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
- Identify and abstract common logic
- Split up your event handlers
- Expose reactive references and methods in the return value
- Test in a real browser
- Recognize that accessibility leaves room for creativity
- Get inspiration from other open-source projects

Introduction

Chapter summary

It's time to rethink and revise design patterns in Vue 3. "Rethinking Reusability in Vue" explores that concept.

I've been hooked on `reusability` in my Vue code ever since I took an advanced Vue component design course in late 2018.

When you're using a highly reusable 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, read through piles of documentation on extensive component props, fight CSS specificity wars, or do anything that distracts you from designing and building a great user experience.

The developer experience of reusable components is awesome! But in my opinion, Vue 3 lets us improve on the authoring experience.

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, 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 gives us the ability to 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

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, giving further recommendations and detailed technical guidance on how to organize code, how to approach reactivity and DOM side effects, how to refine your function's API and intended usage, and 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 the Vue 2 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 should be minimal enough that you can ignore them.

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 dialogs/modals 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 explained by Vue's "Introduction" guide.

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 "activate" 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 active 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 active 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 activate 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 active 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,
      active, activate, activateFirst, activateLast,
      selected, select
    }"
  >
    <ul v-bind="bindings">
      <ListboxOption
        v-for="option in options"
        :key="option"
        :option="option"
        v-slot="{
          bindings,
          isActive, isSelected,
          activatePrevious, activateNext
        }"
      >
        <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:

```
// CompoundComponent.js
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 active option.

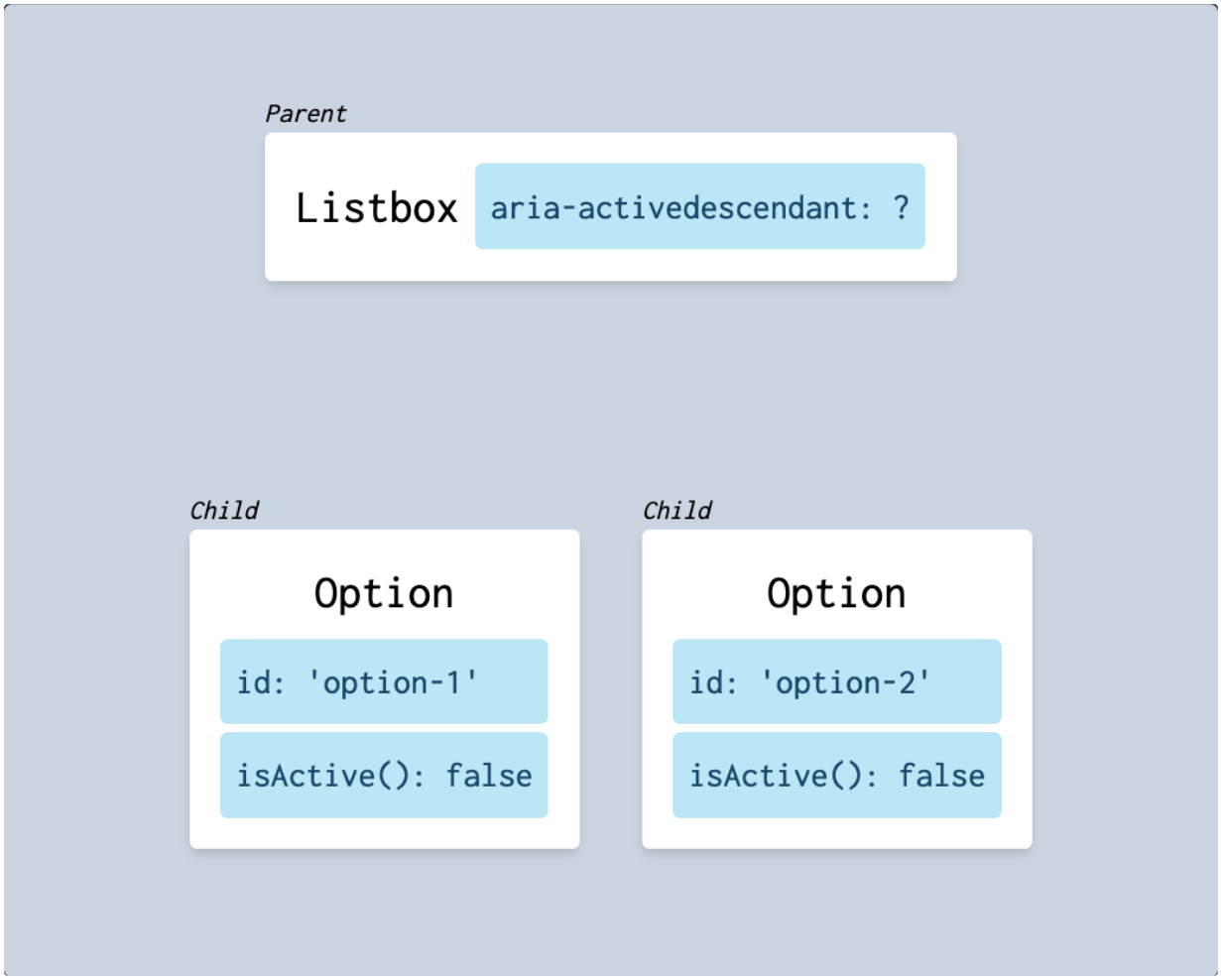
Also, to make it easier for developers to apply styles and classes to the active option, the `ListboxOption` component should provide an `isActive` function to its scoped slot, which returns `true` for the active option.

Finally, when an option element detects a `mouseenter` event, it should become the active item. It should notify the root element to update its `aria-activedescendant`. Its `isActive` function should return `true`, and the `isActive` 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 active descendant.

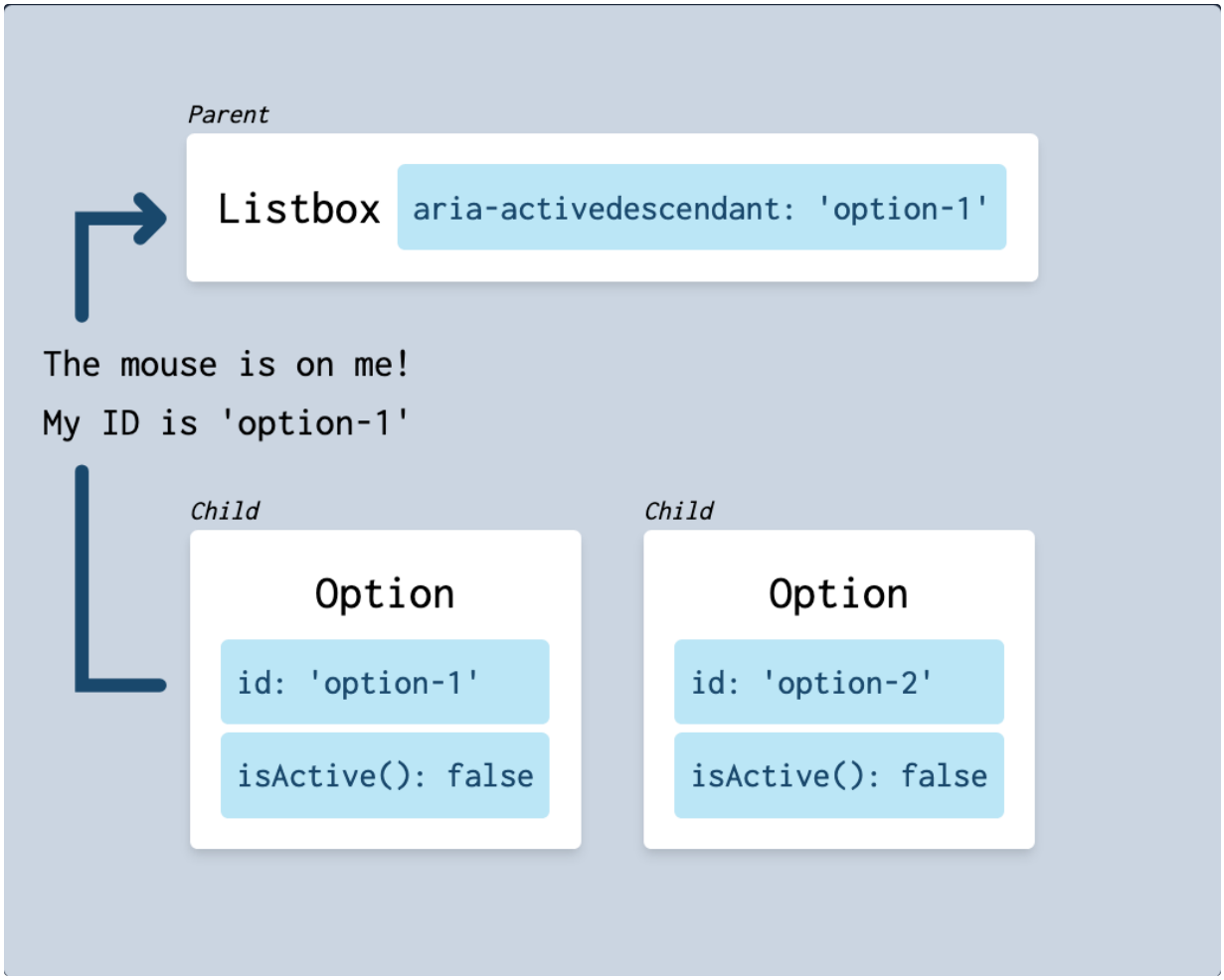
Each child of that parent—the `ListboxOption` components—also need to be aware of which option is currently active, and they need a way to notify the `Listbox` when their `mouseenter` event happens.

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



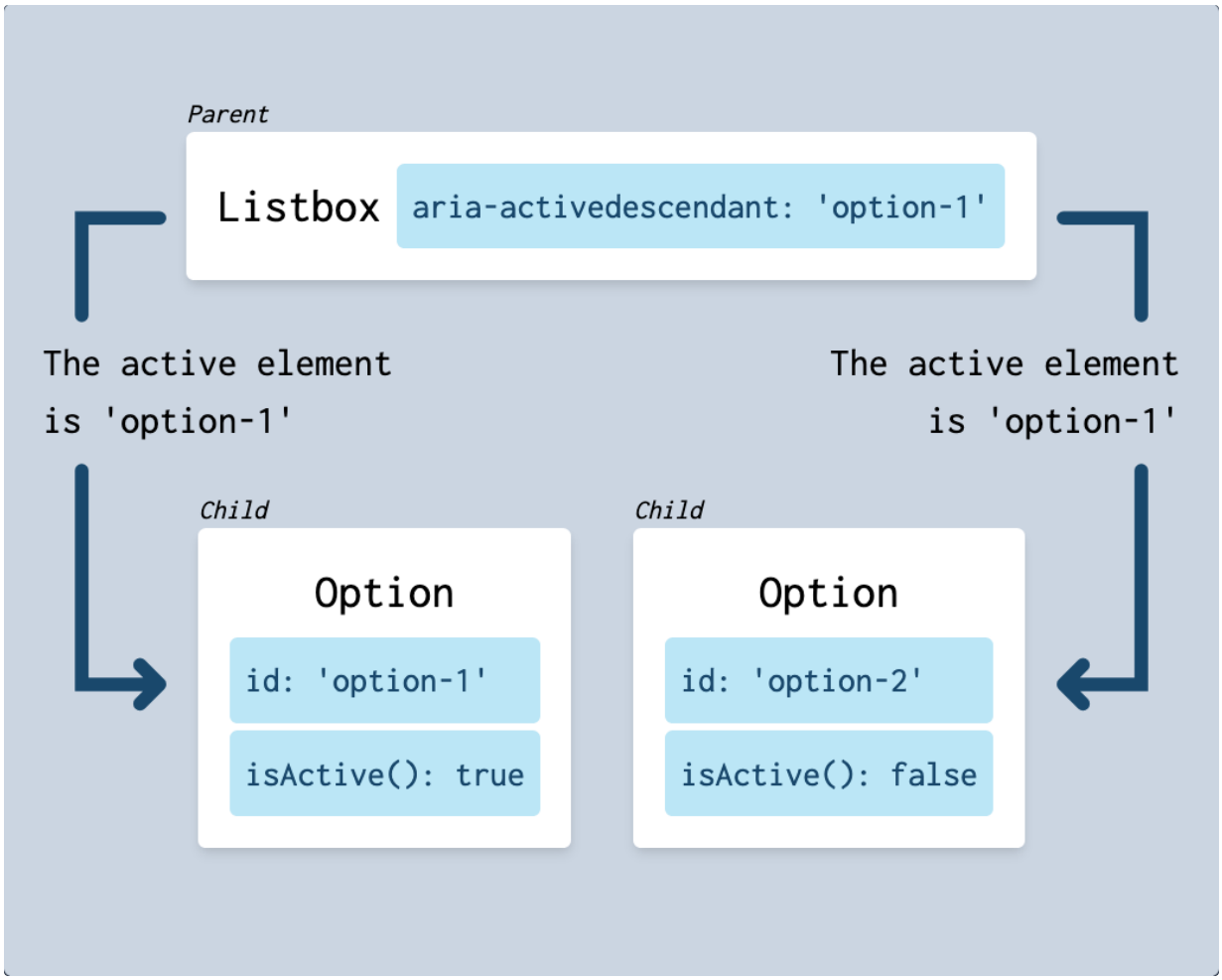
When the end user's mouse enters a `ListboxOption`, that descendant component needs to tell the `Listbox` parent component, "Hey, the mouse is on me! Here's my ID."

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 `isActive` function returns `true`.

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



In some ways, this feels like basic component communication. You might instinctively think the now-active `ListboxOption` can use Vue's `emit` feature to emit an event when the mouse enters, and `Listbox` can listen for that emitted 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 for larger or more highly nested component trees, with lots of reactive data to keep track of, it gets verbose and difficult to maintain.

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,
      active, activate, activateFirst, activateLast,
      selected, select
    }"
  >
    <ul v-bind="bindings">
      <ListboxOption
        v-for="option in options"
        :key="option"
        :option="option"
        v-slot="{
          bindings,
          isActive, isSelected,
          activatePrevious, activateNext
        }"
      >
        <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 active, 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 active descendant.
2. The `Listbox` component also sets up an `activate` method that can update that reactively tracked data.
3. The `Listbox` component uses `provide` to expose that `activate` method to all of its descendants in the component tree.
4. The `ListboxOption`s `inject` the `activate` 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 code 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 for a long while, that technique—which is 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 your compound group of components up 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 this 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`.

Refactoring compound components

Chapter summary

Converting reusable components over to the function ref pattern is a great way to reduce boilerplate, drastically simplify component communication, and cleanly collocate tightly coupled logic.

At the end of Chapter 1, I promised that Vue 3 composables, specifically those that follow the `function ref pattern`, would improve on the compound component authoring experience.

In this chapter, we're going to deeply test that hypothesis by refactoring a compound component to a function ref composable. Before we get started, let's learn what a "function ref" is.

What is a "function ref"

In Vue, a "template ref" is a reactive reference to a DOM element or component instance.

```
<script setup>
import { ref } from 'vue'

// First, create a reactive reference by calling the
// `ref` function from the Vue Composition API, with
// an initial value of `null`:
const button = ref(null)
</script>

<template>
  <!--
    In your template, add the `ref` attribute to an element,
    and assign the name of your template ref.

    This name should match the name of the *key* on the
    setup function's return object that points to your
    reactive reference.

    When the component mounts, Vue changes the value of your
    reactive reference from `null` to the actual HTML button
    element seen here.

    In that moment, your reactive reference becomes a template
    ref!
  -->
  <button ref="button"></button>
</template>
```

A "function ref" is a specific type of template ref.

At its most basic level, a function ref is a JavaScript function that accepts a DOM element as an argument, and stores that DOM element as the value of a reactive reference.

```
<script setup>
import { ref } from 'vue'

const button = ref(null)

// This function is a function ref!
function myFunctionRef (element) {
  button.value = element
}
</script>
```

So how do you use this in a Vue template? As shown in the code on the next page, we just need to make one small change: instead of assigning a string to the `ref` attribute on our element, we bind our function to the `ref` attribute.

```vue
<script setup>
import { ref } from 'vue'

const button = ref(null)

function myFunctionRef (element) {
  button.value = element
}
</script>

<template>
  <!--
    We *bind* `myFunctionRef` to the `ref` attribute,
    instead of assigning a string.

    This small change instructs Vue to wait until the
    component is mounted, then call `myFunctionRef`, passing
    the button element as the first and only argument.

    `myFunctionRef` then receives that element and sets it
    as the value of the `button` reactive reference in
    our `setup` function.
  -->
  <button v-bind:ref="myFunctionRef"></button>
</template>
```

Okay, that makes sense for individual DOM elements, but what about elements created by `v-for`?

Vue can handle that too! When you bind a function ref to the `ref` attribute of a `v-for` element in your template, Vue will call that function for each and every DOM element.

So what do we do—create a separate reactive reference for each element? Nope—instead, we create a reactive reference to an `array`, and we store our DOM elements in that array.

When we do this, there are two important "gotchas" to be aware of:

1. To store a DOM element, you have to assign it directly to an index of your reactive array. If you try to use `array.push(element)`, you'll create an endless render loop, and your computer will explode.
2. Vue calls your function ref every time the component updates, so you have to empty the array using the `onBeforeUpdate` lifecycle hook.

Let's see how this all looks in code:

```
<script setup>
import { ref, onBeforeUpdate } from 'vue'

// Our reactive reference should be initialized as an
// array when dealing with v-for elements:
const listItems = ref([])

// For v-for elements, our function ref will also
// accept an `index` parameter, so that the function
// know where in the array the element should go.
//
// We'll get this `index` from the template in just a sec.
function myFunctionRef (element, index) {
  // Don't use the `push` method!
  // Assign directly to the index:
  listItems.value[index] = element
}

// Empty out the array before the component updates,
// to make sure we never have stale references to old
// elements that no longer exist in the DOM.
onBeforeUpdate(() => {
  listItems.value = []
})
</script>
```

Code continues on the next page

```vue
<template>
  <ul>
    <!--
      When you bind a function to the `ref` attribute of
      a v-for element, Vue will call that function for each
      element in the list after the component is mounted.

      However, Vue will *only* pass the element to your
      function—it won't pass the `index` we need.

      But, using parentheses at the beginning of the v-for
      statement, we can get access to the `index` that
      our function ref needs.
    -->
    <li
      v-for="(item, index) in list"
      :key="item"
      :ref="element => myFunctionRef(element, index)"
    ></li>
  </ul>
</template>
```

With that done, we can reliably reach into the `listItems` array at any time after the component is mounted, and we can trust that it will be filled with DOM elements. The order of that array will always match the exact order of the elements in the DOM itself.

In the code we just saw, note that developers have to build their own arrow function in the `v-for` list so that they can pass the `index` into our function ref.

I'm not a huge fan of that developer experience—it just feels a little bit more verbose and brittle. Instead, I prefer to define a function ref getter for `v-for` use cases.

A "function ref getter" is a function that accepts an index and returns the correct function ref for that index.

Here's what that looks like in practice:

```
<script setup>
import { ref, onBeforeUpdate } from 'vue'

const listItems = ref([])

// Instead of accepting the index as the second argument
// of our function ref, we can define a function ref getter.
//
// This function ref getter accepts the index as its
// only argument, and returns a function ref that will
// store the element in the correct position in our
// array.
function getFunctionRef (index) {
  // Inside the getter, we'll define a new function ref:
  function functionRef (element) {
    // From inside the scope of `functionRef`, we can
    // access both the `index` and the `element`, so
    // we can store the element correctly.
    listItems.value[index] = element
  }

  // Our function ref getter returns the new function ref
  // that we just defined.
  return functionRef
}

onBeforeUpdate(() => {
  listItems.value = []
})
</script>
```

Code continues on the next page

```
<template>
  <ul>
    <!--
      Now, in our template, we can get rid of the custom
      arrow function that each developer would have been
      forced to write.

      Instead, developers can just bind `getFunctionRef`
      to the `ref` attribute, passing in the index.

      At runtime, Vue will call `getFunctionRef` for each
      element in the list. So, each element will pass its index
      to `getFunctionRef` and create its own function ref.

      All elements will still be stored in the array in the
      correct order.
    -->
    <li
      v-for="(item, index) in list"
      :key="item"
      :ref="getFunctionRef(index)"
    ></li>
  </ul>
</template>
```

Using a function ref getter isn't necessary for `v-for`, but in my experience, the getter's usage is easier to follow and document.

To see how function ref getters improve developer experience, consider this code, where we return one function ref and one function ref getter from a composable, then bind them to a template:

```
<script setup>
import { useThing } from 'path/to/useThing'

// Our imagined `useThing` composable returns two
// function refs: one for a `ul` element, and one for
// `li` elements rendered by v-for.
const { ulRef, getLiRef } = useThing()
</script>
<template>
  <!--
      In the template, we bind these refs to the
      appropriate elements.
    -->
  <ul :ref="ulRef">
    <li
      v-for="(item, index) in list"
      :key="item"
      :ref="getLiRef(index)"
    ></li>
  </ul>
</template>
```

Note in that code how there are no arrow functions in sight.

Imagine you're a developer who has no idea what a function ref is. In theory, you could just follow instructions from the `useThing` documentation to bind `ulRef` and `getLiRef` in their correct places. Even with zero knowledge that you're actually binding functions to those `ref` attributes, you'd still be able to take full advantage of the composable's features.

That's pretty cool! And that's why I generally prefer to use function ref getters for elements rendered by `v-for`.

Why use function refs?

Why learn this somewhat niche Vue 3 feature? Weren't we trying to *avoid* niche features?

The answer is that it's a worthwhile tradeoff. By refactoring from compound components to function ref composables, we have to learn one niche feature (function refs), but we get to avoid three niche features (scoped slots, render functions, and `provide` & `inject`).

It also improves the experience of developers using our code. Developers who use compound components generally don't need to know anything about render functions or `provide` & `inject`, but they definitely need to understand scoped slots.

Developers who consume our function ref composables, though, will not need to understand or even be aware of scoped slots, render functions, `provide` & `inject`, or function refs. Everybody wins!

Finally, learning function refs allows us to take full advantage of the composition API, so we can achieve three goals:

- Reduce boilerplate
- Drastically simplify component communication
- Cleanly collocate tightly coupled logic

To test that hypothesis, I refactored the compound listbox from Chapter 1 into a single `useListbox` composable, following the function ref pattern.

As I found out, you don't need multiple composables or multiple files to achieve these goals. In my experience, entire groups of compound components stored in multiple `.vue` files (or a large `.js` file with render functions) can be refactored into a single composable in a single `.js` file.

That single composable doesn't render any markup or impose any styles—it only manages reactive state and performs DOM side effects. The function returns reactive data, useful methods, and most importantly function refs that the developer can attach to their component's template. Hence, "the function ref pattern".

I'll be referencing snippets from the compound listbox and the `useListbox` composable throughout the rest of this chapter, but you can also see their full source code on GitHub:

- `Listbox` group
- `useListbox`

For the next section of this chapter, we're going to study how the compound group tracks which option is active, then we'll refactor all of that code to the function ref pattern.

Let's study a compound component

A core feature of the compound listbox is that it tracks which option is currently `active`.

The compound group also handles a lot of UI logic related to the active item:

- It updates the `aria-activedescendant` attribute of the root element to the HTML `id` of the active option
- End users can hover their mouse over an option to activate it
- Once an option is active, end users can use the up and down arrow keys to activate the previous or next option
- It exposes methods to the `Listbox` and `ListboxOption` scoped slots so that slots can easily know which option is active, and programmatically update it if need be.

Right now, we're going to study how that works inside the compound listbox, then refactor the code to the function ref pattern.

Keep in mind that our goal here is not to deeply understand the inner workings and design of the compound listbox. Our goal is to `get familiar with patterns`.

As we read `Listbox` code, you'll see how I defined reactive references and useful methods, and how I wired up the components in this compound group to communicate with one another. Focus your attention more on those abstract patterns than on the listbox implementation details.

Initialize component state

To lay the groundwork for this feature, the top-level `Listbox` component stores three pieces of state:

- `active`, a reactive reference to the currently active option (String)
- `ids`, a non-reactive object that stores options and their `id`s as key/value pairs, so that `Listbox` can look up any `id` when needed
- `ariaActivedescendant`, a reactive reference to the `id` of the active option

```
// Listbox.ts
export const Listbox = defineComponent({
  ...,
  props: {
    options: { type: Array as PropType<string[]> },
    ...,
  },
  setup (props, { slots, emit }) {
    ...
    const active = ref(props.options[0])

    // Here, we set up a non-reactive object to store IDs.
    const ids = shallowRef({})
    // We'll see  later how this function gets passed down to
    // `ListboxOption`s so they can generate & store their own IDs.
    const storeId = (option: string, id: string) => {
      ids.value[option] = id
    }

    // aria-activedescendant is the ID of the active option.
    const ariaActivedescendant = computed(() => {
      return ids.value[active.value]
    })
    ...
  }
})
```

Render a scoped slot

The `Listbox` component also returns a render function from its `setup` option. This render function only renders a scoped slot, and it uses that slot to expose the binding for the `aria-activedescendant` attribute on the root listbox element.

```
// Listbox.ts
...

export const Listbox = defineComponent({
  ...,
  setup (props, { slots, emit }) {
    ...
    const active = ...
    const ids = ...
    const storeId = ...
    const ariaActivedescendant = ...
    ...

    // `Listbox` returns a function that returns
    // the default slot, with data passed as the first
    // argument. This is the proper way to render scoped
    // slots when you're returning a render function from
    // the `setup` option.
    return () => slots.default({
      bindings: {
        ...,
        'aria-activedescendant': ariaActivedescendant.value,
      },
      ...,
    })
  }
})
```

Write methods

The `Listbox` component also contains the methods that activate specific options, as well as a method that tells you if a given option is active:

```
// Listbox.ts
export const Listbox = defineComponent({
  ...,
  setup (props, { slots, emit }) {
    ...
    const activate = (option: string) => {
      active.value = option
    }

    const activatePrevious = (option: string) => {
      const index = props.options.indexOf(option)
      if (index === 0) return
      active.value = props.options[index - 1]
    }

    const activateNext = (option: string) => {
      const index = props.options.indexOf(option)
      if (index === props.options.length - 1) return
      active.value = props.options[index + 1]
    }

    const activateFirst = () => {
      active.value = props.options[0]
    }

    const activateLast = () => {
      active.value = props.options[props.options.length - 1]
    }

    const isActive = (option: string) => {
      return option === active.value
    }
  }
})
```

Provide references and methods to the component tree

The `Listbox` component never actually calls any of its methods, nor does it make any UI decisions based on `active`. Instead, it `provides` methods to its descendant components using `provide`:

```
// Listbox.ts
...

const ListboxSymbol = Symbol('Listbox')

export const Listbox = defineComponent({
  ...,
  setup (props, { slots, emit }) {
    ...

    provide(ListboxSymbol, {
      activate,
      activatePrevious,
      activateNext,
      activateFirst,
      activateLast,
      isActive,
      // This is where the `storeId` gets passed down to
      // the `ListboxOption`s as well.
      storeId,
      ...,
    })

    return ...
  }
})
```

Inject references and methods from the component tree

`ListboxOption`, though, *does* access those methods and use them to implement the features I listed.

To get access, they first `inject` the items using—you guessed it—`inject`.

```
// Listbox.ts
...

const ListboxSymbol = Symbol('Listbox')

export const Listbox = defineComponent(...)

export const ListboxOption = defineComponent({
  ...,
  setup (props, { slots }) {
    const {
      activate,
      activatePrevious,
      activateNext,
      activateFirst,
      activateLast,
      isActive,
      storeId,
      ...,
    } = inject(ListboxSymbol)

    ...

    return ...
  }
})
```

Render another scoped slot

With those methods now injected, `ListboxOption` can use them inside its `setup` function (which creates the keyboard and mouse handlers), and expose them to its scoped slot.

```
// Listbox.ts
...
const ListboxSymbol = Symbol('Listbox')
export const Listbox = defineComponent(...)

export const ListboxOption = defineComponent({
  // Each `ListboxOption` accepts its option as a prop
  props: {
    option: { type: String },
  },
  setup (props, { slots }) {
    const {
      activate,
      activatePrevious,
      activateNext,
      activateFirst,
      activateLast,
      isActive,
      storeId,
      ...,
    } = inject(ListboxSymbol)

    // Here is where each `ListboxOption` generates its own
    // unique ID and uses the `storeId` method to pass the ID
    // back up to `Listbox` for future use.
    const id = 'compound-listbox-option-' + totalIds++
    storeId(props.option, id)



    ... // Code continues on the next page
```

```
    // `ListboxOption` returns a function that renders a scoped slot.
    return () => slots.default({
      bindings: {
        ...,
        // Activate the option when the end user hovers over it
        onMouseenter: () => activate(props.option),
        onKeydown: event => {
          switch (event.key) {
            case 'ArrowUp':
              event.preventDefault()
              // `cmd+up` activates the first option
              if (event.metaKey) { activateFirst(); break; }
              // `up` activates the previous option
              activatePrevious(props.option)
              break
            case 'ArrowDown':
              event.preventDefault()
              // `cmd+down` activates the last option
              if (event.metaKey) { activateLast(); break; }
              // `down` activates the next option
              activateNext(props.option)
              break
            ...
          }
        },
      },
      // Useful methods are exposed to the scoped slot
      // so that developers can conditionally add classes
      // or styles, or add buttons or additional keyboard event
      // handlers to activate different items.
      isActive: () => isActive(props.option),
      activatePrevious: () => activatePrevious(props.option),
      activateNext: () => activateNext(props.option),
    })
  }
})
```

And that's everything! With all of that code in place, the compound listbox is able to track which option is active, in a way that is fully compatible with assistive technology. All of the nitty gritty implementation details are hidden from developers who use the compound listbox.

This compound group only includes two components, `Listbox` and `ListboxOption`. For a larger group with more features, we would still follow the same pattern:

- Define reactive state and methods inside the root-level component
- Provide that data to the tree
- Inject that data inside other components
- Render a scoped slot for each component

Revisit compound component usage

Recall from Chapter 1 that this is how developers can make use of all the functionality around the active item:

```
<script setup>
import { Listbox, ListboxOption } from 'path/to/Listbox'
import { options } from 'path/to/options'
</script>

<template>
  <!-- In the `Listbox` scoped slot, we can access useful
  methods for activating different options programmatically. -->
  <Listbox
    :options="options"
    v-slot="{ bindings, active, activate, activateFirst, activateLast }"
  >
    <ul v-bind="bindings">
      <!-- In the `ListboxOption` scoped slot, we access the `isActive`
      function, which returns `true` for the active option. -->
      <ListboxOption
        v-for="option in options"
        :key="option"
        :option="option"
        v-slot="{ bindings, isActive, activatePrevious, activateNext }"
      >
        <!-- This element shows how we could use the `isActive`
        method to conditionally bind the `.active` class. -->
        <li v-bind="bindings" :class="{ active: isActive() }">
          <span>{{ option }}</span>
        </li>
      </ListboxOption>
    </ul>
  </Listbox>
</template>
```

Recall as well that you can find a full example styled with Tailwind on GitHub.

Let's refactor a compound component

For the refactor, we're going to write our `useListbox` composable in the following order:

1. Write boilerplate. This includes creating our function refs.
2. Initialize reactive references
3. Write methods that mutate our reactive state
4. Bind attributes to our DOM elements
5. Attach event listeners to our DOM elements
6. Return function refs, reactive references, and methods
7. Write a reimagined Vue template

We're only working on the `active` feature in this refactor, so we won't see the entire `useListbox` function come together, but we'll definitely touch on every major

Afterward, we'll close this chapter with some high-level analysis of the differences between the full compound listbox and function ref listbox codebases.

Write boilerplate, including function refs

Yes, composables do have their own boilerplate! It's pretty lightweight, though. In the function ref pattern, here's what I consider to be boilerplate code:

- Import statement, bringing in tools from the Vue Composition API
- Global counters for ID generation
- Export statement for the function
- Function ref setup
- The return value

This scaffolding is part of virtually every composable that follows the function ref pattern:

```
// useListbox.ts
// For now, we just need a couple tools from the Vue Composition API
import { ref, onBeforeUpdate } from 'vue'
// This is a global counter that we'll use later on when we generate IDs
// for the listbox options. You'll  often see a counter like this one,
// in composables that work with lists of elements.
let totalIds = 0

export function useListbox () {
  // To implement the active option feature, we'll need references to
  // the root listbox element and each element in the list of options.
  //
  // These references are each initialized with no value just like any
  // template ref you would create inside a component's `setup` function.
  const rootElement = ref()

  // For the root element, we need the simpler type of function ref,
  // that just accepts an element as its first argument and stores
  // that element in the reactive reference.
  const rootRef = (element: HTMLElement) => rootElement.value = element


  ... // Code continues on the next page
```

```
  // In our intended use case, the options elements will be  rendered by
  // v-for. We'll initialize a reactive array where we can store elements.
  const optionsElements = ref([])

  // Then, we'll create the more complex type of function ref, which
  // captures each element in a v-for list and stores it in our array.
  //
  // In this composable, we're using the function ref getter
  // technique mentioned earlier in this chapter. Our
  // `getOptionRef` function accepts an index, and returns the
  // correct function ref for each option in the list.
  const getOptionRef = (index: number) => (element: HTMLElement) => {
    optionsElements.value[index] = element
  }

  // Recall that we have to empty out the array of options before each
  // component update, to ensure we're not referencing stale data.
  onBeforeUpdate(() => {
    optionsElements.value = []
  })

  // Our core logic will go in the middle here
  ...

  // Our return statement will include other reactive
  // references and useful methods, but for now, we'll
  // just return our function refs.
  return {
    rootRef, getOptionRef,
    ...,
  }
}
```

Here's that same code without comments so you can get a better sense of the relatively small amount of boilerplate code required:

```
// useListbox.ts
import { ref, onBeforeUpdate } from 'vue'

let totalIds = 0

export function useListbox () {
  const rootElement = ref<HTMLElement>()
  const rootRef = (element: HTMLElement) => {
    rootElement.value = element
  }

  const optionsElements = ref<HTMLElement[]>([])
  const getOptionRef = (index: number) => (element: HTMLElement) => {
    optionsElements.value[index] = element
  }
  onBeforeUpdate(() => {
    optionsElements.value = []
  })

  ...

  return {
    rootRef, getOptionRef,
    ...,
  }
}
```

Initialize composable state

Recall that in the compound listbox, we initialized three pieces of state:

- `active`, a reactive reference to the currently active option (String)
- `ids`, a non-reactive object that stores options and their `id`s as key/value pairs, so that `Listbox` can look up any `id` when needed
- `ariaActivedescendant`, a reactive reference to the `id` of the active option

We're going to handle these items a bit differently inside our composable:

- `active` will be a reactive reference to the index (Number) of the currently active option
- `ids` will be a non-reactive array of unique IDs. The order of IDs in the array will exactly match the order of their corresponding elements in the `optionsElements` array (filled by our function ref).
- `ariaActivedescendant` won't change at all—it will still be a reactive reference to the `id` of the active option

This is a design choice that is explained in detail in Chapter 3, in the section titled "For rendered lists, track index-based positions, not raw data".

For now, don't think too hard about the reasoning behind this design choice. It makes our code a bit cleaner and less complex, but doesn't change the overall process of refactoring the compound component.

Here's what the code looks like to initialize our reactive references inside `useListbox`:

```
// useListbox.ts

// We add `shallowRef`, `computed`, and `onMounted` to our
// list of tools imported from the Vue Composition API.
import { ref, onBeforeUpdate, shallowRef, computed, onMounted } from 'vue'

// Still got our global ID counter! We'll make use of this below.
let totalIds = 0

export function useListbox () {
  // Function refs are still at the top of our composable
  const rootElement = ...
  const rootRef = ...
  const optionsElements = ...
  const getOptionRef = ...
  onBeforeUpdate(...)

  // We initialize the array of IDs
  const ids = shallowRef([])

  // After the component has been mounted, we can be confident
  // that the `optionsElements` array has been filled up
  // with all the DOM elements for the list of options.
  //
  // We can iterate over the elements to generate unique IDs.
  onMounted(() => {
    const generatedIds = optionsElements.value.map(() => {
      // For each element, we'll increment our global ID
      // counter to ensure that IDs are unique.
      return 'function-ref-listbox-option-' + totalIds++
    })

    // Finally, we assign the generated IDs to our `ids` array.
    ids.value = generatedIds
  })


  ... // Code continues on the next page
```

```
  // We initialize `active` as `0`, i.e. the index-based position
  // of the first option in the list.
  const active = ref(0)

  // The code for `ariaActivedescendant` is actually exactly the same as
  // it was in the compound listbox. The underlying difference is that
  // we're retrieving a unique ID based on its position in an array,
  // rather than retrieving a key/value pair from an object.
  const ariaActivedescendant = computed(() => {
    return ids.value[active.value]
  })

  return {
    rootRef, getOptionRef,
    // We'll add `active` to our return value. Developers can use
    // `watchEffect` to watch this reactive reference if they want to.
    active,
  }
}
```

Write methods

With our composable's state in place, we can add methods to work with it.

```
// useListbox.ts
import { ref, onBeforeUpdate, shallowRef, computed, onMounted } from 'vue'
let totalIds = 0
export function useListbox () {
  ...

  // Our new methods for working with `active` have changed
  // a bit, because `active` is now a number instead of a string.
  const activate = (index: number) => active.value = index

  const activatePrevious = (index: number) => {
    if (index === 0) return
    active.value = index - 1
  }

  const activateNext = (index: number) => {
    if (index === optionsElements.value.length - 1) return
    active.value = index + 1
  }

  const activateFirst = () => active.value = 0

  const activateLast = () => {
    active.value = optionsElements.value.length - 1
  }

  const isActive = (index: number) => index === active.value

  ...

  return {
    rootRef, getOptionRef, active,
    // We'll add all our new methods to the return value as well.
    activate, activatePrevious, activateNext,
    activateFirst, activateLast, isActive,
  }
}
```

Bind data to DOM attributes

In our composable, we need to bind data DOM attributes:

- The `id` of each option must be bound to the `id` property of its DOM element
- The `ariaActivedescendant`—i.e., the `id` of the active option—must be bound to the `aria-activedescendant` attribute of the root listbox element.

The `id` binding only needs to happen when the component is mounted and the elements aer first rendered. The `ariaActivedescendant` binding, on the other hand, needs to be updated each time the `active` option changes.

But inside of composition functions, we don't have access to the Vue template, nor do we have access to a render function. So how can we bind any data to DOM attributes, let alone keep those bindings up-to-date over time?

The answer: we can schedule a side effect to assign the IDs when the component is mounted, and we can schedule another side effect to update `aria-activedescendant` each time our `ariaActivedescendant` reference changes.

If you were writing this code manually, it would look something like this:

```
// useListbox.ts
// We would add `onMounted` and `watchPostEffect` to our imported tools.
import { ..., onMounted, watchPostEffect } from 'vue'
export function useListbox () {
  const rootElement = ...
  const optionsElements = ...
  const ids = shallowRef([])

  // When the component is mounted, and after the IDs have all been
  // generated, we want to assign the IDs to their corresponding DOM
  // elements. We can do that with a `for` loop in an `onMounted` hook.
  onMounted(() => {
    for (let i = 0; i < ids.value; i++) {
      optionsElements.value[i].id = ids.value[i]
    }
  })

  const active = ref(0)
  const ariaActivedescendant = computed(...)

  // When the component is mounted, we want to attach the value of
  // `ariaActivedescendant` to the root listbox element. We also want to
  // perform that side effect again every time `ariaActivedescendant` changes.
  // To do this, we can set up a watcher in an `onMounted` hook.
  //
  // Note: we're using `watchPostEffect` here instead of just
  // `watchEffect`. This is because `watchPostEffect` gives Vue the
  // time it needs to call each of our function refs and collect
  // elements from the DOM. Vue will only run our `watchPostEffect`
  // _after_ the DOM elements have been retrieved.
  onMounted(() => watchPostEffect(() => {
    // Vue will automatically run this `setAttribute` code each time
    // `ariaActivedescendant` changes. Vue is awesome.
    rootElement.value.setAttribute(
      'aria-activedescendant',
      ariaActivedescendant.value
    )
  }))
  return ...
}
```

There's a significant amount of complexity and repetition in that code:

- Your composable gets littered with `onMounted` hooks
- You need to remember to use `watchPostEffect` instead of `watchEffect`, otherwise you'll see all kinds of hard-to-debug errors
- Nested callbacks and `for` loops quickly get confusing

Instead of writing this code manually, I recommend you `import a tool` from Baleada Features.

Baleada Features exports a function called `bind`, which is designed to be the composable version of `v-bind`. In other words, `bind` lets you use `v-bind` functionality from inside a composable.

On the next page, let's import the `bind` function from Baleada Features, and see how it works.

```
// useListbox.ts
import { ... } from 'vue'
import { bind } from '@baleada/vue-features'
...
export function useListbox () {
  const rootElement = ...
  const optionsElements = ...
  const ids = shallowRef([]) ...

  // We'll remove our `onMounted` hook with its nested
  // for loop, and we'll replace it with `bind`.
  bind({
    // We'll tell `bind` that we want to bind something to each element in
    // `optionsElements`. `bind` already knows that it should only try to
    // bind data after the component is mounted.
    element: optionsElements,

    // We pass an object with key/value pairs describing the attributes or
    // properties we want to bind data to, and what data we want to bind.
    values: {
      // In this case, we want to bind to the `id` property. We also want
      // to bind a different value for each item in the array.
      //
      // `bind` allows us to pass a callback function to make that happen.
      //
      // From the callback function's first and only argument, we can
      // retrieve the `index` of the element we're currently binding to.
      id: ({ index }) => {
        // Our callback function should return the value that
        // we want to bind.
        return ids.value[index]
      }
    }
  })




  ... // Code continues on the next page
```

```
  const active = ref(0)
  const ariaActivedescendant = computed(...)

  // We'll also remove the other `onMounted` hook and its
  // nested `watchPostEffect` and replace them with `bind`.
  bind({
    // In this case, we're binding a value to the root element.
    element: rootElement,
    values: {
      // We want to bind to the aria-activedescendant attribute.
      //
      // `bind` allows us to camelCase attribute names. It also
      // understands that if we pass a `computed` reference, it
      // should set up a `watchPostEffect` to re-assign the new
      // value each time it changes.
      //
      // So, we can just destructure our `ariaActivedescendant`
      // reference right in:
      ariaActivedescendant,
    }
  })
  ...

  return ...
}
```

Note that `bind` can handle individual elements and arrays of elements. Also note that there are several different things you can pass as `values`:

- If you just want to bind a static piece of data one time, when the component is mounted, you can pass a string, number, or boolean.
- If you want `bind` to reactively update the DOM each time a reactive reference (created by `ref` or `computed`) changes, pass the reactive reference directly.
- If you want to bind data to each element in an array, but the value is different for each element, you can pass a callback function that receives the `index` of the element and returns the correct value.

The `active` feature of `useListbox` gives us a nice glimpse of what `bind` is capable of, but `bind` can do a lot more! To see all possibilities, and examples of different use cases, check out the full documentation for `bind`.

Attach event listeners to our DOM elements

To make our `active` feature work properly, we need to attach a few even listeners:

- `mouseenter` should active the option that was entered
- `up` arrow should activate the previous option
- `down` arrow should activate the next option
- `cmd+up` should activate the first option
- `cmd+down` should activate the last option

In composables, event listeners suffer from the same problem as attribute bindings: if we don't have access to a Vue template or a render function, how can we attach event listeners to our elements?

The answer once again is to schedule side effects to run when the component is mounted.

To do this manually, we would get inside an `onMounted` hook and loop over each element in our `optionsElements` array. Then, we would use an `onBeforeUnmount` hook to remove each event listener, so that our composable doesn't cause any memory leaks.

I'll spare you that code snippet, though, because there's a better solution: Baleada Features also exports a function called `on`, which is designed to be the composable version of `v-on`. `on` lets you use `v-on` functionality from inside a composable.

On the next page, we'll import `on` and watch the magic happen.

```ts
// useListbox.ts
import { ... } from 'vue'
import { bind, on } from '@baleada/vue-features'
...
export function useListbox () {
  const rootElement = ...
  const optionsElements = ...

  ...

  on({
    // Just like `bind`, `on` requires you to pass a reactive reference
    // to either a single element or an array of elements. In this case,
    // we want to add event handlers to each  option in the list, so we
    // pass `optionsElements`.
    element: optionsElements,
    // `bind` requires you to pass a `values` object, but `on` requires an
    // `effects` object. The `effects` object contains key/value pairs
    // that describe the events you want to listen for, and the side effects
    // you want to run when the events happen.
    effects: {
      // `on` allows you to pass keyboard shortcuts as event names.
      // Internally, it knows that it should listen for the `keydown` event
      // and check that all the keys in the combination were pressed.
      //
      // Just like any normal keydown event handler, our callback receives
      // the KeyboardEvent as the first and only argument.
      'cmd+down': event => {
        event.preventDefault()

        // When the end user presses cmd+down, we activate
        // the last option.
        activateLast()
      },
      'cmd+up': event => {
        event.preventDefault()

        // When the end user presses cmd+up, we activate
        // the first option.
        activateFirst()
      },

      // Code continues on the next page
```

```
// When the mouse enters an option, we want to activate
// that specific option. To accomplish that, we'll pass
// an object here, instead of a normal MouseEvent listener.
//
// Our object can have a `createEffect` key, which contains
// a callback function that will create and return our
// MouseEvent listener.
mouseenter: {
  // The `createEffect` callback receives one argument,
  // which includes the `index` of the option that
  // detected the `mouseenter` event.
  createEffect: ({ index }) => {
    // Here, we'll create the event handler
    function handleMouseenter (event) {

      // Inside the event handler, we use that `index` to
      // activate the exact option that detected the
      // `mouseenter`.
      activate(index)
    }

    // We return the event handler. Internally, `on` will
    // loop over our `optionsElements`, creating and
    // attaching a different event handler to each element.
    return handleMouseenter
  }
},

// For the up and down arrow keys, we take the same approach as
// we did with `mouseenter`, creating a different event handler for
// each element in `optionsElements`.
//
// Here, they're written with arrow functions to be more concise.
down: {
  createEffect: ({ index }) => event => {
    event.preventDefault()

    // On arrow down, activate the next option
    activateNext(index)
  }
},


// Code continues on the next page
```

```
        up: {
          createEffect: ({ index }) => event => {
            event.preventDefault()

            // On arrow up, activate the previous option
            activatePrevious(index)
          }
        },
      }
    })

    ...

    return ...
  }
```

Just like `bind`, `on` can handle individual elements and arrays of elements, and it accepts a couple different things as `effects`:

- Normal event handlers
- Objects with a `createEffect` callback that creates a slightly different event handler for each element in an array

`on` can also understand a lot of different types of events:

- It understands normal DOM events, like `mouseenter` or `keydown`
- It understands keyboard shortcuts, like `cmd+down` or `ctrl+alt+delete`
- It understands complex things like `resize` (for setting up a `ResizeObserver`) or even `(prefers-color-scheme: dark)` (for setting up a `MediaQueryList`)

To see all possibilities, and examples of different use cases, check out the full documentation for `on`.

Write a reimagined Vue template

Now that we've coded our function refs, reactive references, methods, attribute bindings, event listeners, and return statement, we're ready to wire `useListbox` up with a Vue template.

Note that the markup below only shows the code related to the active option feature, but you can see a full example styled with Tailwind on GitHub.

```
<script setup>
import { useListbox } from 'path/to/useListbox'
import { options } from 'path/to/options

// Set up all listbox state management and methods. We
// can destructure lots of useful things here.
const {
  // First, we'll grab the function refs we need to attach
  // to our template.
  rootRef, getOptionsRef


  // We're not going to use these methods for programmatically
  /// activating specific options, but I just wanted to point
  // out that we can destructure them all from the return value.
  activate, activateFirst, activateLast,
  activatePrevious, activateNext,

  // We'll use the `isActive` method to do some conditional
  // class binding.
  isActive,
} = useListbox()
</script>
```

Code continues on the next page

```vue
<template>
  <!-- First, we bind the root element's function ref. -->
  <ul :ref="rootRef">
    <!--
      We use `v-for` to get access to the index of each element.

      We pass that index into our `getOptionRef` function ref
      getter, which returns the correct function ref for each
      element in the list.

      We also conditionally bind the `.active` class, only for
      the active element.
    -->
    <li
      v-for="(option, index) in options"
      :key="option"
      :ref="listbox.getOptionRef(index)"
      :class="{ active: isActive(index) }"
    >
      <span>{{ option }}</span>
      ...
    </li>
  </ul>
</template>
```

Some really important things to notice in our reimagined template:

- As developers using this composable, we no longer have to wade through nested scoped slots to get access to useful state and methods. Instead, we access all of it in one place (`useListbox`'s return value), and it's available to our `setup` function as well as our entire component.
- Since we're not nesting markup inside of renderless components, our indentation stays low, and our markup is more readable.
- We still have complete control over markup, styles, and classes. It's all readily visible and editable.
- Assuming `useListbox` has documentation that tells us where and how to bind `rootRef` and `getOptionRef`, we don't even need to be aware that function refs exist, let alone understand how they work. We just need to bind things to `ref` attributes as instructed.
- It feels like basic Vue. There are no advanced Vue features in sight—everything is tucked away inside `useListbox`.
- It works. All of our active option features work perfectly, enhancing our markup to make it compatible with user needs and assistive tech.

In my opinion, it's safe to say that our function ref composable improved developer experience for any developer who uses it in their app.

But what about us, the authors of `useListbox`? What did we gain from refactoring our compound listbox into a function ref composables?

To close out this chapter, let's answer that question.

Why refactor to function ref composables?

Before we dive in, recall that you can see the full source code of the compound listbox and the function ref composable listbox on GitHub:

- `Listbox` group
- `useListbox`

That said, we're not going to look too closely at any more code. For the rest of this chapter, we're going to focus on a high-level analysis of the differences between the full compound listbox and function ref listbox codebases.

Let's start with the basics: how much code does each pattern require?

Here's some data on how many `lines` and how many `characters` each codebase uses:

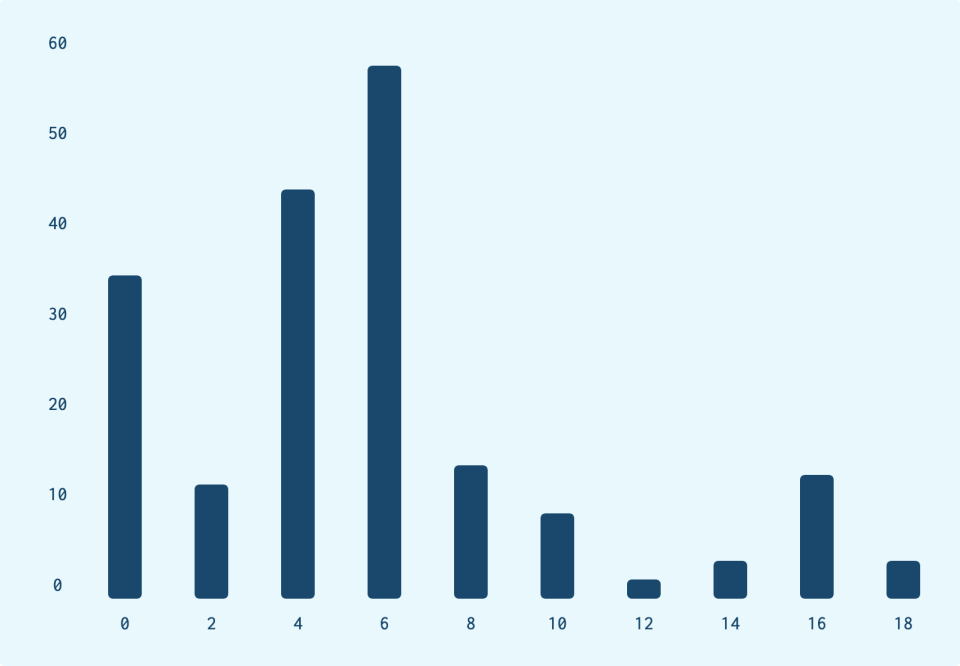
CODE	LINE COUNT	CHARACTER COUNT
Compound component	191	4,804
Characters	179	3,693

The compound component only has 12 more lines than the function ref composable, but it has `well over 1,000` more characters.

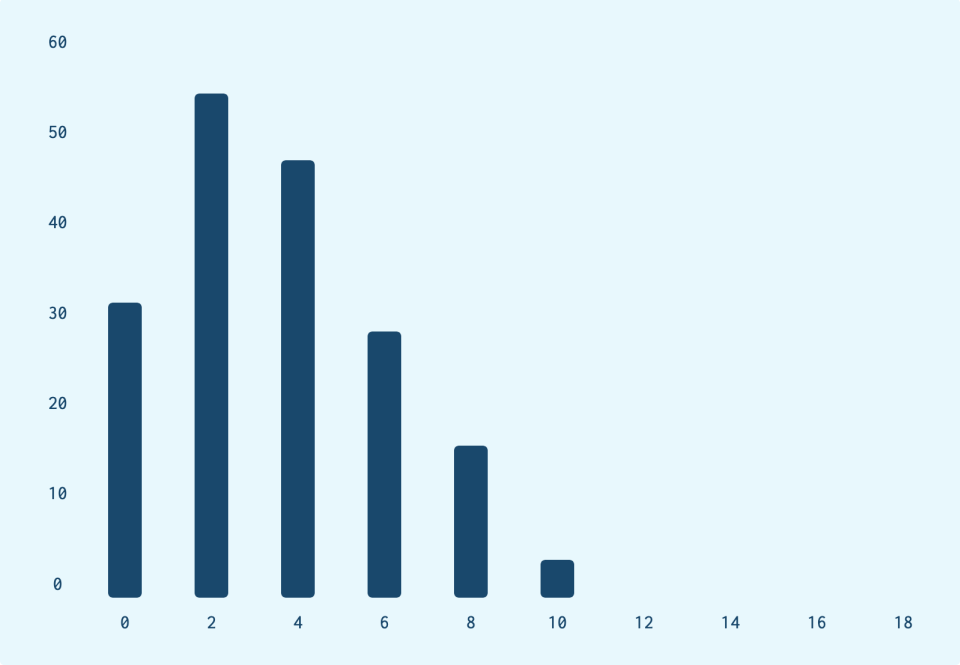
The feature-related code in these two codebases is virtually identical, so it appears that most of the extra characters are coming from `increased indentation`.

Below are two histograms, visualizing the indentation of each codebase. The x-axis shows the amount of indentation, measured as the number of spaces, and the y-axis shows how many lines of code share that number of spaces.

Here's the histogram for the `compound component`:



And here's the histogram for the `function ref composable`:



These visualizations show that many lines of code in the compound component are heavily indented, while the function ref composable stays pretty flat.

In my opinion, these differences are interesting, but they don't have a huge impact on productivity.

The three differences that `do` impact productivity are the three points we've been discussing throughout the book:

- Function ref composables reduce boilerplate
- Function ref composables drastically simplify component communication
- Function ref composables cleanly collocate tightly coupled logic

Function ref composables reduce boilerplate

Most compound components that I've seen are written using the Vue Options API. They include lengthy `props` definitions, and boilerplate code for `data`, `computed`, `methods`, etc. that organizes code into different categories, but doesn't add meaningful functionality.

By writing our compound listbox using a `setup` function and the Composition API, I gave my best effort to reduce all that boilerplate code. Even so, the compound listbox includes `55` lines of boilerplate, while the function ref composable only includes `27`.

That's about twice as much!

And keep in mind that our compound listbox only contains two components. Larger, more complex groups usually include more than three components, making you repeat the same boilerplate component definitions each time.

Also keep in mind that the largest chunk of boilerplate code in `useListbox` is responsible for defining function refs. In any real project where you're writing function ref composables, that code would be extracted into utility functions. Using those utility functions, you'd remove about 15 lines of code from `useListbox`, and replace them with just 2 lines. (Chapter 3 has some more advice about how to do that.)

In my experience, even complex function ref composables max out at around 20-30 lines of boilerplate code. Exporting a single JavaScript function simply requires less code.

Function ref composables drastically simplify component communication

For me, this benefit can't be overstated, especially for larger compound groups. It's easily my biggest motivation to write function ref composables.

Inside the function ref composable, there is zero use of `provide` & `inject`.

There's no concept of component communication, because there are no components—there's just reactive references, methods, and side effects, all working within the scope of the same JavaScript function. Every variable is accessible from everywhere.

Compound components, on the other hand, use `provide` & `inject` as the backbone of pretty much all their functionality.

When you write code for a feature inside a function ref composable, you just need to focus on the code that makes that feature work. When you write the same feature inside a compound component group with 3+ components, you take on a lot of cognitive load just to figure out how to transport variables where they need to go. You can only work on feature-related code *after* you build all those communication pathways.

For TypeScript users, this gap is even larger. Type-checking works seamlessly with the function ref pattern, because every variable is in scope.

In a compound component, though, `provide` & `inject` instantly destroy type safety.

I'll concede that both `provide` and `inject` accept generic types, so you can fix type safety by defining types manually outside of your components.

This isn't a great workflow though—it means you have to define types pretty far away from the code that is actually being type-checked. With function ref composables, you can type-hint your code exactly where you write it, and type-checking will work everywhere else.

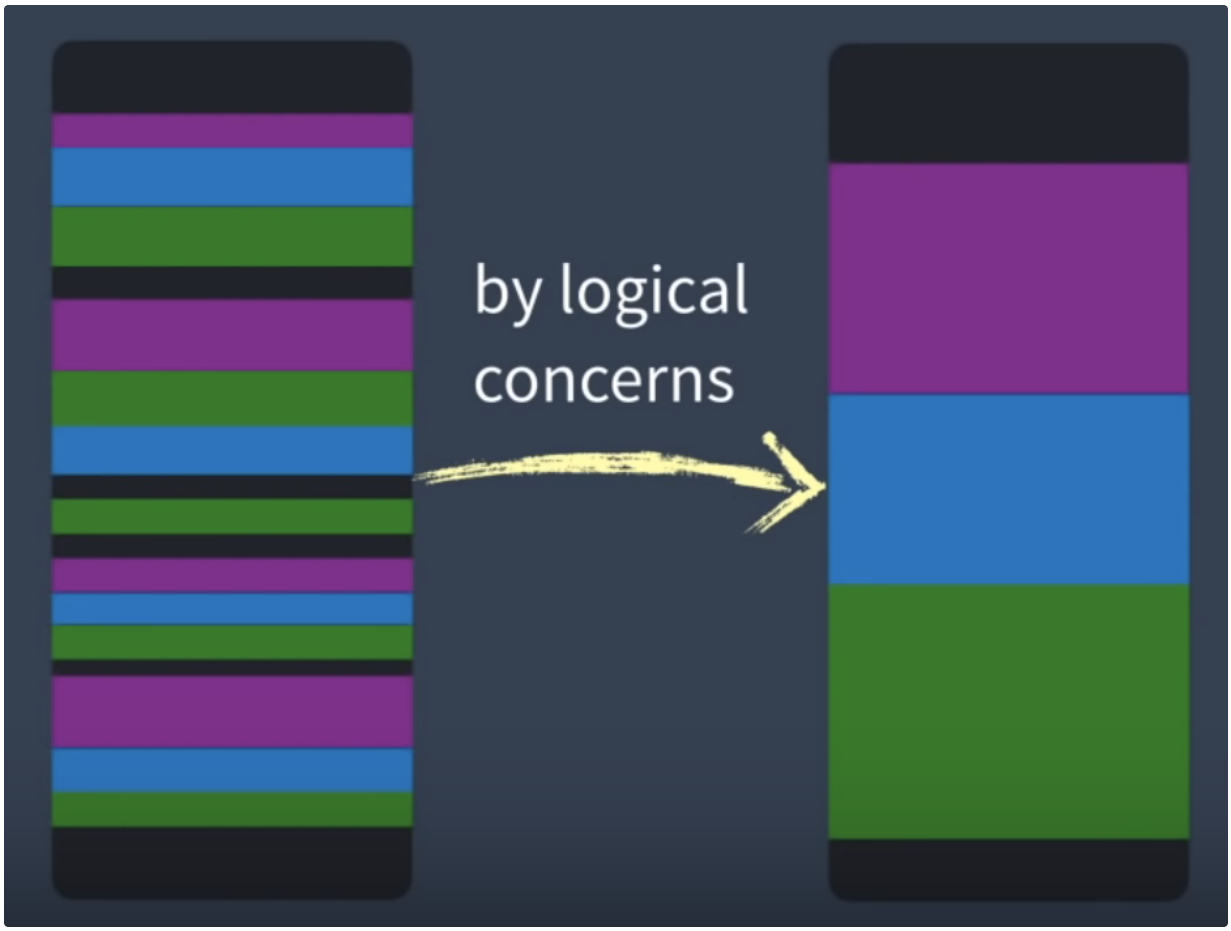
Function ref composables cleanly collocate tightly coupled logic

Simplified component communication might be my biggest motivation to write function ref composables, but code organization is the most fun to analyze!

When Vue 3 launched, the Vue team made us a promise: Vue 3 will make it easier to organize your code by logical concern, rather than by Vue component options.

That sounds great, and it looks even better—I loved the Vue team's eye-catching illustrations of color-coded lines of code, all mixed up in the Options API, then in perfect organization in the Composition API.

Take a look at the next page to see an example:



In the next chapter, I included a section on what it means to organize code by logical concern, why that's desirable, and how to actually do it in practice.

In this chapter, though, I don't want to get into the weeds. I just want to show that compound components, even when they're written with `setup` functions and the Composition API, suffer from disorganized code.

To prove it, I went through each line of code in the `Listbox` group and each line of code in `useListbox`, color-coding each one according to which logical concern it belongs to.

The results are striking.

Despite my best efforts, the compound listbox splits up tightly coupled code. It distributes related code across several components, interwoven with `provide`/`inject` calls and render functions.

In contrast, it was super easy to organize code in `useListbox`. Since everything is inside the same function scope, everything can be freely moved around wherever you want it.

In my experience writing function ref composables that are quite a bit more complex and powerful than the one we've studied in this book, that trend holds.

Okay, let's wrap this up.

Function ref composables reduce boilerplate, drastically simplify component communication, and cleanly collocate tightly coupled logic.

And so, after two probably-too-long chapters of analysis, my advice to you is: write function ref composables.

> `Write function ref composables!`

How do I take it to the next level?

Chapter 3, Authoring the function ref pattern, is less of a structured story, and more of reference for problems I've encountered and solved in function ref composables.

Feel free to dive into Chapter 3 right now, or go write a function ref composable of your own, and return to Chapter 3 if you run into a problem you're not sure how to solve.

Enjoy!

Authoring the function ref pattern

Chapter summary

Until you've trained your brain to follow the function ref pattern, you might be unsure of how to organize code or solve certain problems. This chapter is an unstructured reference for a lot of those questions.

As mentioned in Chapter 2, this final chapter is less of a structured story, and more of reference for problems I've encountered and solved in function ref composables.

Feel free to dive right in, or go write a function ref composable of your own, and come back here if you run into a problem you're not sure how to solve.

Understand the function ref core concepts

A few times in this book, I've mentioned that a downside of compound components is the need to master three niche Vue features:

- Scoped slots
- Render functions
- `provide` and `inject`

As we refactor compound components to the function ref pattern, we remove the need to learn those features, but we take on the responsibility of learning an equally niche feature: function refs.

Everything is a tradeoff! For me, it's been worthwhile to learn function refs, in order to avoid using the other three niche features, and gain all the other benefits of the function ref pattern.

Understanding the function ref core concepts is absolutely necessary if you want to be successful with this pattern. Chapter 2 explains function refs in a fair amount of depth, but to give even more guidance, I recorded a three-part screencast series on function refs.

Those screencasts are published in a playlist on my YouTube channel.

Start with a Vue component

Before I start writing any composable code, I like to write or at least imagine a Vue template and a `setup` function, showing how I would like the composable to look and feel inside the components where I'll actually be using it.

When writing the template, I usually skim through popular component library examples and relevant WAI-ARIA specs.

I don't think deeply about logic, roles, attributes, styling, or keyboard accessibility at this stage—I'm just collecting ideas for what semantic markup might be necessary to make a particular feature work, and where I'll have to attach function refs to get my DOM side effects wired up.

In the `setup` function, I think about what data and context would be available to me in a real app, and how I can comfortably translate that into what I suspect will be the parameters for the composable.

The goal is to create the simplest possible, unstyled version of a Vue component that would consume your composable.

Reference this outline while you code your composable, and definitely don't throw it away—it's perfect material for usage examples in documentation.

On the next page is my simplest-possible Vue component sketch for `useTablist`, which I eventually included in the `useTablist` docs:

```vue
<!-- CustomTablist.vue -->
<script setup>
import { ref, computed, readonly } from 'vue'
import { useTablist } from '@baleada/vue-features'

const metadata = ref([
  {
    tab: 'Educate Girls',
    panel: 'https://www.educategirls.ngo/'
  },
  {
    tab: 'Kheyti',
    panel: 'https://www.kheyti.com/'
  },
  {
    tab: 'One Heart Worldwide',
    panel: 'https://oneheartworldwide.org/'
  },
])

const tablist = readonly(
  useTablist()
)
</script>

<template>
  <div :ref="tablist.label.ref">My Tablist</div>
  <div :ref="tablist.root.ref">
    <div
      v-for="({ tab }, index) in metadata"
      :key="tab"
      :ref="tablist.tabs.getRef(index)"
    >
      {{ tab }}
    </div>
    <div
      v-for="({ tab, panel }, index) in metadata"
      :key="tab"
      :ref="tablist.panels.getRef(index)"
    >
      {{ panel }}
    </div>
  </div>
</template>
```

Organize code by logical concern

One of the biggest benefits of the composition API is that it gives you the ability to `organize code by logical concern`.

I touched on this concept in Chapter 2, but it's worth exploring more deeply here. Let's talk about `what` it means to organize code by logical concern, `why` it's worth the effort, and `how` we can go about it.

First, the "what". In my mind, the options from the Options API have one important thing in common with logical concerns: they are `categories` for your code. The difference is in what those categories represent.

Options are categories that describe what your code `is`. Got some reactive references? Put them in the `data` category. Memoized values go in the `computed` category. Functions that mutate reactive data go in `methods`, and so on and so forth. Code is organized based on what it is.

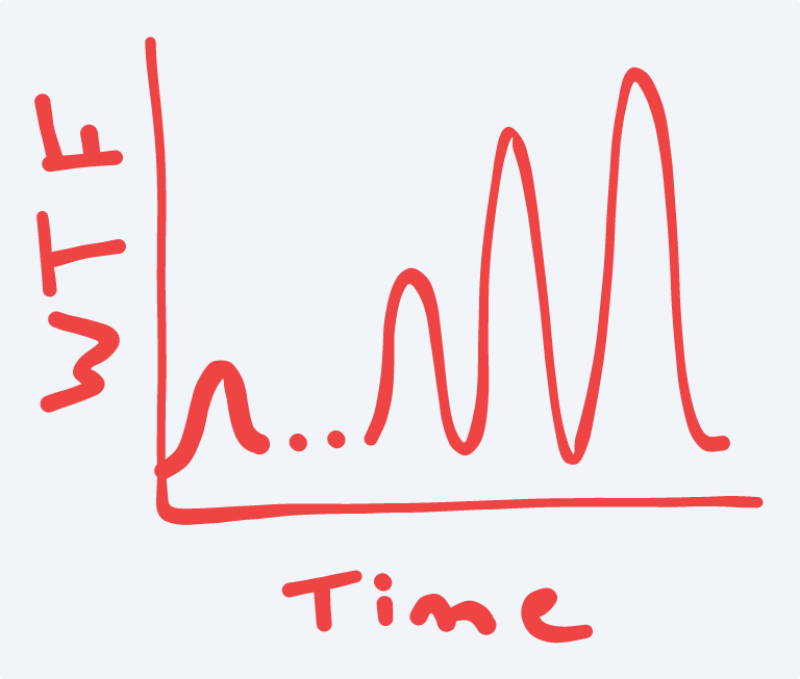
Logical concerns, on the other hand, are categories that describe what code `does`. In our `useListbox` composable in Chapter 2, there are several things the composable is supposed to do:

- Track and control the active option
- Track and control the selected option
- Manage focus
- etc.

Reactive references, methods, watchers, and lifecycle hooks are sprinkled throughout those categories inside `useListbox`. Code is organized by what it does, not by what it is.

Why is it worth the effort to organize code this way? I'll give a tongue-in-cheek answer to that question by examining how many times I tend to ask "Where the F  is that code?" while I'm authoring a composable.

Let's look at some beautiful and highly scientific graphs that show the results. Organizing code by the categories in the Options API yields the results in this graph:



Note that there are fewer WTFs initially, because it's usually easy to identify whether an individual piece of code is a reactive reference, a computed variable, a method, etc.

In my experience, though, the WTF count grows steadily over time. Each time I came back to a complex component or composable to refactor or fix a bug, I would find myself scrolling all over the place, hunting and pecking for the right places to change code. Adding new features involves splitting up your new code and weaving it into your existing categories.

For smaller codebases, it's not the end of the world, but large codebases can get very unwieldy, very quickly.

Contrast that with the graph of WTFs over time for code organized by logical concern.

There are plenty of WTFs initially, while I'm still determining exactly what a composable is supposed to do, and all the features it's going to have.

Eventually, the code settles into distinct categories, based on what it does. Over time, the WTF count stays low, because when I'm revising an individual feature, I know that all the code for that feature is in one place. To add a new feature, I just add a new section at the bottom of the composable.

So, now that I've dazzled you with my science, I'm sure you're wondering how you can organize code by logical concern.

The best advice I can give you is to go check out the talk I gave on this topic at VueConf Toronto 2021.

As I'm writing this book, I don't have a link to the official VueConf recording, so here's a link to the unlisted pre-recorded version on YouTube. Visit VueConf Toronto's channel and look for the 2021 playlist to see the full version with Q&A.

In case you're reading this offline, here are a few tactics to help you organize code by logical concern:

- Often, each `data` and `computed` property is the foundation of a separate logical concern. Create a section for each one.
- Within sections, start with reactive state, then write methods, then write side effects.
- Avoid the temptation to mix concerns inside watchers, lifecycle hooks, and event handlers. Always prefer to create new hooks in the section of code where they belong.
- Use functions like `bind`, `on`, `show`, and `model` from Baleada Features to organize template-related code by logical concern.
- You'll eventually find code that impacts multiple logical concerns. Put that code in a new "Multiple Concerns" section, and b in the habit of double-checking that section after you make changes to another section.
- Be consistent.

Within logical concerns, write code in a consistent order

With the Composition API, we have complete freedom to define tightly coupled reactive references, methods, and side effects, all within the same section of code. This is in contrast to the Options API, which forces us to split up that related code across different options.

This freedom has its downsides, though. When you're first getting started with function ref composables, it can be difficult to know where exactly you should put your code.

As we discussed in the previous section, I recommend you organize code by logical concern. Within each logical concern, I like to follow a consistent order as well:

1. Write a or imagine what the Vue template should look like (as explained in the previous section)
2. Write boilerplate. This includes creating function refs.
3. Initialize reactive references
4. Write methods that mutate our reactive state
5. Bind attributes to our DOM elements
6. Attach event listeners to our DOM elements
7. Add any new function refs, reactive references, and methods to the return value

You might notice this is the exact order we followed in Chapter 2 when we wrote the active option feature inside our `useListbox` composable. That was intentional!

The reason I like this workflow is because code complexity increases gradually.

Reactive references are the least complex, and they provide the foundation for the rest of your code.

As you write methods, you'll start to read from and write to your reactive references in more complex ways, and you'll probably uncover any edge cases that need to be handled.

Setting up side effects, including watchers, attribute bindings, and event listeners, is always the most complex code, because it calls your methods with specific timings.

Gradually increasing the complexity of my code has proven to be a good strategy in my function ref composables. As an added benefit, the consistency lets me navigate my codebases extremely quickly, even when I'm digging through the implementation details of a complicated features.

Understand effect timing in Vue 3

If your reactive side effects aren't properly timed in a Vue app, you'll see very confusing behavior.

To fully understand Vue effect timing, you'd have to learn about microtasks in browser-based JavaScript. But a deep understanding of microtasks and of the browser event loop is really only practical for framework authors—it's not practical knowledge for framework users.

Instead of trying to learn how Vue effect timing actually works under the hood, try learning the following:

- Simplified versions of effect timing concepts
- Some opinionated guidelines on how to use the two effect timing tools at your disposal in Vue 3: the `flush` option for `watch` and `watchEffect`, and `nextTick`.

Here are some simplified effect timing concepts:

- `flush: 'sync'` runs effects right away
- `flush: 'pre'` runs effects asynchronously, always before the DOM is updated
- `flush: 'post'` runs effects asynchronously, always after the DOM is updated, but always before the browser actually lays out and repaints your web app based on your DOM changes
- `nextTick` runs effects asynchronously, always after the DOM is updated, but not always before the browser actually lays out and repaints your web app based on your DOM changes

To see how effects run exactly in that order, check out this demo on the Vue SFC playground.

In practice, there are some good rules of thumb you can follow.

First, use `flush: 'pre'` if your side effect does not access or change a DOM element that has reactive bindings, or a DOM element rendered by `v-for`. (This is the default `watch` and `watchEffect` setting.)

```
// Example code for a reactive tablist component
const selectedTabIndex = ref(0),
      selectedTabPanelIndex = ref(0)

// No DOM changes or DOM access here, so `flush 'pre'` is fine
if (props.selectsPanelWhenTabIsFocused) {
  watch(
    selectedTabIndex,
    () => selectedTabPanelIndex.value = selectedTabIndex.value,
  )
}
```

Second, use `flush: 'post'` if your side effect does access or change a DOM element that has reactive bindings, or a DOM element rendered by `v-for`.

See the code snippet on the next page for an example.

```
// Example code for a selectable HTML input
const selection = ref({ start: 0, end: 0, direction: 'none' }),
      element = ref(null)

// This effect code *does* update a DOM element, so
// `flush 'post'` is needed.
//
// Note that `watch` won't run the side effect right away.
// It will wait for one of the dependencies to change. If
// you want to the run the effect right away, you can use
// `watchPostEffect`.
watch(
  selection,
  () => {
    element.value.setSelectionRange(
      selection.start,
      selection.end,
      selection.direction,
    )
  },
  { flush: 'post' }
)
```

Third, avoid `nextTick`. I've definitely found appropriate use cases for `nextTick` in the past, but they're extremely few and far between.

Fourth, avoid `flush: 'sync'`. I'm sure there's a use case for `flush: 'sync'`, but I haven't found it yet!

Finally, the most important rule: if you see unexpected behavior, break one of the previous rules to troubleshoot.

Abstract your function ref creation

You'll be creating *a lot* of function refs. I highly recommend abstracting a function that creates them more easily.

In Baleada Features, for example, I have a `useElementApi` function that I can call to create objects that include function refs.

```
// Inside my Baleada Features project
import { useElementApi } from 'path/to/useElementApi'

// Use it to work with individual elements
const single = useElementApi()
single.ref // A function ref for a single element
single.element // A reactive reference to the actual element

// Or, use it to work with multiple elements, usually
// rendered by v-for.
const multiple = useElementApi({ multiple: true })
multiple.getRef // A function that accepts an index and
                // returns a function ref.
multiple.elements // A reactive reference to an array of elements
                  // captured by the function refs.
```

As I write this book, `useElementApi` is not currently a publicly exported function from Baleada Features, because it's designed pretty specifically to work with the kinds of composables I write in that package.

But, if you want to look at the `useElementApi` source code for inspiration on how to abstract function ref creation, without losing type safety, you can find the source code on GitHub.

Abstract your ID generation

All kinds of interfaces require IDs to work properly with assistive tech:

- Each tab in a tablist should have its `aria-controls` attribute set to the ID of the corresponding tab panel
- The root element of a listbox should have its `aria-activedescendant` attribute set to the ID of the focused option
- The `for` attribute of an input's label should be set to the ID of the input

In almost every case, the best way to meet these requirements is to generate your IDs programmatically, so you can access them whenever you need, and you can guarantee IDs will be unique across the entire application. I've seen several open source projects abstract this kind of ID generation into functions like the one below:

```
let totalIds = 0

export function generateId () {
  return `my-unique-project-name-${totalIds++}`
}
```

This works perfectly well, and will save you the hassle of fiddling with string templates and global counters each time you need to generate IDs inside a composable.

Be aware that IDs tend to regenerate when components re-render, but as long as every generated ID is referencing the same global counter, this won't cause any problems.

In Baleada Features, I prefer to generate my IDs with a dedicated tool like `nanoid`, and to store generated IDs for the entire lifecycle of a component. You can see the source code on GitHub.

For rendered lists, track index-based positions, not raw data

As we've seen a few times in this book, function refs can be used to capture elements rendered by `v-for`.

```
<script setup>
import { ref, onBeforeUpdate } from 'vue'

const elements = ref([])

const getFunctionRef = index => element => {
  elements.value[index] = element
}

onBeforeUpdate(() => elements.value = [])

const list = ref([...]) // A list of items to iterate over
</script>

<template>
  <div
    v-for="(item, index) in list"
    :key="item"
    :ref="getFunctionRef(index)"
  >
    {{ item }}
  </div>
</template>
```

`v-for` will call a function ref for each element in the list, storing it in your reactive array. The interesting fact to be aware of is that elements are stored in the `exact same order` that they render in the DOM, which is also the `exact same order` as the items in the original list.

This fact allows us to simplify the reactive state we track inside our composables. To explain that more deeply, let's examine `useListbox`.

Note how `useListbox` doesn't accept the listbox options as a parameter:

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

const options = ref([...])
const listbox = useListbox()
</script>

<template>
  <ul :ref="listbox.rootRef">
    <li
      v-for="(option, index) in options"
      :key="option"
      :ref="listbox.getOptionRef(index)"
    >
      ...
    </li>
  </ul>
</template>
```

So if we don't pass the options into `useListbox`, how does it track which option is selected?

Keep in mind that `useListbox` has access to all the `<li>` elements after the component is mounted. In theory, we could read the `textContent` of each `<li>` to capture the options.

But what if the options are complex objects instead of just strings? What if each option in the array has a `title` and `description` property, which we render inside an `<h2>` tag and a `<p>` tag inside each list item? In that case, the `title` is probably the only text we care about, so we can't grab the full `textContent`.

The trick here is that `useListbox` doesn't care which option is selected. It doesn't track option content at all. Instead, it tracks the index-based position of the selected option.

Here's what the core code inside of `useListbox` looks like for storing, reading, and writing the selected option:

```
// useListbox.ts
export function useListbox () {
  ...

  // We're only tracking the index of the selected option.
  // Inside this function, we don't care about the actual
  // value of that option.
  const selected = ref(0)

  const select = (index: number) => {
    selected.value = index
  }

  const isSelected = (index: number) => {
    return index === selected.value
  }

  ...

  // We return the reactive reference and methods
  return {
    selected, select, isSelected,
    ...
  }
}
```

This approach drastically simplifies the logic inside our composable until it's just value assignments, with a little arithmetic here and there. The `active` option is tracked in the same simplified way.

Also, this approach still gives the developer using our composable everything they need to know. They know the index of the selected option, and they can set up a watcher to handle changes.

```vue
<!-- CustomListbox.vue -->
<script setup>
import { ref, watchEffect, onMounted } from 'vue'
import { useListbox } from 'path/to/useListbox'
import { updateDatabase } from 'path/to/updateDatabase'

const options = ref([
  { title: 'One', description: 'This is the first option.' }
  { title: 'Two', description: 'This is the second option.' }
  { title: 'Three', description: 'This is the third option.' }
])

const listbox = useListbox()

// Watch for changes to the listbox.selected index (Number),
// so we can update the database with the new value.
watch(
  listbox.selected,
  () => {
    // Use the index to retrieve the actual selected option
    const selectedOption = options.value[listbox.selected.value]

    // Only store the relevant data from selected option
    updateDatabase(selectedOption.title)
  }
)
</script>

<template>
  <ul :ref="listbox.rootRef">
    <li
      v-for="(option, index) in options"
      :key="option"
      :ref="listbox.getOptionRef(index)"
    >
      <!--
        We can render simple or complex options, without
        introducing any more complexity to `useListbox`.
      -->
      <h2>{{ option.title }}</h2>
      <p>{{ option.description }}</p>
    </li>
  </ul>
</template>
```

In Baleada Features, I use this technique for every single interface that deals with rendered lists of items.

Note that if the original list changes in length or order during user interaction, `v-for` will rebuild the reactive array of elements inside your composable.

In those cases, I can guarantee that your composable will always have the correct number and order of rendered elements, but be prepared to do some arithmetic to keep your index-based positions up to date.

Identify and abstract common logic

As you get more experience with composables, you'll start to notice recurring patterns in your UI logic.

Consider the following examples of logic:

- When the end user is focused on the list in a listbox, the up and down arrow keys should update the active item to the previous or next item respectively. When you pass the first or last option, loop around to the other end of the list.
- When the end user is focused on the tabs in a tablist, the right and left arrow keys should cycle the active tab panel. When you pass the first or last tab, loop around to the other end of the list.
- When the end user is focused on a grid or editable table, they should be able to use all four arrow keys to navigate to different cells.

The common denominator in all of these examples is `list navigation`.

In pretty much every keyboard accessible list—tablists, listboxes, grids, tables, menus, etc.—arrow keys change the active item, selected tab, focused gridcell, etc.

I quickly got tired of writing that "if we're at the end of the list, loop around to the beginning" sort of logic, so I abstracted all of it into the `Navigateable` class.

On this next page is a rough approximation of how I handle that logic in composables nowadays:

```
export default function useTablist (...) {
  ...

  const navigateable = useNavigateable(reactiveArrayOfTabs)

  // Right arrow: navigates to the next tab, looping around
  // to the beginning when necessary.
  function rightEffect (event) {
    event.preventDefault()
    navigateable.value.next()
  }

  // Left arrow: navigates to the previous tab, looping around
  // to the end when necessary.
  function leftEffect (event) {
    event.preventDefault()
    navigateable.value.previous()
  }

  // Home: Navigates to the first tab.
  function homeEffect (event) {
    event.preventDefault()
    navigateable.value.first()
  }

  // End: Navigates to the last tab.
  function endEffect (event) {
    event.preventDefault()
    navigateable.value.last()
  }
```

Inside `Navigateable` (which is plain, framework-agnostic TypeScript), I handle all kinds of navigation logic, so in my composables, I just need to create an instance of the class, then use its very simple, very semantic, very readable methods.

Much easier in the long run!

Keep an eye out for these patterns, and abstract them into more reusable, more easily testable, plain JavaScript or TypeScript implementations.

Check out Baleada Logic for inspiration!

> Keep an eye out for repetitive UI logic, especially when it forces you to write repetitive tests across multiple composables. Abstract that logic into self-contained functions, classes, or whatever feels best to you.

Split up your event handlers

When you're writing event handlers inside a composable, it's tempting to just add one listener per event type, and handle all possible outcomes of that event inside the same callback function.

For example, `useListbox` needs to handle all these different variations of the `keydown` event:

KEY	EFFECT
Up arrow	Activate the previous option.
Down arrow	Activate the next option.
Home	Activate the first option
End	Activate the last option.
Spacebar	Select the active option.
Enter	Select the active option.
Single characters	Append the character to the typeahead.

It's tempting to handle all this logic inside a single `keydown` event handler with a big `switch` statement:

```
function keydownEffect (event) {
  switch (event.key) {
    case 'ArrowUp':
      ...
    case 'ArrowDown':
      ...
    case 'Home':
      ...
    case 'End':
      ...
    case ' ':
      ...
    case 'Enter':
      ...
    default:
      ...
  }
}
```

At first glance, this feels like a sensible and efficient way to organize things. But take another look at that table of keys, and note that their side effects touch several different logical concerns:

- Managing the active item
- Managing the selected item
- Managing the typeahead

I've found that it makes more sense in the long run to just write separate keydown handlers for each bit of logic, so you can collocate event handlers with other related logic.

I almost always follow this advice. The only time I prefer to lump multiple concerns together in the same event handler is when the timing of certain side effects is very delicate and/or stateful.

For example, in `useTablist` I implemented the ability to delete the currently selected tab with a keystroke. This feature is designed for things like spreadsheet tabs, which can be created and deleted by the end user on the fly.

After a tab is deleted, the timing of side effects turned out to be very particular:

1. The index of the currently focused tab is internally cached.
2. The tab is deleted.
3. On the next tick, `useTablist` focuses the tab whose new index now matches the cached index.
4. In certain (user-configureable) situations, the selected panel doesn't automatically sync with the focused tab. I won't get further into the weeds here, but just know that panel selection updates need to happen *after* tab navigation updates in this case.

When I'm handling that delete keystroke, could I split all this logic up so that the tab focusing is in my "Focused" section of code, and panel navigation is in my "Selected" section? Yes, I could. Do I want to do that? No, because I prefer feeling confident about the timing of side effects, and because this stateful logic is so tightly coupled that I would never want to edit the various pieces in isolation during any refactor anyway.

Instead, I just moved the entire event handler down to a section of code titled "Multiple Concerns". If a refactor is ever necessary, I'll know to code carefully.

> Except when logic is *very* tightly coupled, split up event handler code based on **logical concern**, not based on the type of event being handled.

Expose reactive references and methods in the return value

When writing with the function ref pattern, you'll always return an object that contains function refs and/or function ref getters:

```
export function useListbox () {
  ...

  return { rootRef, getOptionRef }
}
```

Developers will attach these function refs to their template to unlock almost all functionality they need:

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

const options = ref([...])
const listbox = useListbox()
</script>

<template>
  <ul :ref="listbox.rootRef">
    <li
      v-for="(option, index) in options"
      :key="option"
      :ref="listbox.getOptionRef(index)"
    >
      ...
    </li>
  </ul>
</template>
```

But function refs won't cover everything! For example, developers will likely want to apply different styles to the active listbox option. They may also want to execute some side effect when the selected option changes, or add custom buttons to control the active or selected option.

You definitely could set up `useListbox` to accept an optional `activeOptionClasses` parameter, then internally bind those classes to the active item. You could also accept an `onSelect` parameter, passing a callback that you'll execute each time the selected item changes.

But for these more specific use cases, which are not essential for `useListbox`'s core behavior, I prefer to expose reactive references and methods in the return value, and let the developer use normal Vue APIs to work with that data.

For example, in `useListbox`, I return lots of core state:

```
export default function useListbox () {
  ...

  return {
    // FUNCTION REFS
    rootRef, getOptionRef,

    // REACTIVE REFERENCES
    active, selected,

    // METHODS THAT READ REACTIVE REFERENCES
    isActive, isSelected,

    // METHODS THAT WRITE TO REACTIVE REFERENCES
    activate, activatePrevious, activateNext, select
  }
}
```

Now, the developer can work with that data in a simpler, more familiar Vue style of programming:

```
<!-- CustomListbox.vue -->
<script setup>
import { ref, computed, watchEffect, onMounted } from 'vue'
import { useListbox } from 'path/to/useListbox'
import { updateDatabase } from 'path/to/updateDatabase'

const options = ref([ ... ])
const listbox = useListbox()
const selectedOption = computed(() =>
  options.value[listbox.selected.value]
)

// Every time the selected option changes,
// update the database with the new value.
watchEffect(() => {
  updateDatabase(selectedOption.value)
})

// Add a custom keyboard shortcut so the end user can
// press Command + a number 1 through 5 to select one
// of the first 5 options.
const handleKeydown = event => {
  if (!event.metaKey) return

  for (const key of ['1', '2', '3', '4', '5']) {
    if (event.key === key) {
      // The `select` method is exposed by `useListbox`
      // to make this kind of custom behavior possible.
      listbox.select(Number(key))
      return
    }
  }
}
</script>
```

Code continues on the next page

```vue
<template>
  <ul :ref="listbox.listbox.rootRef">
    <li
      v-for="(option, index) in options"
      :key="option"
      :ref="listbox.getOptionRef(index)"
      @keydown="handleKeydown"
    >
      <!--
        When this option is the active option, it has
        a blue background and white text.
      -->
      <div
        class="p-3"
        :class="[
          index === listbox.isActive(index)
            ? 'bg-blue-600 text-white'
            : 'bg-white text-gray-900'
        ]"
      >
        <span>{{ option }}</span>
        <!--
          When this option is the selected option, show a
          checkmark icon.
        -->
        <CheckmarkIcon v-show="listbox.isSelected(index)" />
      </div>
    </li>
  </ul>
</template>
```

Instead of accepting an awkward `activeItemClasses` optional parameter, and cluttering our composable with extra code to handle it, we can just return the `isActive` method, which lets developers easily read and react to `active` state.

And instead of accepting an `onSelect` optional parameter, we can just include the `selected` piece of state in our returned object. Developers can set up their own watchers to react to the changes.

The takeaway is: inside your function ref composable, only do the work you feel is necessary to implement essential logic and behavior.

After that's done, get out of the way! Expose your reactive references and methods to the developer consuming your composable, and let them have some fun with it.

> **Don't** accept wonky optional parameters to support all possible use cases.
>
> **Do** handle the complex, difficult, essential UI logic, then give your reactive references and methods to the developer, and let them have fun.

Test in a real browser

Often, you might not be using any special browser APIs inside your composables, and you can get pretty far with a mock environment like jsdom.

But as I discovered, and as Headless UI author Robin Malfait discovered, mocked browser APIs can be frustratingly imprecise at times.

This problem is magnified any time you're using the function ref pattern to build keyboard-accessible UIs, since keyboard navigation and its side effects can be particularly tricky to test in general, let alone in a mock environment.

These days, tools for testing in a real browser are ergonomic and quite fast. Using a mock environment will probably shave *some* time off your test runs, but I just can't stand the prospect of staring into the abyss of a failing test, my screen littered with the smoking remnants of old `console.log` statements, my eyes dry and bloodshot, wondering if the problem is my composable code, my test code, or some mock environment edge case.

Just use a real browser.

Plus, when browsers ship new features that you want to use, you don't have to wait for the mock environment to catch up—you can just write new code, test it, and ship.

Here are a some good options for real browser testing:

- Cypress if you want a truly feature-complete, properly staffed and funded, cross-browser testing stack, with an absurdly polished DX.
- Jest and `jest-puppeteer` if you're already super productive with Jest, and like the idea of adding just a touch of extra tooling and configuration to unlock in-browser testing.
- Vite with `@vitejs/plugin-vue` for serving testable Vue components to a local dev server, uvu for running tests and making assertions, and Puppeteer or Playwright for controlling the browser during tests. Perfect for minimalists, configuration haters, and speed freaks. Best when running on Node 15+, giving you native ES module support and allowing uvu to run modern JavaScript directly, with no test compilation step. My personal favorite for solo projects!

Recognize that accessibility leaves room for creativity

Sometimes, we get caught up in the idea that a single unequivocal standard will someday prescribe all the steps we need to take to make our apps and website accessible. Many people think that `WAI-ARIA` already is this set of standards.

It's true that the WAI ARIA specification lays out almost all the rules you need to follow to work with assistive technology, but there's still room for creativity, customization, and individual decisions about user experience.

A small example: in a listbox, menu, tablist, or any kind of list with keyboard navigation, certain items can be `disabled`. WAI-ARIA is intentionally ambiguous about whether or not disabled items can receive focus and have their contents read by assistive technology. The specification allows the developer building the listbox, menu, tablist, etc. to make that choice, based on what they believe is the best experience for assistive tech users.

When you're engineering for accessibility with the function ref pattern, remember that this kind of flexibility exists. It allows you to be creative, and tailor your user interface to provide the user experience you think is best.

Also, remember that the function ref pattern is uniquely well-suited to leaving accessibility decisions up to the developer who consumes your composable in their app.

For example, in the official `useListbox` function that I wrote as part of Baleada Features for production use, I accept a `disabledOptionsReceiveFocus` option to control focus behavior:

```
import { useListbox } from '@baleada/vue-features'

// `disabledOptionsReceiveFocus` defaults to `false`. Set it
// to `true`, and keyboard navigation will focus disabled options,
// allowing assistive tech to read the options' contents.
const listbox = useListbox({ disabledOptionsReceiveFocus: true })
```

Keep an eye out for things like this! In my experience, the code that supports this kind of creative behavior is usually the most interesting and most fun to write.

Further info: the Web Accessibility episode of Remotely Interesting was a really great resource for me when I was learning to think of accessibility as a creative part of my projects.

Get inspiration from other open-source projects

I'm not currently aware of any Vue composables, other than the ones I write in Baleada Features, that follow the function ref pattern. If you write or find some, tweet @AlPalVipond and send them my way!

That said, there are still plenty of great open-source projects you can use as inspiration for your next function ref pattern experiment.

In this book, I've mentioned code from the Headless UI project several times. Headless UI doesn't ship functions that follow the function ref pattern, but it's definitely worth studying for ideas on what kinds of logic can be implemented in composables.

VueUse is another good source of inspiration. It's a collection of composables written by many different authors, so the API designs vary significantly, but it covers all kinds of interesting UX cases that are sure to spark some new ideas.

Outside of Vue, you can look at Downshift, and don't miss React Aria. Both of these projects offer React hooks that follow patterns extremely similar to the function ref pattern to achieve the same DX and UX goals. When I'm writing code for Baleada Features, React Aria is the project I reference most closely for guidance on accessibility features.