox` features, every single time they try to use this compound group in their apps.

It also means that we're left with only one final solution for component communication: `provide` and `inject`.

Instead of using props to tell our `ListboxOption`s which option is focused, we would `provide` that data from the `Listbox` component and `inject` it into each `ListboxOption`. And instead of using `emit` to tell the `Listbox` when the mouse has entered a `ListboxOption`, we would use a technique that is common in React, but rarely used in Vue:

1. The `Listbox` component sets up a reactive reference to track the ID of the focused descendant.
2. The `Listbox` component also sets up an `focus` method that can update that reactively tracked data.
3. The `Listbox` component uses `provide` to expose that `focus` method to all of its descendants in the component tree.
4. The `ListboxOption`s `inject` the `focus` method, and call it whenever they need to update the reactive reference.

The `provide` and `inject` solution, while very effective and pretty interesting, carries its own significant downsides:

- In our listbox, we're looking at the simplest possible `provide`/`inject` relationship: a parent providing data to a single layer of direct children. Component communication gets *way* more complex when you have a professional-grade, production-ready compound listbox, with multiple layers of nested components in a compound group, all reading and writing reactive data inside a shared parent at the same time.
- `provide` and `inject` are *much* cleaner and simpler in Vue 3 compared to Vue 2, but they still represent a big chunk of boilerplate code for every compound component group you write.
- I suspect that the majority of Vue developers are comfortable with props and emit, but pretty unfamiliar with that idea of passing down a function that a child can call to update reactive data in the parent component. When I first learned React after using Vue, that technique (standard practice in React) totally threw me for a loop.
- `provide` and `inject` behavior and syntax are pretty specific to Vue. As soon as you use them in your components, you make it significantly more difficult for you and others to port that component to React, Svelte, or any other component framework.
- `provide` and `inject` are not mainstream Vue features. Lots of component authors could benefit from the compound component pattern, but since `provide` and `inject` are relatively obscure, yet so integral to the pattern, compound components are out of reach for many (if not the majority) of Vue developers.
- All of this code and communication logic doesn't meaningfully enhance the UI logic we're trying to achieve in our `Listbox`. It's just a bunch of complexity we encounter while working within the constraints of the compound component pattern in search of an API that is truly pleasant to use and reuse.

> `provide` and `inject`! They're effective and interesting, but relatively obscure, and can get dizzyingly complex.

Compound components split up tightly coupled logic.

Compound component complexity also stems from the limitations of single file components. In a `.vue` file, you can only write one component. One `template` tag, one `script` tag, one `style` tag, maximum.

This leaves compound component authors with two options:

1. Split up your compound group into multiple `.vue` files, or
2. Write your entire compound group of components in one `.js` file.

Both of these options have downsides.

If you write multiple `.vue` files, you'll be forced to arbitrarily split up tightly coupled logic across your files. When you're working on an individual feature, you'll constantly be flipping back and forth between your files, trying to keep that logic straight in your head. Any kind of deep work will be a distant dream.

If instead you write in a single `.js` file, you'll still be splitting up tightly coupled logic, but you'll scroll up and down in your single file instead of flipping between multiple files. Arguably less jarring, but still far from ideal.

More importantly, since you're in a plain old `.js` file, you won't be able to write Vue templates. Instead, you'll write—you guessed it—JavaScript render functions!

Don't get me wrong—I think the render function API is extensive, fantastic, and necessary, and with my code as my witness, I love writing JS!

But render functions suffer from one of the same problems as `provide` and `inject`: they're integral to the compound component pattern, but they're still a niche, obscure feature. Their relative obscurity puts the compound component pattern out of reach for most Vue devs.

Seriously though, even wizards can't read compound components.

A while back, I saw a Tweet that sums up the problems that plague compound components in Vue:

Let's unpack that a little.

First, it's important to note that Rich Harris is the creator of the frontend framework Svelte and the JavaScript module bundler Rollup, and he spent years building interactive data visualizations with The New York Times.

In other words, this dude knows JS like the back of his hand, and he's a full-on expert in reactivity who likely also has serious chops in component design.

I also know from a talk he gave that he thoroughly understands Vue's internal architecture.

In other other words, Rich Harris is a wizard.

In that tweet, he's replying to a thread about a `Listbox` compound similar to the one we've been studying, saying he can't rebuild it in Svelte because he "can't make heads or tails" of it.

The `Listbox` compound group he's talking about is an early version of the Headless UI compound listbox. Relative to other compound components, it certainly isn't the simplest application of the pattern, but I wouldn't say it's exceedingly complicated either. It's also masterfully written, with all code organized as best as the Vue 2 API allows.

If an expert in JavaScript, reactivity, and component design, with intimate knowledge of Vue's internals, can't make heads or tails of a well-organized, moderately-complex compound component in Vue, what hope do the rest of us have of understanding them, let alone writing them?

> Usually, writing reusable components is a great experience! In my opinion, compound components are the exception to the rule.

Where do we go from here?

Great news: the Vue Composition API solves a lot of problems for Vue authors!

The composition API opens up a new realm of possibilities for reusing logic through composables. Composition functions open the door to:

- Reduced boilerplate
- Drastically simpler component communication
- Clean collocation of tightly coupled logic

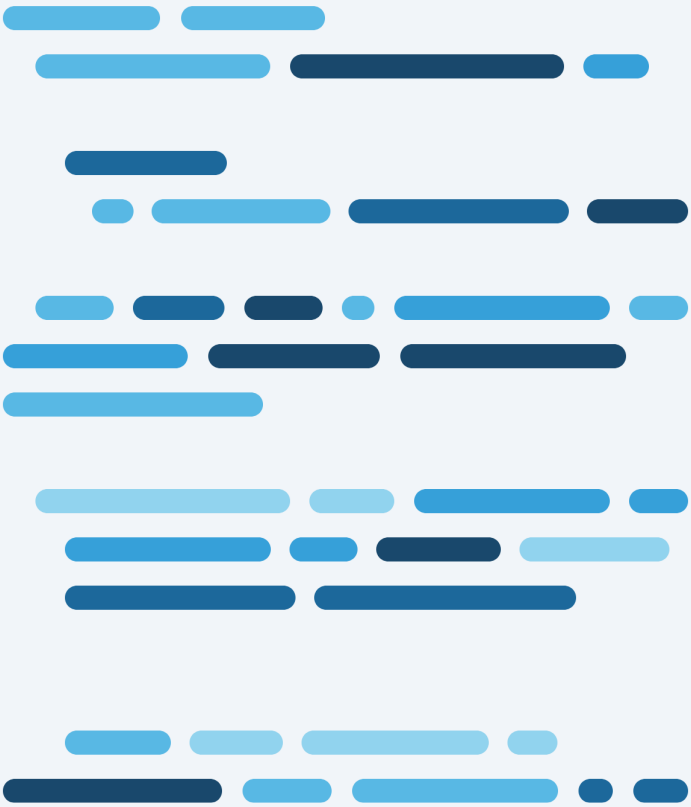
For anyone who's ever been discouraged or defeated by the downsides of renderless components and especially of compound components, the Vue Composition API is a dream come true.

In the next two chapters, we're going to dive deeper into how the function ref pattern for composables will make that all work, and we'll see how the APIs we expose to other developers become even cleaner and more flexible.

Next stop: Chapter 2, Refactoring compound components.

150+ pages of writing, code, and data viz

Buy the book



About

I'm Alex Vipond, the author of Rethinking Reusability in Vue.

I'm a front end developer with BetterHelp, and I also created Baleada, an open source toolkit for web dev, especially Vue apps. I put all of my book's lessons into practice in my work at BetterHelp and in the Baleada Features package.

I've been in web dev for about 6 years, and a technical writer for a while longer. One of my favorite contributions to the Vue community was my talk at VueConf Toronto 2021, "Organizing Code by Logical Concern in Vue 3".

I wrote Rethinking Reusability in Vue primarily for Vue developers who are comfortable in the Options API, have at least played around with the Composition API, and are interested in learning new Vue 3 patterns for reusability.

I also put a lot of effort into making sure devs who are experienced in other front-end frameworks, like React, could still get value from the book. I'm already using this book's patterns in BetterHelp's React app, shipping highly reusable React code to thousands of licensed therapists and millions of their clients. This stuff works!

I'm active on GitHub and on Twitter, and you can email me too. Send me your thoughts or questions about the book!