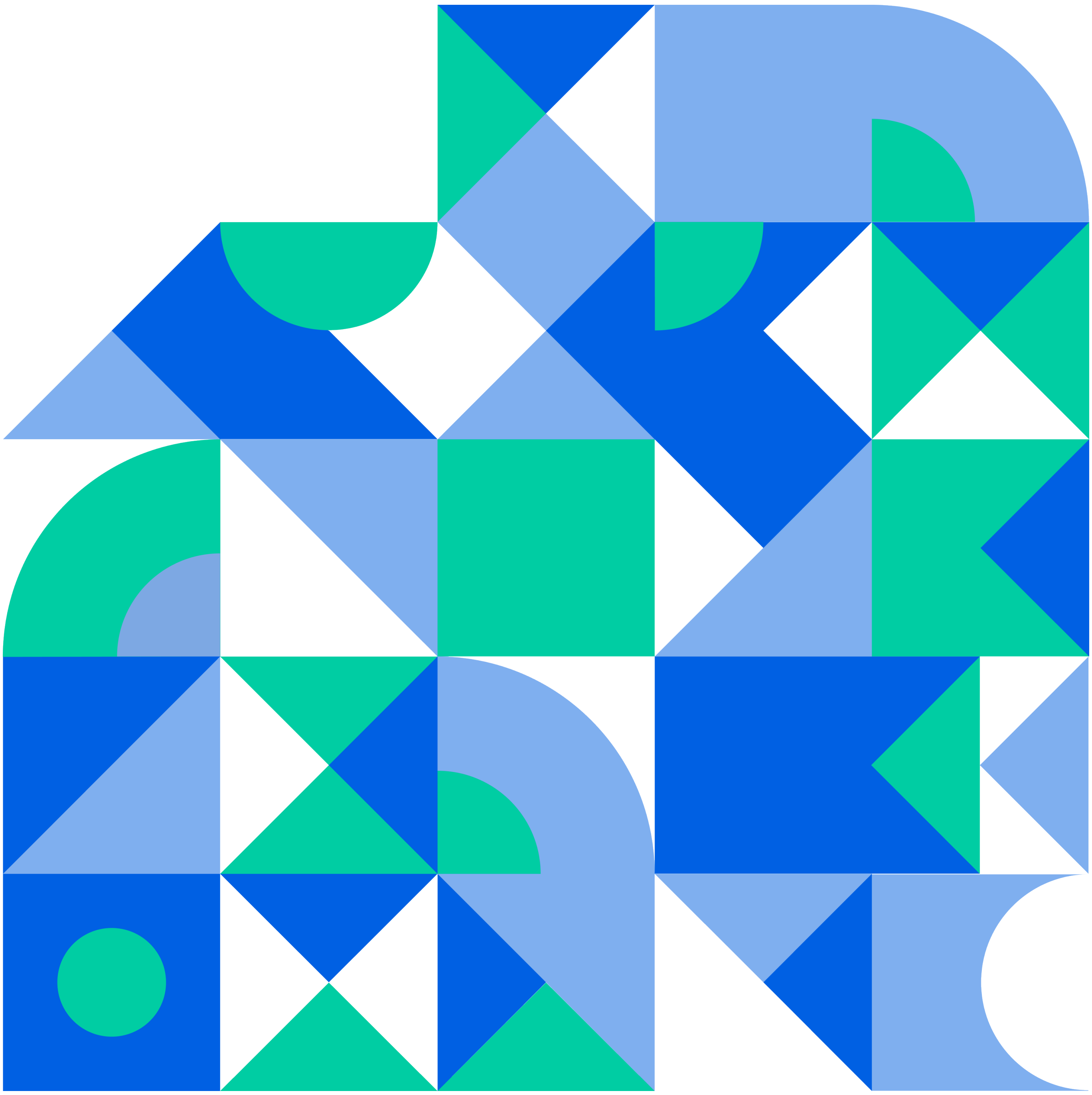
VUE TIPS COLLECTION

Michael Thiessen



INTRODUCTION

Hey there!

This book is a collection of short, concise tips on using Vue better, because dedicating hours to learning isn't always possible (or easy).

But 5 minutes a day, reading a tip here and there, is much more manageable!

The tips range from things you already knew but forgot about, obscure but practical features, to patterns and tricks to make your code better. They're loosely organized into categories to help you navigate the book.

Most of the tips are written by me, but some are contributions from a handful of incredible Vue community members. These contributions were edited by me from previously published content to fit the book's format and style.

I have put my heart and soul into this book, and I truly hope that you enjoy it.

— Michael

Michael Thiessen

@MichaelThiessen
michaelnthiessen.com

TABLE OF TIPS

Chapter 1

Forgotten Features

All those things you forgot Vue could do, or didn't know about in the first place.

1. A simpler way to pass lots of props

Instead of passing in tons of props to a component individually:

```
<template>
  <User
    :name="user.name"
    :profile="user.profile"
    :twitter="user.twitter"
  />
</template>
```

You can take a whole object and have all of its properties automatically bound to the component as props:

```
<template>
  <User v-bind="user"/>
</template>
```

```
export default {
  setup() {
    return {
      user: {
        name: 'Anakin',
        profile: 'ani-profile.jpg',
        twitter: '@TatooineJedi',
      },
    };
  },
};
```

This also works with `v-on` if you have a lot of event handlers:

```
<template>
  <User v-on="userEventHandlers"/>
</template>
```

```
export default {
  setup() {
    return {
      userEventHandlers: {
        updateName(newName) {
          // ...
        },
        deleteUser() {
          // ...
        },
        addFriend(friend) {
          // ...
        }
      },
    };
  },
};
```

2. Watching Arrays and Objects

The trickiest part of using a watcher is that sometimes it doesn't seem to trigger correctly.

Usually, this is because you're trying to watch an Array or an Object but didn't set `deep` to `true`:

```
export default {
  name: 'ColourChange',
  props: {
    colours: {
      type: Array,
      required: true,
    },
  },
  watch: {
    // Use the object syntax instead of just a method
    colours: {
      // This will let Vue know to look inside the array
      deep: true,

      // We have to move our method to a handler field
      handler()
        console.log('The list of colours has changed!');
      }
    }
  }
}
```

Using the reactivity API from Vue 3 would look like this:

```
watch(
  colours,
  () => {
    console.log('The list of colours has changed!');
  },
  {
    deep: true,
  }
);
```

Here are the docs for Vue 3 and Vue 2 if you want to read more on this.

3. Restrict a prop to a list of types

Using the `validator` option in a prop definition you can restrict a prop to a specific set of values:

```
export default {
  name: 'Image',
  props: {
    src: {
      type: String,
    },
    style: {
      type: String,
      validator: s ⇒ ['square', 'rounded'].includes(s)
    }
  }
};
```

This validator function takes in a prop and returns either `true` or `false` — if the prop is valid or not.

I often use this when I need more options than a `boolean` will allow but still want to restrict what can be set.

Button types or alert types (info, success, danger, warning) are some of the most common uses — at least in what I work on. Colours, too, are a really great use for this.

But there are many more!

4. How to watch anything in your component

It took me a very long time to realize this, but anything in your component that is reactive can be watched:

```
export default {
  computed: {
    someComputedProperty() {
      // Update the computed prop
    },
  },
  watch: {
    someComputedProperty() {
      // Do something when the computed prop is updated
    }
  }
};
```

You can watch:

- computed props
- props
- nested values

Any value can be watched if you're using the composition API, as long as it's a `ref` or `reactive` object.

5. Global Components

When you register a component globally, you can use it in any template without importing it a second time:

```
// Vue 3
import { createApp } from 'vue';
import GlobalComponent from './GlobalComponent.vue';

const app = createApp({})
app.component('GlobalComponent', GlobalComponent);
```

In Vue 2 you can register global components like this:

```
// Vue 2
import Vue from 'vue';
import GlobalComponent from './GlobalComponent.vue';
Vue.component('GlobalComponent', GlobalComponent);
```

Now you can use `GlobalComponent` in your templates without any extra work!

Of course, globally registered components have the same pros and cons as global variables. So use this sparingly.

6. Use quotes to watch nested values

You may not have known this, but you can easily watch nested values directly, just by using quotes:

```
watch: {
  '$route.query.id'() {
    // ...
  }
}
```

This is really useful for working with deeply nested objects!

7. Getting Meta with Nuxt 3

Tim Benniks
Principal Developer Advocate at Uniform

With Nuxt 3 we get some simple ways to update our meta tags, `<head>` and `<body>`.

You can do it from within the template with the meta components:

```
<template>
  <Head>
    <Title>{{ data }}</Title>
  </Head>
  <Body class="body-class"></Body>
  <!-- ... rest of the component here -->
</template>
```

Or using the `useMeta` composable that is globally available:

```
<script setup>
const data = await useFetch("/api/stuff");
useMeta({
  title: data,
  bodyAttrs: {
    class: "body-class",
  },
});
</script>
```

There's *a lot* more you can do with meta tags in Nuxt 3. Check it out on the docs.

8. h and Render Functions

When using the `render` function instead of templates, you'll be using the `h` function a lot:

```
<script setup>
import { h } from 'vue';
const render = () ⇒ h('div', {}, 'Hello Wurld');
</script>
```

It creates a VNode (virtual node), an object that Vue uses internally to track updates and what it should be rendering.

The first argument is either an HTML element name or a component (which can be async if you want):

```
<script setup>
import { h } from 'vue';
import MyComponent from './MyComponent.vue';

const render = () ⇒ h(MyComponent, {}, []);
</script>
```

The second argument is a list of props, attributes, and event handlers:

```
<script setup>
import { h } from 'vue';
import MyComponent from './MyComponent.vue';

const render = () ⇒ h(MyComponent, {
  class: 'text-blue-400',
  title: 'This component is the greatest',
  onClick() {
    console.log('Clicked!');
  },
}, []);
</script>
```

The third argument is either a string for a text node, an array of children VNodes, or an object for defining slots:

```
<script setup>
import { h } from 'vue';
import MyComponent from './MyComponent.vue';

const render = () => h(MyComponent, {}, [
  'Simple text node',
  h('span', {}, 'Text inside of a <span> element'),
]);
</script>
```

These render functions are essentially what is happening "under the hood" when Vue compiles your single file components to be run in the browser.

But by writing out the render function yourself, you are no longer constrained by what can be done in a template.

You have the full power of Javascript at your fingertips!

This is just scratching the surface on what render functions and `h` can do. Read more about them on the official docs.

9. Deep Linking with Vue Router

You can store (a bit of) state in the URL, allowing you to jump right into a specific state on the page.

For example, you can load a page with a date range filter already selected:

```
someurl.com/edit?date-range=last-week
```

This is great for the parts of your app where users may share lots of links, for a server-rendered app, or for communicating more information between two separate apps than a regular link provides typically.

You can store filters, search values, whether a modal is open or closed, or where in a list we've scrolled to — perfect for infinite pagination.

Grabbing the query using `vue-router` works like this (this will work on most Vue frameworks like Nuxt and Vuepress too):

```
const dateRange = this.$route.query.dateRange;
```

To change it we use the `RouterLink` component and update the `query`:

```
<RouterLink :to="{
  query: {
    dateRange: newDateRange
  }
}">
```

Here's a demo of this in action:

https://codesandbox.io/s/deep-linking-with-vue-router-vhxkq?file=/src/components/DeepLinking.vue

10. toRef default value

You've been using `toRef` for a while, but did you know you can also supply a default value?

```
const bank = reactive({
  Rand: 3400,
  Egwene: 20,
  Matrim: 230340,
  Padan: -20340,
})

// toRef(object, property, default)
const myBankAccount = toRef(bank, 'Michael', 1000 * 1000);
```

Probably the easiest way to become a millionaire.

11. Directives in Render Functions

Vue comes with some methods to help you use custom directives on your VNodes:

```
<script setup>
import { resolveDirective, withDirectives, h } from 'vue';

// Find the already registered directive by name
const focusDirective = resolveDirective('focus');

// Wrap the button with the directive
const render = () ⇒ withDirectives(
  h('button', {}, []),
  // An array of directives to apply
  [
    [focusDirective]
  ]
);
</script>
```

You can find more info on using `withDirectives` on the docs.

12. Vue to Web Component in 3 Easy Steps

First, create the custom element from a Vue component using `defineCustomElement`:

```
import { defineCustomElement } from 'vue';
import MyVueComponent from './MyVueComponent.vue';

const customElement = defineCustomElement(MyVueComponent);
```

Second, register the custom element with the DOM:

```
customElements.define('my-vue-component', customElement);
```

Third, use the custom element in your HTML:

```
<html>
  <head></head>
  <body>
    <my-vue-component></my-vue-component>
  </body>
</html>
```

Now you've got a custom web component that doesn't need a framework and can run natively in the browser!

Check out the docs for more details on how this works.

13. Private properties with script setup

You can limit what properties are available when a component is accessed by `$ref`:

```
export default {
  expose: ['makeItPublic'],

  data() {
    return {
      privateData: 'Keep me a secret!',
    };
  },

  computed: {
    makeItPublic() {
      return this.privateData.toUpperCase();
    },
  },
};
```

With only `makeItPublic` exposed, you can't access the `privateData` property through a `$ref` anymore:

```
this.$refs.component.privateData // Will always be undefined
```

If you're using `<script setup>`, everything is locked down by default. If you want to expose a value you have to do so explicitly:

```
<script setup>
  import { ref, computed } from 'vue';
  const privateData = ref('Keep me a secret!');
  const makeItPublic = computed(
    () => privateData.value.toUpperCase()
  );

  // We don't need to import this because it's a compiler macro
  defineExpose({
    makeItPublic
  });
</script>
```

Here `defineExpose` is a compiler macro, not an actual function, so we don't have to import anything.

You can find more info on this in the docs.

14. Special CSS pseudo-selectors in Vue

If you want some styles to apply specifically to slot content, you can do that with the `:slotted` pseudo-selector:

```
<style scoped>
  /* Add margin to <p> tags within the slot */
  :slotted(p) {
    margin: 15px 5px;
  }
</style>
```

You can also use `:global` to have styles apply to global scope, even within the `<style scoped>` block:

```
<style scoped>
  :global(body) {
    margin: 0;
    padding: 0;
    font-family: sans-serif;
  }
</style>
```

Of course, if you have lots of global styles you want to add, it's probably easier to just add a second `<style>` block:

```
<style scoped>
  /* Add margin to <p> tags within the slot */
  :slotted(p) {
    margin: 15px 5px;
  }
</style>

<style>
  body {
    margin: 0;
    padding: 0;
    font-family: sans-serif;
  }
</style>
```

Check out the docs for more info.

15. Custom Directives

Vue has a bunch of built in directives, like `v-for` and `v-if`, but we can also build our own:

```
<template>
  <h1 v-bg-colour="'skyblue'">This background is blue</h1>
</template>
```

```
import { createApp } from "vue";
const app = createApp({});

app.directive("bg-colour", {
  mounted(el, { value }) {
    // Update the background colour
    el.style.background = value;
  }
});
```

(Obviously, this simple directive is ridiculous because we would just use CSS)

You can also specify a single argument:

```
<template>
  <h1 v-bg-colour:colour="'skyblue'">
    This background is blue
  </h1>
</template>
```

```
app.directive("bg-colour", {
  mounted(el, { value, arg }) {
    // Update the background colour
    el.style.background = value;
    console.log(arg);   // "colour"
  }
});
```

Unfortunately, if you want multiple arguments Vue will treat them as distinct directives:

```
<template>
  <!-- Two directives will be mounted -->
  <h1
    v-bg-colour:colour="'skyblue'"
    v-bg-colour:animate="true"
  >
    This background is blue
  </h1>
</template>
```

Luckily for us, the value is evaluated as a Javascript expression, so we can pass in an options object instead:

```
<template>
  <!-- Two directives will be mounted -->
  <h1
    v-bg-colour="{
      colour: 'skyblue',
      animate: true,
    }"
  >
    This background is blue
  </h1>
</template>
```

```
app.directive("bg-colour", {
  mounted(el, { value }) {
    // Update the background colour
    el.style.background = value.colour;

    // Do something cool with value.animate
  }
});
```

There's a lot more you can do with custom directives. Check out the docs to find out more.

16. Next Tick: Waiting for the DOM to Update

Vue gives us a super handy way for us to wait for the DOM to finish updating:

```
// Do something that will cause Vue to re-render
changeTheDOM();

// Wait for the re-render to finish
await nextTick();

// Now we can check out the freshly minted DOM
inspectTheDOM();
```

Or if you're using the options API:

```
await this.$nextTick();
```

A tick is a single render cycle. First, vue listens for any reactivity changes, then performs several updates to the DOM in one batch. Then the next tick begins.

If you update something in your app that will change what is rendered, you *have to* wait until the next tick before that change shows up.

17. Destructuring in a v-for

Did you know that you can destructure in a `v-for`?

```
<li
  v-for="{ name, id } in users"
  :key="id"
>
  {{ name }}
</li>
```

It's more widely known that you can grab the index out of the v-for by using a tuple like this:

```
<li v-for="(movie, index) in [
  'Lion King',
  'Frozen',
  'The Princess Bride'
]">
  {{ index + 1 }} - {{ movie }}
</li>
```

When using an object you can also grab the key:

```
<li v-for="(value, key) in {
  name: 'Lion King',
  released: 2019,
  director: 'Jon Favreau',
}">
  {{ key }}: {{ value }}
</li>
```

It's also possible to combine these two methods, grabbing the key as well as the index of the property:

```
<li v-for="(value, key, index) in {
  name: 'Lion King',
  released: 2019,
  director: 'Jon Favreau',
}">
  #{{ index + 1 }}. {{ key }}: {{ value }}
</li>
```

Chapter 2

Slots and Reusability

Slots are the most powerful feature in Vue. They let us create more flexible and reusable components.

18. The limitations of props

Props are helpful, but they have two glaring issues:

1. Impossible to pass markup*
2. Not that flexible

**not technically impossible, but not something you want to do.*

The solution to these two problems is the same, but we'll get there in a second.

Many components you create are **contentless components**. They provide a container, and you have to supply the content. Think of a button, a menu, an accordion, or a card component:

```
<Card title="Shrimp Tempura">
  <img src="picOfShrimp.jpg" />
  <div>
    <p>Here are some words about tempura.</p>
    <p>How can you go wrong with fried food?</p>
  </div>
  <a href="www.michaelnthiessen.com/shrimp-tempura">
    Read more about Shrimp Tempura
  </a>
</Card>
```

You can often pass this content in as a regular `String`. But many times, you want to pass in a whole chunk of HTML, maybe even a component or two.

You can't do that with props.*

**again, yes, you could do this, but you'll definitely regret it.*

Props also require that you **plan for all future use cases** of the component. If your `Button` component only has two values for `type`, you can't just use a third without modifying the `Button`:

```
<!-- You just have to believe it will work -->
<Button type="AWESOME" />
```

```
// Button.vue
export default {
  props: {
    type: {
      type: String,
      // Unfortunately, 'AWESOME' doesn't work here :/
      validator: val => ['primary', 'secondary'].includes(val),
    }
  }
};
```

I'm not a psychic, and I'm guessing you aren't either.

The solution to these problems?

I think I gave it away with my card example above...

...slots!

Slots allow you to pass in whatever markup and components you want, and they also are relatively open-ended, giving you lots of flexibility. This is why in many cases, slots are simply better than props.

19. Splitting slots

Let's take a slot and split it into two slots:

```
<!-- Parent.vue -->
<template>
  <Child>
    <!-- Splitting a slot into multiple requires
         a conditional. This allows you to branch
         into however many slots you want -->
    <slot v-if="left" name="left" />
    <slot v-else name="right" />
  </Child>
</template>
```

Our `Child` component only accepts one slot, but the `Parent` component accepts two. Here, the `Parent` component switches between which slot it uses based on the value of `left`.

We can also use default slot content, if one or both of the `Parent` slots have no content:

```
<!-- Parent.vue -->
<template>
  <Child>
    <slot v-if="left" name="left">
      <!-- Default content works like you'd expect -->
      <div key="left">Default left</div>
    </slot>
    <slot v-else name="right">
      <div key="right">Default Right</div>
    </slot>
  </Child>
</template>
```

20. Conditionally rendering slots

First, I'll show you *how*, then we'll get into *why* you'd want to hide slots.

Every Vue component has a special `$slots` object with all of your slots in it. The default slot has the key `default`, and any named slots use their name as the key:

```
const $slots = {
  default: <default slot>,
  icon: <icon slot>,
  button: <button slot>,
};
```

But this `$slots` object only has the slots that are *applied* to the component, not every slot that is *defined*.

Take this component that defines several slots, including a couple named ones:

```
<!-- Slots.vue -->
<template>
  <div>
    <h2>Here are some slots</h2>
    <slot />
    <slot name="second" />
    <slot name="third" />
  </div>
</template>
```

If we only apply one slot to the component, only that slot will show up in our `$slots` object:

```
<template>
  <Slots>
    <template #second>
      This will be applied to the second slot.
    </template>
  </Slots>
</template>
```

```
$slots = { second: <vnode> }
```

We can use this in our components to detect which slots have been applied to the component, for example, by hiding the wrapper element for the slot:

```
<template>
  <div>
    <h2>A wrapped slot</h2>
    <div v-if="$slots.default" class="styles">
      <slot />
    </div>
  </div>
</template>
```

Now the wrapper `div` that applies the styling will only be rendered if we actually fill that slot with something.

If we don't use the `v-if`, we will have an empty and unnecessary `div` if we don't have a slot. Depending on what styling that `div` has, this could mess up our layout and make things look weird.

So why do we want to be able to conditionally render slots?

There are three main reasons to use a conditional slot:

1. When using wrapper `div's to add default styles
2. The slot is empty
3. If we're combining default content with nested slots

For example, when we're adding default styles, we're adding a `div` around a slot:

```
<template>
  <div>
    <h2>This is a pretty great component, amirite?</h2>
    <div class="default-styling">
      <slot >
    </div>
    <button @click="$emit('click')">Click me!</button>
  </div>
</template>
```

However, if no content is applied to that slot by the parent component, we'll end up with an empty `div` rendered to the page:

```
<div>
  <h2>This is a pretty great component, amirite?</h2>
  <div class="default-styling">
    <!-- No content in the slot, but this div
          is still rendered. Oops. -->
  </div>
  <button @click="$emit('click')">Click me!</button>
</div>
```

Adding that `v-if` on the wrapping `div` solves the problem though. No content applied to the slot? No problem:

```
<div>
  <h2>This is a pretty great component, amirite?</h2>
  <button @click="$emit('click')">Click me!</button>
</div>
```

Here's a Codesandbox with a working demo if you want to take a look:

Demo

I wrote more tips on slots in this article: Tips to Supercharge Your Slots (Named, Scoped, and Dynamic)

21. Understanding scoped slots

Here's the best way to think about scoped slots:

Scoped slots are like functions that are passed to a child component that returns HTML.

Once the template is compiled, they *are* functions that return HTML (technically `vnodes`) that the parent passes to the child.

Here's a simple list that uses a scoped slot to customize how we render each item:

```
<!-- Parent.vue -->
<template>
  <ScopedSlotList :items="items">
    <template v-slot="{ item }">
      <!-- Make it bold, just for fun -->
      <strong>{{ item }}</strong>
    </template>
  </ScopedSlotList>
</template>
```

```
<!-- ScopedSlotList.vue -->
<template>
  <ul>
    <li
      v-for="item in items"
      :key="item"
    >
      <slot :item="item" />
    </li>
  </ul>
</template>
```

We can rewrite this example to use a function instead of a scoped slot:

```
<!-- Parent.vue -->
<template>
  <ScopedSlotList
    :items="items"
    :scoped-slot="(item) => `<strong>${item}</strong>`"
  >
</template>
```

```
<!-- ScopedSlotList.vue -->
<template>
  <ul>
    <li
      v-for="item in items"
      :key="item"
      v-html="scopedSlot(item)"
    />
  </ul>
</template>
```

22. Slot transitions

It's possible to use transitions with slot content, but there's one key to making them work smoothly:

```
<!-- SlotWithTransition.vue -->
<template>
  <!-- Creating the base slot with a transition -->
  <transition name="fade" mode="out-in">
    <slot></slot>
  </transition>
</template>
```

Always make sure that content provided to the slot is keyed.

This helps Vue keep track of when to trigger the transition:

```
<template>
  <SlotWithTransition>
    <div v-if="isThisTrue" key="true">
      This is true.
    </div>
    <div v-else key="false">
      This is false.
    </div>
  </SlotWithTransition>
</template>
```

23. How to watch a slot for changes

Austin Gil
Building Vuetensils and ParticlesCSS

Sometimes we need to know when the content inside of a slot has changed:

```
<!-- Too bad this event doesn't exist -->
<slot @change="update" />
```

Unfortunately, Vue has no built-in way for us to detect this.

However, there is a very clean way of doing this using a mutation observer:

```
export default {
  mounted() {
    // Call `update` when something changes
    const observer = new MutationObserver(this.update);

    // Watch this component for changes
    observer.observe(this.$el, {
      childList: true
      subtree: true
    });
  }
};
```

But don't forget you'll also need to clean up the observer!

24. Reusable components aren't what you think

Reusable components don't have to be big or complex things.

I often make small and short components reusable.

Because I'm not re-writing this code all over the place, updating it becomes much easier, and I can make sure that every `OverflowMenu` looks and works exactly the same — because they *are* the same!

```
<!-- OverflowMenu.vue -->
<template>
  <Menu>
    <!-- Add a custom button to trigger our Menu -->
    <template #button v-slot="bind">
      <!-- Use bind to pass click handlers,
           a11y attributes, etc. -->
      <Button v-bind="bind">
        <!-- Use our own "..." icon and no text
             for this button -->
        <template #icon>
          <svg src="./ellipsis.svg" />
        </template>
      </Button>
    </template>
  </Menu>
</template>
```

Here we take a `Menu` component but add a '...' (ellipsis) icon to the button that triggers it to open.

It almost seems like it's not worth making a reusable component out of this because it's only a few lines. Can't we just add the icon every time we want to use a `Menu` like this?

But this `OverflowMenu` will be used dozens of times, and now if we want to update the icon or its behaviour, we can do it very quickly. And using it is much simpler too!

```
<template>
  <OverflowMenu
    :menu-items="items"
    @click="handleMenuClick"
  />
</template>
```

25. Recursive slots

I decided to see if I could make a `v-for` component using only the template. Along the way, I discovered how to use slots recursively, too.

This is what the component looks like:

```
<!-- VFor.vue -->
<template>
  <div>
    <!-- Render the first item -->
    {{ list[0] }}
    <!-- If we have more items, continue!
         But leave off the item we just rendered -->
    <v-for
      v-if="list.length > 1"
      :list="list.slice(1)"
    />
  </div>
</template>
```

If you wanted to do this with scoped slots — and why wouldn't you?! — it just takes a few tweaks:

```
<template>
  <div>
    <!-- Pass the item into the slot to be rendered -->
    <slot v-bind:item="list[0]">
      <!-- Default -->
      {{ list[0] }}
    </slot>

    <v-for
      v-if="list.length > 1"
      :list="list.slice(1)"
    >
      <!-- Recursively pass down scoped slot -->
      <template v-slot="{ item }">
        <slot v-bind:item="item" />
      </template>
    </v-for>
  </div>
</template>
```

Here is how this component is used:

```
<template>
  <div>
    <!-- Regular list -->
    <v-for :list="list" />

    <!-- List with bolded items -->
    <v-for :list="list">
      <template v-slot="{ item }">
        <strong>{{ item }}</strong>
      </template>
    </v-for>
  </div>
</template>
```

For a more detailed explanation of this example and nested slots, check out my blog post on it:

How to Use Nested Slots in Vue (including scoped slots)

26. Default content with slots

You can provide fallback content for a slot, in case no content is provided:

```html
<!-- Child.vue -->
<template>
  <div>
    <slot>
      Hey! You forgot to put something in the slot!
    </slot>
  </div>
</template>
```

This content can be anything, even a whole complex component that provides default behaviour:

```html
<!-- Child.vue -->
<template>
  <div>
    <slot name="search">
      <!-- Can be overridden with more advanced functionality -->
      <BasicSearchFunctionality />
    </slot>
  </div>
</template>
```

27. 6 levels of reusability

There are six different levels of reusability that you can use in your components.

Each level adds more complexity but also increases your ability to reuse code.

These levels are the focus of my course, Reusable Components.

Here are the six levels of reusability:

1. **Templating** — Reusing code by wrapping it up inside of a component
2. **Configuration** — Using configuration props to allow for varying behaviour
3. **Adaptability** — Allowing components to become future-proof
4. **Inversion** — Letting other components control the process
5. **Extension** — Using reusability throughout our component
6. **Nesting** — Creating powerful hierarchies of components

I cover this in more detail in this excerpt from the course.

28. Nesting slots

As you start creating more abstractions with Vue, you may need to begin nesting your slots:

```
<!-- Parent.vue -->
<template>
  <Child>
    <!-- We take the content from the grand-parent slot
         and render it inside the child's slot -->
    <slot />
  </Child>
</template>
```

This works similarly to how you would catch an error and then re-throw it using a `try ... catch` block:

```
try {
  // Catch the error
  reallyRiskyOperation();
} (e) {
  // Then re-throw as something else for the next
  // layer to catch and handle
  throw new ThisDidntWorkError('Heh, sorry');
}
```

Normally when using a slot we just render the content that's provided to us:

```
<template>
  <div>
    <!-- Nothing to see here, just a regular slot -->
    <slot />
  </div>
</template>
```

But if we don't want to render it in this component and instead pass it down again, we render the slot content inside of *another* slot:

```
<template>
  <Child>
    <!-- This is the same as the previous code example,
         but instead of a `div` we render into a component. -->
    <slot />
  </Child>
</template>
```

29. Improve reusability by converting template props into slots

One kind of prop, a template prop, can be directly converted into slots without very much work.

This makes your component more reusable.

The `text` prop here is a template prop, because it is only ever used in the template:

```
<template>
  <button @click="$emit('click')">
    {{ text }}
  </button>
</template>
```

```
export default {
  name: 'Button',
  props: {
    text: {
      type: String,
      required: true,
    },
  },
};
```

It doesn't get used in any calculations or passed as a prop anywhere. Instead, it just gets directly interpolated and rendered to the page.

These props can be directly replaced with slots:

```
<template>
  <button @click="$emit('click')">
    <slot />
  </button>
</template>
```

```
export default {
  name: 'Button',
};
```

This sort of cleans up the code, but more importantly, it allows us to be more flexible with how the component can be used.

With a prop we have to use the component like this:

```
<Button text="Click me" @click="handleClick" />
```

But with a slot, we can add in whatever we want:

```
<Button @click="handleClick">
  Click on <strong>this</strong> button
</Button>
```

30. Shorthand for named slots

Named slots also have a shorthand syntax, one that's much nicer to look at.

Instead of writing this:

```
<DataTable>
  <template v-slot:header>
    <TableHeader />
  </template>
</DataTable>
```

We can write this:

```
<DataTable>
  <template #header>
    <TableHeader />
  </template>
</DataTable>
```

Not a huge difference, but a little cleaner for sure. I think the `#` character is easier to pick out than `v-slot` when reading code.

31. Default content with nested slots

If you have multiple levels of nested slots, it's possible to have defaults at *each* level:

```
<!-- Parent.vue -->
<template>
  <Child>
    <slot>
      We're in the Parent!
    </slot>
  </Child>
</template>
```

```
<!-- Child.vue -->
<template>
  <div>
    <slot>
      We're in the Child!
    </slot>
  </div>
</template>
```

The slot content provided at the highest point in the hierarchy will override everything below it.

If we render `Parent`, it will always display `We're in the Parent`. But if we render just the `Child` component, we get `We're in the Child!`.

And if the component rendering the `Parent` component provides slot content, that will take precedence over everything:

```
<!-- Grandparent.vue -->
<template>
  <Parent>
    Haha this content rules them all!
  </Parent>
</template>
```

Chapter 3

Logic Lore

We can't forget about the business logic, now can we?

32. Global properties

It's possible to add global properties to your Vue app in both Vue 2 and Vue 3:

```
// Vue 3
const app = createApp({});
app.config.globalProperties.$myGlobal = 'globalpropertiesftw';

// Vue 2
Vue.prototype.$myGlobal = 'globalpropertiesftw';
```

I would recommend prefixing any global properties with a `$`.

This helps prevent naming conflicts with other variables, and it's a standard convention that makes it easy to spot when a value is global.

This global property can be accessed directly off of any component when using the Options API:

```
computed: {
  getGlobalProperty() {
    return this.$myGlobal;
  },
},
```

Why can't this be used with the composition API?

Because the composition API is designed to be context-free and has no access to `this`.

Instead, you can create a simple composable to access your globals:

```
<script setup>
import useGlobals from './useGlobals';
const { $myGlobal } = useGlobals();
</script>
```

```js
// useGlobals.js
export default () ⇒ ({
  $myGlobal: 'globalpropertiesftw',
});
```

33. Detect clicks outside an element

Sometimes I need to detect whether a click happens *inside* or *outside* of a particular element. This is the approach I typically use:

```
window.addEventListener('mousedown', e => {
  // Get the element that was clicked
  const clickedEl = e.target;

  // `el` is the element you're detecting clicks outside of
  if (el.contains(clickedEl)) {
    // Clicked inside of `el`
  } else {
    // Clicked outside of `el`
  }
});
```

34. Composable return values

When you're creating composables, here are some things you can do to make it easier to work with their return values.

Return an object of `refs`, so that reactivity is preserved when destructuring:

```
// Composable
const useBurger = () ⇒ {
  const lettuce = ref(true);
  const ketchup = ref(true);

  return {
    lettuce,
    ketchup,
  };
};
```

```
// Component
setup() {
  // We can destructure but still keep our reactivity
  const { ketchup } = useBurger();

  watchEffect(() ⇒ console.log(ketchup.value));

  return {
    ketchup,
    removeKetchup: () ⇒ ketchup.value = false
  };
},
```

If you don't want to destructure the values, you can always wrap it in `reactive` and it will be converted to a reactive object:

```
// Component
setup() {
  // Wrap in `reactive` to make it a reactive object
  const burger = reactve(useBurger());

  watchEffect(() ⇒ console.log(burger.ketchup));

  return {
    burger,
    removeKetchup: () ⇒ burger.ketchup = false
  };
},
```

One great thing VueUse does is return a single value by default. If you happen to need more granular control, you can pass in an option to get an object returned instead:

```
import { useTimestamp } from '@vueuse/core';

// Most of the time you only need the timestamp value
const timestamp = useTimestamp();

// But sometimes you may need more control
const {
  timestamp,
  pause,
  resume,
} = useTimestamp({ controls: true });
```

I think presenting different interfaces based on how you need to use the composable is a brilliant pattern. This makes it simpler to work with while not sacrificing any precise control.

35. Detect mouse hover in your component

You can detect a mouse hover in Vue just by listening to the right events:

```
<template>
  <div
    @mouseover="hover = true"
    @mouseleave="hover = false"
  />
</template>
```

Then you can use this state to change how the background is rendered, update computed props, or anything else you want.

Depending on your use case, you may want to check out the `mouseout` and `mouseenter` events as well. There are some subtleties with how they bubble and interact with child elements.

36. A bunch of composable mini tips

Yes, mini tips *within* a tip. It's meta.

Here they are:

- Start with the end in mind, and write the return first. Once you know how you want the composable to be used, filling in the implementation details is much easier.
- Use an options object as the parameter. This makes it easy to add new parameters in the future without breaking anything, and you won't mess up the ordering anymore.
- Keep them small. Embrace the UNIX philosophy and make sure each composable only does one thing but does it well.
- Name them consistently: `use___`, `create___`, `on___`
- Always make sure your reactivity is hooked up before any async logic. By using a `ref` of `null`, you can update those values later when your logic completes. No need to `await` around.
- Use `effectScope` to group effects if you have lots of them in your composable. This makes cleaning up your reactivity a lot simpler. If you have large objects, use `shallowRef` instead to prevent Vue from recursively making the whole thing reactive. Of course, you'll need to use `triggerRef` to trigger any reactive effects for it, but it can improve performance.

Some tips on making your composables more flexible:

- If you're using a `watch`, make `immediate` and `flush` configurable
- Accept both `refs` and primitive values as inputs. By passing the variable through `ref`, you'll either reuse the existing `ref` or create a new one.
- The same trick works with `unref` if what you need in your composable is a primitive and not a `ref`.

37. A simple undo composable

patak
Vite and Vitest core team member

Here's the basic undo composable that we'll break down:

```
function useSyncUndo(source, options) {
  const history = ref([source.value])
  // We use this flag to stop tracking updates
  const undoing = ref(false)

  const _commit = () => {
    history.value.unshift(source.value)
  }
  const _undo = () => {
    if (history.value.length > 1) {
      history.value.shift()
      source.value = history.value[0]
    }
  }
  const stop = watch(
    source,
    () => {
      if (!undoing.value) {
        _commit()
      }
    },
    { ...options, flush: 'sync' }
  )
  function undo() {
    undoing.value = true
    _undo()
    undoing.value = false
  }
  return { undo, history, stop }
}
```

First, we need a way to keep track of our `ref` history:

```
const history = ref([source.value])
const _commit = () => {
  // Add new value to the beginning of the array
  history.value.unshift(source.value)
}
```

Next, we automatically commit any updates by using a watcher.

With flush mode set to `sync` each update will get tracked, even if there are multiple during a single tick:

```
// We use this flag to stop tracking updates
const undoing = ref(false)
const stop = watch(
  source,
  () => {
    // This watch will also be triggered when we undo,
    // so we use this guard to avoid tracking that.
    if (!undoing.value) {
      _commit()
    }
  },
  { ...options, flush: 'sync' }
)
```

Now we can finally add our `undo` function:

```
function undo() {
  undoing.value = true
  // With flush set to `sync` the watch is triggered immediately.
  // Other flush modes _will not_ work for this.
  _undo()
  undoing.value = false
}
```

But don't write this composable yourself. Instead, you should use the bullet-proof and production-ready `useRefHistory` from VueUse.

38. When ref and reactive work the same

It can be confusing to know whether or not to use `ref` or `reactive`.

Here are a few instances where they end up working basically the same.

When using `watchEffect` dependencies are tracked automatically, so there isn't much difference between `ref` and `reactive`:

```
// Ref - just need to access through `value` property
const refBurger = ref({ lettuce: true });
watchEffect(() => console.log(refBurger.value.lettuce);

// Reactive
const reactiveBurger = reactive({ lettuce: true });
watchEffect(() => console.log(reactiveBurger.lettuce));
```

Also, because `refs` are automatically unwrapped in the template, there is no difference there:

```
<template>
  <!-- Ref -->
  {{ burger.lettuce }}

  <!-- Reactive -->
  {{ burger.lettuce }}
</template>
```

If you destructure an object you'll need to convert back to `refs` if you want reactivity:

```
// Using `ref`
const { lettuce } = toRefs(burger.value);

// Using `reactive`
const { lettuce } = toRefs(burger);
```

If you have to convert everything to `refs` anyway, why not just use them to start?

39. Structuring composables

To keep your composables — those extracted functions written using the composition API — neat and easy to read, here's a way to organize the code.

1. Component and directives
2. `provide` and `inject`
3. `defineProps`, `defineEmits`, and `defineExpose` (when using `script setup`)
4. `refs` and `reactive` variables
5. Computed properties
6. Immediate watchers
7. Watchers
8. Lifecycle methods, in their correct order
9. Non-reactive state
10. Methods
11. Async code using `await` (or Promises if you're into that sort of thing)

Why this order? Because it more or less follows the order of execution of the code.

It's also based on the Vue 3 styleguide.

The `await` needs to go at the end because most of your logic you'll want to be registered before the `setup` function returns. Anything after the `await` will only be executed asynchronously.

40. Ref vs. reactive

Is it better to use `ref` or `reactive` when using the composition API?

Here are a few situations where `ref` is better than `reactive`.

Using `ref` on objects makes it clear where an object is reactive and where it's just a plain object:

```
// I can expect this ref to update reactively
if (burger.value.lettuce) {
  // ...
}

// I have no clue if this value is reactive
if (burger.lettuce) {
  // ...
}
```

When using one of the `watch` methods, `refs` are automatically unwrapped, so they're nicer to use:

```
// Ref
const refBurger = ref({ lettuce: true });
watch(
  // Not much, but it's a bit simpler to work with
  refBurger,
  () => console.log("The burger has changed"),
  { deep: true }
);

// Reactive
const reactiveBurger = reactive({ lettuce: true });
watch(
  () => burger,
  () => console.log("The burger has changed"),
  { deep: true }
);
```

One last reason why `refs` make more sense to me — you can put `refs` into a `reactive` object. This lets you compose `reactive` objects out of `refs` and still use the underlying `refs` directly:

```
const lettuce = ref(true);
const burger = reactive({
  // The ref becomes a property of the reactive object
  lettuce,
});

// We can watch the reactive object
watchEffect(() ⇒ console.log(burger.lettuce));

// We can also watch the ref directly
watch(lettuce, () ⇒ console.log("lettuce has changed"));

setTimeout(() ⇒ {
  // Updating the ref directly will trigger both watchers
  // This will log: `false`, 'lettuce has changed'
  lettuce.value = false;
}, 500);
```

41. Vue Testing Library

One of my favourite tools for testing is Vue Testing Library:

```
test('displays correct text', () ⇒ {
  const { getByText } = render(MyComponent);
  getByText(/Fail the test if this text doesn't exist/);
})
```

It builds on top of `vue-test-utils`, making it easier to write tests closer to how users actually interact with your app.

Users look for specific text or look for a button to click. They don't look for the `nth child of a div with the class .booking-card-container`.

Writing tests this way makes them easier to understand, simpler to write, and more robust against changes to your code. Nothing about this test is concerned with the implementation, so it's unlikely to break even under heavy refactoring.

If this idea is new to you, I highly encourage you to read more about the Testing Library guiding principles.

42. Lazy loading images in a single line

Alex Jover Morales
DevRel at Storyblok

Using `v-lazy-image` we can lazily load images only when they're in the viewport:

```
<template>
  <v-lazy-image src="fullSizeImage.jpg" />
</template>
```

With `src-placeholder` we can also use progressive enhancement. First, the tiny low resolution image is loaded, then we switch to the full resolution image once it's been loaded:

```
<template>
  <v-lazy-image
    src="fullSizeImage.jpg"
    src-placeholder="tinySizeImage.jpg"
  />
</template>
```

With some simple CSS classes we can also animate the transition when the image loads. We can start with a blurred image and unblur to reveal the full image:

```
.v-lazy-image {
  filter: blur(10px);
  transition: filter 0.7s;
}
.v-lazy-image-loaded {
  filter: blur(0);
}
```

And we can also do this elegant fade in of the full resolution image:

```css
.v-lazy-image {
  opacity: 0;
  transition: opacity 2s;
}
.v-lazy-image-loaded {
  opacity: 1;
}
```

43. Requiring injections

Abdelrahman Awad
Senior Frontend Engineer at Octopods

Here's a simple composable that lets us require injections in our components:

```
// useRequiredInjection.js
import { inject } from 'vue';

export default useRequiredInjection = (key, fallback) => {
  const resolved = inject(key, fallback);
  if (!resolved) {
    throw new Error(`Could not resolve ${key.description}`);
  }
  return resolved;
}
```

Throwing exceptions is fine here. However, because this injection is required, we want to say there is no way this component works without that injection.

This is an effective way to declare that injection intent and makes it easier for your fellow developers to debug missing injections.

Using this composable is also very straightforward:

```
import useRequiredInjection from './useRequiredInjection';
const userId = useRequiredInjection('userId');
```

Chapter 4

CSS Tips

Although not *technically* part of Vue, CSS is an important part of any web application.

44. Don't override component CSS

It can be really tempting to quickly modify a component's CSS from outside the component. If all you need is a slight modification, it seems harmless — but it's not.

Let's say you have a normally blue button, but you need it to be green in this specific case. You can override the background colour from the parent component like this:

```
<template>
  <Button class="green">Make this button green</Button>
</template>

<style>
.green.button {
  background: green;
}
</style>
```

This *does* work, but it's very fragile and prone to breaking.

What if the class name changes?

What if the HTML of the component is modified?

Anyone making changes to the button component will have no idea that this component's background colour is overridden. They won't know to update this component too.

Instead, we can just extend the functionality of the button component. That way, we keep all of the code that modifies the button inside the button component.

Here, we can add a `is-green` prop to the button component:

```
<template>
  <Button is-green>Make this button green</Button>
</template>

<style>
/* No extra styles needed! */
</style>
```

Adding to the component itself makes it easier for anyone else who might need this button to be green in the future!

I've created a demo showing the original component and the new one with the added prop: https://codesandbox.io/s/wizardly-aryabhata-kn37d?file=/src/components/Button.vue

45. Defining your own utility classes in Tailwind

One of the best parts of using Tailwind is defining your own utility functions.

I recently needed a negative `z-index`, but Tailwind doesn't have one, so I created my own:

```css
@layer utilities {
  .-z-1 {
    z-index: -1;
  }
}
```

Anything wrapped with `@layer utilities { ... }` will be picked by Tailwind as a utility class.

If you need to use a custom utility class responsively or deal with hover states or dark mode, you get that all for free in v3.0!

However, if you're in an older version of you can wrap it in a `@variants responsive { ... }` block:

```css
@layer utilities {
  @variants responsive {
    .-z-1 {
      z-index: -1;
    }
  }
}
```

This lets you write `md:-z-1 lg:z-0` and have the utility class respond to screen size.

46. Two-sided transitions

I *love* adding nice transitions in CSS. It adds extra movement and makes the page feel more alive.

One thing I really like doing is adding **two-sided transitions**, where the effects of adding and removing a class are different.

Here's an example to check out: https://codepen.io/michaelthiessen/pen/vYgyGKe

To achieve this effect, you just need to have a `transition` property on both states:

```css
.transition {
  transition: background 1s;
  background: rgb(100,100,200);
}

.transition:hover {
  transition: background 0.15s;
  background: rgb(150,150,250);
}
```

This transition adds a nice fade between two background colours when we hover over the element with our mouse.

When the mouse goes over, the transition lasts just 150 ms. When the mouse is taken off, it does a slow 1 second transition back to the original colour.

Try playing around with this Codepen and see if you come up with something awesome!

47. Don't set height and width

Setting the `height` (or `width`) explicitly on an element is a code smell for me, whether you're using `px` or percentages or any other units.

Why do I not like it?

Because there's almost always a better way, and the `height` property usually makes things more difficult. It also messes with how the flexbox and grid layout systems work.

Instead, it's better to rely on the natural size of the content, padding, borders, and margin.

Of course, there are some caveats where I *do* like using `height` and `width`:

- Filling space using `100%` or `100vh`
- Making specific shapes, like a circle that has a height and width of `50px` (you'll see this all over my site)

I'm more likely to set `min-height` or `max-height` to create more responsive designs.

48. Grid template areas

Sometimes complicated layouts are, well, complicated. But using `grid-template-areas` is here to help!

```
<section>
  <header>Page header</header>
  <nav>Nav bar</nav>
  <main>Content goes here!</main>
  <footer>Not for your feet</footer>
</section>
```

With this HTML, you'd first attach `grid-area` names to each element:

```
header { grid-area: header; }
nav { grid-area: nav; }
main { grid-area: main; }
footer { grid-area: footer; }
```

Now you can describe your layout:

```
section {
  /* ... some other CSS grid set up here */
  grid-template-areas: "header header"
                       "nav    main"
                       "footer footer";
}
```

And if you need a single column layout for mobile (with the nav at the bottom, just for fun):

```
section {
  grid-template-areas: "header"
                       "main"
                       "nav"
                       "footer";
}
```

It's pretty easy to see exactly how the page is laid out with `grid-template-areas`.

49. Creating grids without holes

If you're using CSS grid with different sized items, you'll find that you end up with gaps here and there. You can force the browser to fill those holes:

```
grid-auto-flow: dense;
```

Keep in mind that this will break the ordering of your elements, which also breaks the tab order.

The `grid-auto-flow` property also takes two other interesting values: `row` and `column`. By default, it will fill each row one by one, but you can set it to fill by column instead.

You can also combine the two and get a dense column-based ordering!

More details and a good example illustrating how this works on MDN.

50. Using multiple box shadows

One box shadow is nice, but adding even more can produce some really complex and interesting effects.

We can have as many as we want, as long as they are all separated by a comma:

```css
.shadows {
  box-shadow: 0 0 0 1px rgba(0,0,0,0.4),
              0 0 20px 0 rgba(0,0,0,0.2);
}
```

Stacking them can create some interesting effects, like here where I've stacked different colors, but positioned them differently around the element:

```css
.shadows {
  box-shadow: 0 0 0 1px rgba(0,0,0,0.4),
              4px 4px 10px 2px rgba(100,100,250,0.4),
              -4px 4px 10px 2px rgba(250,100,100,0.4),
              -4px -4px 10px 2px rgba(100,250,100,0.4),
              4px -4px 10px 2px rgba(250,250,100,0.8);
}
```

This is definitely a fun one to play around with! Check out my Codepen if you want to try it out: https://codepen.io/michaelthiessen/pen/VwPzqOB?editors=1100

51. How I deal with dynamic classes

A pattern I use constantly is triggering classes with `boolean` flags:

```
<template>
  <div :class="disabled && 'disabled-component'">
    Sometimes this component is disabled. Other times it isn't.
  </div>
</template>
```

```
/* Some styles */
.disabled-component {
  background-color: gray;
  color: darkgray;
  cursor: not-allowed;
}
```

Either the trigger is a prop I use directly, or a computed prop that tests for a specific condition:

```
disabled() {
  return this.isDisabled || this.isLoading;
}
```

If I just need one class on an element, I use the logical AND to trigger it:

```
<div :class="disabled && 'disabled-component'"></div>
```

Sometimes it's a decision between two classes, so I'll use a ternary:

```
<div
  :class="disabled ? 'disabled-component' : 'not-yet-disabled'"
/>
```

I don't often use more than two classes like this, but that's where an `Object` or `Array` comes in handy:

```
<div
  :class="{
    primary: isPrimary,
    secondary: isSecondary,
    tertiary: isTertiary,
  }"
/>
```

```
<div
  :class="[
    isPrimary && 'primary',
    isSecondary && 'secondary',
    isTertiary && 'tertiary',
  ]"
/>
```

Of course, when it gets complex enough it's better to just have a computed prop that returns a string of class names (or returns an `Object` or `Array`):

```
<div :class="computedClasses" />
```

52. Reactive CSS

In Vue 3 we can use reactive values in our `<style>` block just like we can in the `<template>` block:

```
<style scoped>
  .button {
    color: v-bind(buttonColor);
  }
</style>
```

Behind the scenes, Vue uses CSS computed properties (aka CSS variables) scoped to each component.

The CSS remains static, but we can dynamically update the CSS variables whenever the reactive value changes.

More info can be found in the docs.

53. Shapes in CSS

Making rectangles in CSS is easy, and maybe you already know how to create a circle. But what about other shapes?

Check out this Codepen to see how these and some other shapes work: https://codepen.io/michaelthiessen/pen/jOyEqwp

Circle

To make a circle, just set `border-radius: 50%` on a square:

```css
.circle {
  width: 100px;
  height: 100px;
  border-radius: 50%;
}
```

Semi-Circle

Here we create a circle but then use another `div` with `overflow: hidden` to cut off half of it:

```html
<div class="semi-circle">
  <div></div>
</div>
```

```css
.semi-circle {
  width: 300px;
  height: 300px;
  overflow: hidden;
}

.semi-circle > div {
  /* Create a circle that fills the box */
  border-radius: 50%;
  width: 100%;
  height: 100%;

  /* Shift the circle down so half of it gets cut off */
  position: relative;
  top: 50%;

  background: teal;
}
```

Triangle

If we play around with border thicknesses, we can also get angled shapes, like triangles and rhombuses:

```css
.triangle {
  border-bottom: solid teal 150px;
  border-left: solid transparent 100px;
  border-right: solid transparent 100px;
}
```

CSS tricks has a great short article on how this works if you want to dive into it more.

54. Overriding styles of a child component — the right way

Scoped CSS is fantastic for keeping things tidy and not accidentally bleeding styles into other parts of your app.

But sometimes, you need to override the styles of a child component and break out of that scope.

Vue has a `deep` selector just for this:

```
<style scoped>
/* Override CSS of a child component
   while keeping styles scoped */
.my-component >>> .child-component {
  font-size: 24px;
}
</style>
```

Yes, I have previously covered why you shouldn't do this, but overriding styles *can* be the best solution (we don't believe in "best practices" here).

Note: If you're using a CSS pre-processor like SCSS, you may need to use `/deep/` instead.

55. Mixing local and global styles together

Normally, when working with styles we want them to be scoped to a single component:

```
<style scoped>
  .component {
    background: green;
  }
</style>
```

In a pinch though, you can also add a non-scoped style block to add in global styles if you need it:

```
<style>
  /* Applied globally */
  .component p {
    margin-bottom: 16px;
  }
</style>

<style scoped>
  /* Scoped to this specific component */
  .component {
    background: green;
  }
</style>
```

Be careful, though — global styles are dangerous and hard to track down. Sometimes, though, they're the perfect escape hatch and precisely what you need.

56. Clean up your Tailwind classes

After using Tailwind for awhile, you'll start to notice you end up with a lot of elements that have tons of classes on them:

```
<div
  class="mx-5 w-1/2 md:mx-0 md:w-auto mb-8 flex justify-end"
>
```

You can combine all of these utility classes together into a single class by using the `@apply` directive:

```
<template>
  <div class="combined"></div>
  <div class="combined"></div>
</template>
```

```
<style>
.combined {
  @apply mx-5 w-1/2 md:mx-0 md:w-auto mb-8 flex justify-end;
}
</style>
```

This lets you reuse those styles without having to create a whole new Vue component for them.

Of course, one of the best parts of Vue is that we can create reusable components, but not everything needs to be (or deserves to be) its own component.

Note: This is considered an anti-pattern in many cases. It is better to create your own utility class or encapsulate this within a component.

57. Invisible outlines

Adding a border to an element can get *really* annoying when it shifts your beautiful layout around.

One way to solve this is by creating an invisible outline — an outline that has no width, no height, and doesn't take up any space.

The way we achieve this is by creating a box shadow that has a spread but with zero blur and zero offset:

```css
box-shadow: 0 0 0 2px black;
```

The values for a CSS box shadow are:

```css
box-shadow: <offset-x> <offset-y> <blur-radius> <spread> <color>;
```

By having no offset, we keep the shadow centred on the element, so it's the same thickness all the way around. With no blur, we make sure that it appears like a solid line.

But because we have no offset or blur, we have to include a spread — otherwise, the shadow has zero width!

Here's a demo if you want to play around with it a bit: https://codepen.io/michaelthiessen/pen/rNjzzdW

Chapter 5

Powerful Patterns

Understand better ways of thinking about your components.

58. Creating magic with context-aware components

Context-aware components are "magical" — they adapt to what's going on around them automatically, handling edge cases, state sharing, and more.

There are 3 main types of context-aware components, but configuration is the one I find most interesting.

1. State Sharing

When you break up a large component into smaller ones, they often still need to share state.

Instead of pushing that work on whoever's consuming the components, you can make this happen "behind the scenes."

To give you more flexibility, you may break up a `Dropdown` component into `Select` and `Option` components. But to make it easier to use, the `Select` and `Option` components share the `selected` state with each other:

```
<!-- Used as a single component for simplicity -->
<Dropdown v-model="selected" :options="[]" />

<!-- Split up for more flexibility -->
<Select v-model="selected">
  <Option value="mustard">Mustard</Option>
  <Option value="ketchup">Ketchup</Option>
  <div class="relish-wrapper">
    <Option value="relish">Relish</Option>
  </div>
</Select>
```

2. Configuration

Sometimes component behaviour needs to change based on what's going on in the rest of the application. This is often done to automagically handle edge cases that would otherwise be annoying to deal with.

A `Popup` or `Tooltip` should reposition itself so it doesn't overflow out of the page. But if that component is inside a modal, it should move, so it doesn't overflow out of the *modal*.

This can be done automagically if the `Tooltip` knows when it's inside a modal.

3. Styling

You already create context-aware CSS, applying different styles based on what's happening in parent or sibling elements.

```css
.statistic {
  color: black;
  font-size: 24px;
  font-weight: bold;
}

/* Give some separation between stats
   that are right beside each other */
.statistic + .statistic {
  margin-left: 10px;
}
```

CSS variables let us push this further, allowing us to set different values in different parts of the page.

59. Where do you put shared state?

Let's say we have a `Button` component that toggles an `Accordion` open and closed by changing the variable `isOpen`.

But the `Button` component changes it's text between "Show" and "Hide" based on the same variable, `isOpen`:

```
// Parent.vue
<template>
  <!-- Both components need access to `isOpen`  -->
  <Button :is-open="isOpen" @click="toggleOpen" />
  <Accordion :is-open="isOpen">
    Some interesting content in here.
  </Accordion>
</template>
```

These two sibling components (because they are beside each other) need access to the same state, so where do we put it?

Answer: The lowest common ancestor!

Which, in this case, is the parent of both components.

Because state only flows down through props, shared state *must* be in a common ancestor. And we also want to keep state as close as possible, so we put it in the *lowest* common ancestor.

While this example may seem obvious to some, it's harder to see that this is the solution when the components sharing state are in separate components, in different folders.

Note: we also want to co-locate state with the logic that modifies it, so we have to put the `toggleOpen` method in the parent.

60. Default content and extension points

Slots in Vue can have default content, which allows you to make components that are much easier to use:

```
<button class="button" @click="$emit('click')">
  <slot>
    <!-- Used if no slot is provided -->
    Click me
  </slot>
</button>
```

My favourite use for default slots is using them to create *extension points.*

Basically, you take any part of a component, wrap it in a slot, and now you can override that part of the component with whatever you want. By default, it'll still work the way it always has, but now you have more options:

```
<template>
  <button class="button" @click="$emit('click')">
    <!-- Adding in the slot tag does nothing at first -->
    <!-- We can override this by providing content to the slot -->
    <slot>
      <div class="formatting">
        {{ text }}
      </div>
    </slot>
  </button>
</template>
```

Now you can use this component in many different ways. The easy, default way, or your own, custom way:

```
<!-- Uses default functionality of the component -->
<ButtonWithExtensionPoint text="Formatted text" />

<!-- Use the extension point to create custom behaviour -->
<ButtonWithExtensionPoint>
  <div class="different-formatting">
    Do something a little different here
  </div>
</ButtonWithExtensionPoint>
```

Here's a Codesandbox you can dive into:

https://codesandbox.io/s/default-content-and-extension-points-bl87m?file=/src/App.vue

61. Make test cases super specific

Markus Oberlehner
Exploring how to build software that lasts

Write tests for specific scenarios:

I often see tests with very short descriptions. I can imagine that they probably were obvious to the author when writing them.

But they are not so clear anymore if one of them breaks in a test suite of 200+ test cases:

Imagine we have a test with a nebulous description that breaks, and you have to figure out what's wrong and how to fix it:

```
test('show an error message', … )
```

It would be much easier if the test case description was more precise:

```
test('It should show an error message when user
      tries to save incomplete data', … )
```

Now we know precisely in which circumstances the app should show an error message. There is no mystery about what we need to fix when the test fails.

62. Directly accessing parent components (and why)

Props down, events up. That's how your components should communicate — most of the time.

But in *rare cases*, that just doesn't work.

If you need direct access to the parent component, you can use the instance property `$parent`:

```
// Tight coupling like this is usually a bad idea
this.$parent.methodOnParentComponent();
```

You can also get direct access to the application root, the very top-most component in the tree, by using `$root`. Vue 2 also has `$children`, but it was taken out for Vue 3 (please don't use this one).

When would these be useful?

There are a few different scenarios I can think of. Usually, when you want to abstract some behaviour and have it work "magically" behind the scenes.

You don't want to use props and events to connect up a component in those cases. Instead, you use `provide` / `inject`, `$parent`, or `$root`, to *automatically* connect the components and make things happen.

But it's hard to come up with an example where this is the best solution. Using `provide` / `inject` is almost always the better choice.

63. Async components

Using async components is a great way to speed up the initial load time of your app.

It's pretty straightforward to use in Vue 3:

```
// Regular, synchronous component
import MyComponent from './MyComponent.vue';

// Loading it asynchronously
import { defineAsyncComponent } from 'vue';
const AsyncComponent = defineAsyncComponent(
  () ⇒ import('./MyComponent.vue')
);
```

That's all there is to it. Seriously! Just use `AsyncComponent` like any other component.

The syntax for Vue 2 is not that different, you just don't need to use a helper function:

```
const AsyncComponent = () ⇒
  import('./components/LazyLoadingFTW.vue');
```

You can also provide a bunch of other options for async components, including:

- Loading and error components
- Delay before showing the loading component
- Timeout before showing the error component
- Custom `onError` handler

It also automatically hooks into the new suspense feature, which creates all sorts of magic.

Here are the docs for Vue 3 async components.

64. Stop testing the implementation

Markus Oberlehner
Exploring how to build software that lasts

When writing tests, avoid any technical details about *how* the system works:

The worst mistake I've made when writing tests was to rely on implementation details. When testing a Vue component, we don't want to make any assumptions about how it works internally.

Instead of writing this test:

```
test('It should fetch a list of products from the `/products`
      endpoint and render them in an HTML table', ... )
```

Write this test:

```
test('It should display a list of products', ... )
```

How our component gets its data isn't essential.

The same is true about which exact way the component renders the information.

In this example, the job of our component under test is to show a list of products to our users. By not putting in any details about exactly how our component achieves this, we can refactor it at any time.

Ideally, we should also avoid testing implementation details in the test code itself.

65. Why there must be one source of truth

This is the most critical principle of state management that I know:

Each piece of state has a single owner, a single source of truth.

No one else is allowed to modify the state. It's just borrowed temporarily.

If you have an array in a Vuex store, only that Vuex store can update it. Anywhere in your app that needs that value *must* get it from the store (directly or through another component).

If the state is owned by a component, only that component can modify it. This means other components must emit an event to that component, which can then decide what it wants to do.

Why?

If you allow state to be modified from anywhere, your code becomes a tangled mess.

With a single source of truth, you can easily understand what's happening.

The best code is easily understood.

66. Calling a method from outside of the component

You can call a method from outside of a component by giving it a `ref`:

```
<!-- Parent.vue -->
<template>
  <ChildComponent ref="child" />
</template>
```

```
// Somewhere in Parent.vue
this.$refs.child.method();
```

Let me explain this one a bit more.

Sometimes "best practices" don't work for what you're doing, and you need an escape hatch like this.

Typically, we communicate between components using props and events. Props are sent down into child components, and events are emitted back up to parent components.

```
<template>
  <ChildComponent
    :tell-me-what-to-do="someInstructions"
    @something-happened="hereIWillHelpYouWithThat"
  />
</template>
```

Occasionally, you may need your parent to trigger a method in the child component. This is where *only* passing props down doesn't work as well.

You *could* pass a boolean down and have the child component watch it:

```
<!-- Parent.vue -->
<template>
  <ChildComponent :trigger="shouldCallMethod" />
</template>
```

```
// Child.vue
export default {
  props: ['trigger'],
  watch: {
    shouldCallMethod(newVal) {
      if (newVal) {
        // Call the method when the trigger is set to `true`
        this.method();
      }
    }
  }
}
```

This works fine, but only on the first call. If you needed to trigger this multiple times, you'd have to clean up and reset the state. The logic would then look like this:

1. The Parent component passes `true` to `trigger` prop
2. Watch is triggered, and the Child component calls the method
3. The Child component emits an event to tell the Parent component that the method has been triggered successfully
4. The Parent component resets `trigger` back to `false`, so we can do this all over again

Ugh.

Instead, if we set a `ref` on the child component we can call that method directly:

```
<!-- Parent.vue -->
<template>
  <ChildComponent ref="child" />
</template>
```

```
// Somewhere in Parent.vue
this.$refs.child.method();
```

Yes, we're breaking the "props down, events up" rule and breaking encapsulation, but it's so much cleaner and easier to understand that it's worth it!

Sometimes the "best" solution ends up being the *worst* solution.

67. Using composition and options API together

Jake Dohm
In love with VueJS and TailwindCSS

It's possible to use a hybrid of the Options API and Composition API — but there are a couple caveats.

First, be aware that you *cannot* access properties declared in the Options API within your `setup` function:

```
<script>
  export default {
    data() {
      return {
        count: 1,
      };
    },
    setup() {
      // You cannot access properties from the Options API
      // It's impossible to access the component instance `this`
      const double = useDouble(this.count);
    }
  };
</script>
```

Second, you can access properties returned from `setup` within your Options API code:

```
<script>
  export default {
    setup() {
      const message = 'Hello, Vue friends!';
      return { message };
    },
    computed: {
      messageUpperCase() {
        // Properties returned from `setup` can be
        // used in the Options API
        return this.message.toUpperCase();
      }
    }
  };
</script>
```

This is a good reason to either:

- Go all-in on the Composition API (the officially recommended approach)
- Only use the Composition API to consume composables for code reusability

Splitting business logic between both approaches within a single component will only cause you headaches.

68. Simpler testing with dependency injection

Jest makes it easy to mock or stub out functions, but you can also use dependency injection to make things easy to stub:

```
export default {
  props: {
    fetchData: {
      type: Function,
      required: true,
    },
  },
  methods: {
    setText() {
      this.text = this.fetchData();
    },
  },
};
```

```
it('should work', () => {
  const { getByText } = render(MyComponent, {
    props: {
      async fetchData() {
        return 'Test text';
      },
    },
  });
  getByText(/Test text/);
});
```

(Example is simplified to illustrate the concept)

This is great for mocking API calls or anything tricky when testing.

If it's coming from outside of the component, it's pretty straightforward to stub it out or mock it how you need to get the test to do what you want.

You can do this in a variety of ways, depending on your use case:

- props
- provide/inject

- Vuex
- custom plugin

(There are probably many more)

69. Controlled props — or how to override internal state

Here we have a simple `Toggle` component that can show or hide content:

```
<template>
  <Toggle title="Toggled Content">
    This content can be hidden by clicking on the toggle.
  </Toggle>
</template>
```

It keeps track of its own `open` state internally right now.

But what if we want to override that internal state, but only *some of the time*?

To do this, we have to dynamically switch between relying on props and events, and relying on the internal state:

```
export default {
  props: {
    title: {
      type: String,
      required: true,
    },
    hidden: {
      type: Boolean,
      // Must be set to `undefined` and not `false`
      default: undefined,
    }
  },

  data() {
    // Internal state
    return { _hidden: false };
  },

  methods: {
    toggleHidden() {
      // Switch between emitting an event and toggling state
      if (this.hidden !== undefined) {
        this.$emit('toggle-hidden');
      } else {
        this._hidden = !this._hidden;
      }
    },
  },

  computed: {
    $hidden() {
      // Dynamically switch between state or prop
      return this.hidden !== undefined
        ? this.hidden
        : this._hidden;
    },
  },
};
```

In the `Toggle` component we now have to use the `$hidden` computed prop:

```
<template>
  <div>
    <div
      class="title"
      @click="toggleHidden"
    >
      {{ title }}
    </div>
    <slot v-if="$hidden" />
  </div>
</template>
```

You can check out a more detailed tutorial on my blog.

70. Component Seams Framework: How to split up components

Here's a technique for splitting up components:

The code you write forms natural groups. You want to identify these groups and the seams that run between them.

Once you do that, it's easy to extract components — by keeping things in their natural groups as much as possible.

The Component Seams Framework helps you do that in just three steps:

1. **Find the seams** — they can be found in your template by looking for *repeated sections* and sections that *perform different tasks*. You can also find them in your props, state, and computed props, by looking for things that are *related* and are often *updated together*.
2. **Stack the seams** — take all of the seams you've found, line them up, and you'll start to see where they agree (and where they don't).
3. **Split along the seams** — piece by piece, we'll pull things out and then figure out what to do with its dependencies. Either *include the dependency* in the new component, or *pass it in* to the new component somehow (prop, slot, or event).

This method is covered in more detail in my course Clean Components if you want to learn more.

71. Component metadata

Not every bit of info you add to a component is state. For example, sometimes, you need to add metadata that gives *other* components more information.

For example, if you're building a bunch of different widgets for an analytics dashboard like Google Analytics:

If you want the layout to know how many columns each widget should take up, you can add that directly on the component as metadata:

```
export default {
  name: 'LiveUsersWidget',
  // Just add it as an extra property
  columns: 3,
  props: {
    // ...
  },
  data() {
    return {
      //...
    };
  },
};
```

You'll find this metadata as a property on the component:

```
import LiveUsersWidget from './LiveUsersWidget.vue';
const { columns } = LiveUsersWidget;
```

You can also access the metadata from within the component through the special `$options` property:

```
export default {
  name: 'LiveUsersWidget',
  columns: 3,
  created() {
    // `$options` contains all the metadata for a component
    console.log(`Using ${this.$options.metadata} columns`);
  },
};
```

Just keep in mind that this metadata is the same for each component instance and is *not* reactive.

Other uses for this include (but are not limited to):

- Keeping version numbers for individual components
- Custom flags for build tools to treat components differently
- Adding custom features to components beyond computed props, data, watchers, etc.
- and many more I can't think of!

See a live example here: https://codesandbox.io/s/vue-metadata-bew9j?file=/src/App.vue

72. Smooth dragging (and other mouse movements)

If you ever need to implement dragging or to move something along with the mouse, here's how you do it:

1. Always throttle your mouse events using `requestAnimationFrame`. Lodash's `throttle` method with no `wait` parameter will do this.

If you don't throttle, your event will fire faster than the screen can even refresh, and you'll waste CPU cycles and the smoothness of the movement. 2. Don't use absolute values of the mouse position. Instead, you should check how far the mouse has moved between frames. This is a more reliable and smoother method.

If you use absolute values, the element's top-left corner will jump to where the mouse is when you first start dragging. Not a great UX if you grab the element from the middle.

Here's a basic example of tracking mouse movements using the composition API. I didn't include throttling in order to keep things clearer:

```
// In your setup() function
window.addEventListener("mousemove", (e) => {
  // Only move the element when we're holding down the mouse
  if (dragging.value) {
    // Calculate how far the mouse moved since the last
    // time we checked
    const diffX = e.clientX - mouseX.value;
    const diffY = e.clientY - mouseY.value;

    // Move the element exactly how far the mouse moved
    x.value += diffX;
    y.value += diffY;
  }

  // Always keep track of where the mouse is
  mouseX.value = e.clientX;
  mouseY.value = e.clientY;
});
```

Here's the full example:

```
<template>
  <div class="drag-container">
    <img
      alt="Vue logo"
      src="./assets/logo.png"
      :style="{
        left: `${x}px`,
        top: `${y}px`,
        cursor: dragging ? 'grabbing' : 'grab',
      }"
      draggable="false"
      @mousedown="dragging = true"
    />
  </div>
</template>
```

```
<script>
import { ref } from "vue";

export default {
  setup() {
    const dragging = ref(false);
    const mouseX = ref(0);
    const mouseY = ref(0);
    const x = ref(100);
    const y = ref(100);

    window.addEventListener("mousemove", (e) ⇒ {
      if (dragging.value) {
        const diffX = e.clientX - mouseX.value;
        const diffY = e.clientY - mouseY.value;
        x.value += diffX;
        y.value += diffY;
      }
      mouseX.value = e.clientX;
      mouseY.value = e.clientY;
    });

    window.addEventListener("mouseup", () ⇒ {
      dragging.value = false;
    });

    return {
      x,
      y,
      dragging,
    };
  },
};
</script>
```

You can check out a working demo here

73. Component variations with the Base Component pattern

The Base Component pattern is one of my favourite ways to make many different versions and variants from a single component.

It has a few basic steps:

1. Create your base component
2. Wrap it with another component to get a variant of the original
3. Repeat step 2 as many times as you need

Here's an example, creating a `DisabledButton` variant out of a `BaseButton` component:

```
<!-- DisabledButton.vue -->
<template>
  <!-- Never forget how to create this disabled button.
       Package it up using the Base Component pattern. -->
  <BaseButton
    type="disabled"
    disabled
  >
    <!-- You can't accidentally use the wrong icon now.
         It's provided here for you -->
    <template #icon>
      <Icon type="disabled" />
    </template>
  </BaseButton>
</template>
```

You can use this pattern in many different ways:

- **Lock down props** — take a `Button` component and hard code a few props to get a `DisabledButton`. Now you can just use the `DisabledButton` directly without fiddling with all the necessary props each time.
- **Lock down slots** — create an `InfoButton` variant where the icon passed to the `Button` is always the same. So now, if you ever need to change the icon (or anything else), you can do it in one place.
- **Simplify props** — sometimes components end up with dozens of props, primarily for edge cases. Instead, create a `BaseButton` with all the props, and a `Button` that passes on only the most common ones. This is a lot safer and easier to use, and the documentation for this component is easier to read.

I've included more on this pattern in Reusable Components.

74. Lonely children

Here's a technique that makes it super easy to simplify nested elements:

You take everything inside a `v-if` or `v-for` and extract it into a new component.

You'll go from this, with some nesting:

```html
<template>
  <div>
    <!-- ... -->
    <div v-for="item in list">
      <h2 class="item-title">
        {{ item.title }}
      </h2>
      <p class="item-description">
        {{ item.description }}
      </p>
    </div>
    <!-- ... -->
  </div>
</template>
```

To this, where the nesting is gone:

```html
<template>
  <div>
    <!-- ... -->
    <ListItem
      v-for="item in list"
      :item="item"
    />
    <!-- ... -->
  </div>
</template>
```

To do this, you extract the code in the `v-for` into a new component:

```
<!-- ListItem.vue -->
<template>
  <div>
    <h2 class="item-title">
      {{ item.title }}
    </h2>
    <p class="item-description">
      {{ item.description }}
    </p>
  </div>
</template>
```

This technique becomes more and more valuable the more nesting you have.

Note: You can choose to do this recursively, taking *every* `v-for` or `v-if` and creating a new component. But often, it's simpler to grab a more significant chunk of the template and remove most of the nesting with one new component.

I've written about this technique in more detail here.

75. Better injection keys

Abdelrahman Awad
Senior Frontend Engineer at Octopods

If you use `provide` and `inject` a lot in your apps, you can run into trouble if your keys are simple strings:

```js
// injectionKeys.js

export const AUTH_USER_KEY = 'USER';
// some injections later...
export const CURRENT_USER_KEY = 'USER';
```

Name collisions and typos are *extremely* hard to track down.

But there's a better way, using symbols:

```js
// injectionKeys.js

export const AUTH_USER_KEY = Symbol('USER');
// some injections later...
export const CURRENT_USER_KEY = Symbol('USER');
```

Every symbol is unique, so we no longer have to worry about name collisions.

But now we need to pass these symbols around so we can reuse them easily:

```js
// In a Vue component far, far, away...
import { inject } from 'vue';
import { AUTH_USER_KEY } from './injectionKeys.js';
const user = inject(AUTH_USER_KEY);
```

76. How to make a variable created outside of Vue reactive

If you get a variable from outside of Vue, it's nice to be able to make it reactive.

That way, you can use it in computed props, watchers, and everywhere else, and it works just like any other state in Vue.

If you're using the options API, all you need is to put it in the `data` section of your component:

```
const externalVariable = getValue();

export default {
  data() {
    return {
      reactiveVariable: externalVariable,
    };
  }
};
```

If you're using the composition API with Vue 3, you can use `ref` or `reactive` directly:

```
import { ref } from 'vue';

// Can be done entirely outside of a Vue component
const externalVariable = getValue();
const reactiveVariable = ref(externalVariable);

// Access using .value
console.log(reactiveVariable.value);
```

Using `reactive` instead:

```
import { reactive } from 'vue';

// Can be done entirely outside of a Vue component
const externalVariable = getValue();
// Reactive only works with objects and arrays
const anotherReactiveVariable = reactive(externalVariable);

// Access directly
console.log(anotherReactiveVariable);
```

If you're still on Vue 2 (as many of us are), you can use `observable` instead of `reactive` to achieve precisely the same result.

77. Tests before implementation

Markus Oberlehner
Exploring how to build software that lasts

Write tests *before* the implementation:

Tests are an excellent tool to guide us to write better code. However, when writing tests after the fact, what happens is that we have a hard time writing them. This is because we need to write complicated test code and our tests rely heavily on implementation details.

In the long run, we will find that our tests are not helping us.

We tightly coupled them to implementation details, making refactoring difficult. Every time we change a tiny piece of code, we need to adapt at least one of our tests as well.

We have to realize that this is not primarily a problem with our tests. It is a problem with the architecture of our code. Our tests work against us because we wrote code that is impossible to write good tests for.

Writing a failing test *before* writing code will help us write decoupled, thus testable and reusable code. TDD is not primarily a tool for writing good tests; it is a tool for writing better code.

Practicing TDD makes us better programmers!

78. Reusability Fundamentals: The Configuration Pattern

So you've got a fantastic `CodeBlock` component that does syntax highlighting and even shows line numbers:

```
<CodeBlock language="js">
  const myMessage = 'Highlighting code is supa ez';
</CodeBlock>
```

But now, you need to support a second colour theme.

Instead of copy and pasting (which is sometimes the right solution!), we can use props to help us create variations:

```
<!-- Uhhh, maybe not the best solution -->
<DarkModeCodeBlock language="js">
  const myMessage = 'Highlighting code is supa ez';
</DarkModeCodeBlock>
```

```
<!-- This is what props were meant for -->
<CodeBlock
  language="js"
  theme="darkMode"
>
  const myMessage = 'Highlighting code is supa ez';
</CodeBlock>
```

You already do this intuitively, so this may not be a huge revelation.

But **the Configuration pattern is a fundamental pattern** — you can't ignore it if you want to master reusability.

Dealing with prop explosions and understanding the Base Component Pattern is also part of mastering Configuration, the second level of reusability.

And the other, more exciting levels of reusability?

Well, mastering Configuration is vital to unlocking them. All the other levels build on top of this one.

79. Suspense: More Flexible Loading State

Anthony Gore
Creator of VueJS Developers

You've probably written lots of components that handle their own loading state.

Either while data is being fetched or while an async component is loaded.

But with the new `Suspense` component, we can instead handle that loading state further up the component tree:

```
<template>
  <Suspense>

    <!-- Shown once ChatWindow loads -->
    <template #default>
      <ChatWindow />
    </template>

    <!-- Loading state -->
    <template #fallback>
      <Spinner color="blue" />
    </template>

  </Suspense>
</template>
```

```
<script setup>
import { defineAsyncComponent } from "vue";
import Spinner from "@/components/Spinner.vue";

// ChatWindow will load asynchronously, but it doesn't
// have to have any loading state at all. Or even know
// that it's being loaded asynchronously!
const ChatWindow = defineAsyncComponent(
  () => import("@/components/ChatWindow")
);
</script>
```

This also works if the child component returns a Promise from the `setup` function:

```
async setup() {
  // `await` implicitly returns a Promise
  const { users } = await loadData();

  return {
    users,
  };
}
```

Even better is that the async child component can be *anywhere* as a descendant.

This means you can have a "root" loading state in your app, and any async components anywhere will trigger this loading state:

```
<!-- App.vue -->
<template>
  <Suspense>
    <template #default>
      <AppWithAsyncBehaviour />
    </template>
    <template #fallback>
      <FullPageLoading />
    </template>
  </Suspense>
</template>
```

80. Master computed props

When a function does more than just return a value, it complicates your code.

These are called side effects, and you should never have them inside of a computed prop:

```
export default {
  computed: {
    fullName() {
      this.fetchUserInformation();  // Side effect
      return `${this.firstName} ${this.lastName}`;
    },
  },
};
```

However, fixing this is quite straightforward. We can just move that side effect into a watcher that is triggered whenever the computed prop updates:

```
export default {
  computed: {
    fullName() {
      return `${this.firstName} ${this.lastName}`;
    },
  },

  watch: {
    fullName() {
      this.fetchUserInformation();  // Side effect
    },
  },
};
```

This applies equally to the composition API, although the syntax is slightly different.

At first glance, this may seem like we made the code more complicated. But actually, we've made it a lot simpler.

This concept is expanded in my course, Clean Components.

81. Stealing prop types

Often I find that I'm copying prop types from a child component just to use them in a parent component. But I've discovered that *stealing* those prop types is much better than just copying them.

For example, we have an `Icon` component being used in this component:

```
<template>
  <div>
    <h2>{{ heading }}</h2>
    <Icon
      :type="iconType"
      :size="iconSize"
      :colour="iconColour"
    />
  </div>
</template>
```

To get this to work, we need to add the correct prop types, copying from the `Icon` component:

```
import Icon from './Icon';
export default {
  components: { Icon },
  props: {
    iconType: {
      type: String,
      required: true,
    },
    iconSize: {
      type: String,
      default: 'medium',
      validator: size => [
        'small',
        'medium',
        'large',
        'x-large'
      ].includes(size),
    },
    iconColour: {
      type: String,
      default: 'black',
    },
    heading: {
      type: String,
      required: true,
    },
  },
};
```

What a pain.

And when the prop types of the `Icon` component are updated, you can be sure that you'll forget to return to this component and update them. Over time bugs will be introduced as the prop types for this component start to drift away from the prop types in the `Icon` component.

So that's why we'll steal them instead:

```
import Icon from './Icon';
export default {
  components: { Icon },
  props: {
    ...Icon.props,
    heading: {
      type: String,
      required: true,
    },
  },
};
```

It doesn't have to get any more complicated than that!

Except in our example, we have an "icon" added to the beginning of each prop name. So we'll have to do some extra work to get that to happen:

```
import Icon from './Icon';

const iconProps = {};

// Do some processing beforehand
Object.entries(Icon.props).forEach((key, val) => {
  iconProps[`icon${key[0].toUpperCase()}${key.substring(1)}`] = val;
});

export default {
  components: { Icon },
  props: {
    ...iconProps,
    heading: {
      type: String,
      required: true,
    },
  },
};
```

If the prop types in the `Icon` component are modified, our component will stay up-to-date.

But what if a prop type is added or removed from the `Icon` component? We can use `v-bind` and a computed prop to keep things dynamic and cover those cases.

82. Prevent navigating away

Austin Gil
Building Vuetensils and ParticlesCSS

We can use the native `beforeunload` event to detect when a user is about to navigate away or refresh the page:

```
// Use `beforeMount` to avoid running during SSR
onBeforeMount(() => {
  // We use the native `beforeunload` event
  window.addEventListener("beforeunload", preventNav);
});

// Make sure to always clean up after yourself!
onBeforeUnmount(() => {
  window.removeEventListener("beforeunload", preventNav);
});
```

The method to actually block the navigation uses `preventDefault`:

```
const blockNavigation = ref(false);
const preventNav = event => {
  if (!blockNavigation.value) return;
  event.preventDefault();
  // Chrome requires returnValue to be set
  event.returnValue = "";
};
```

Here's the full example:

```
<script setup>
import { ref, onBeforeMount, onBeforeUnmount } from 'vue';

// Method to block navigation
const blockNavigation = ref(false);
const preventNav = event => {
  if (!blockNavigation.value) return;
  event.preventDefault();
  // Chrome requires returnValue to be set
  event.returnValue = "";
};

// Use `beforeMount` to avoid running during SSR
onBeforeMount(() => {
  // We use the native `beforeunload` event
  window.addEventListener("beforeunload", preventNav);
});

// Make sure to always clean up after yourself!
onBeforeUnmount(() => {
  window.removeEventListener("beforeunload", preventNav);
});
</script>
```

Chapter 6

Template Tidbits

Time to spice up your view with these tasty template treats.

83. Static and dynamic classes

We can add static *and* dynamic classes to an element at the same time:

```
<ul>
  <li
    v-for="item in list"
    :key="item.id"
    class="always-here"
    :class="item.selected && 'selected'"
  >
    {{ item.name }}
  </li>
</ul>
```

This lets you apply basic styling through static classes and then dynamically add other styles as you need them.

You can also achieve the same thing when using an `Object` or `Array` with dynamic classes:

```
<ul>
  <li
    v-for="item in list"
    :key="item.id"
    :class="{
      'always-here': true,
      selected: item.selected,
    }"
  >
    {{ item.name }}
  </li>
</ul>
```

Or with an `Array`:

```
<ul>
  <li
    v-for="item in list"
    :key="item.id"
    :class="[
      'always-here',
      item.selected && 'selected',
    ]"
  >
    {{ item.name }}
  </li>
</ul>
```

I prefer splitting them out into `class` and `:class` bindings though, since it makes the code clearer. It also makes it less likely to be broken when refactored!

84. Multiple v-models

In Vue 3 we're not limited to a single `v-model`:

```
<AddressForm
  v-model:street-name="streetName"
  v-model:street-number="streetNumber"
  v-model:postal-code="postalCode"
  v-model:province="province"
  v-model:country="country"
/>
```

This makes dealing with complex forms a lot easier!

First, we need to create the props and events for `v-model` to hook into (I've omitted a couple v-models for simplicity):

```
<!-- AddressForm.vue -->
<script setup>
// Set up all the props
defineProps({
  streetName: {
    type: String,
    required: true,
  },
  streetNumber: {
    type: Number,
    required: true,
  },
  // ...
  country: {
    type: String,
    required: true,
  },
});

// Set up our events
defineEmits([
  'update:streetName',
  'update:streetNumber',
  // ...
  'update:country',
]);
</script>
```

Then, inside the component we use the prop to read the value, and emit `update:<propname>` to update it:

```
<template>
  <form>
    <input
      type="text"
      :value="streetName"
      @input="$emit('update:streetName', $event.target.value)"
    >
    <input
      type="text"
      :value="streetNumber"
      @input="$emit('update:streetNumber', $event.target.value)"
    >
    <!-- ... -->
    <input
      type="text"
      :value="country"
      @input="$emit('update:country', $event.target.value)"
    >
  </form>
</template>
```

You can read more about using multiple v-models here.

85. v-pre and v-once

If you've got large chunks of static or mostly static content, you can tell Vue to ignore it using the `v-pre` or `v-once` directives:

```
<template>
  <!-- These elements never change -->
  <div v-pre>
    <h1 class="text-center">Bananas for sale</h1>
    <p>
      Come get this wonderful fruit!
    </p>
    <p>
      Our bananas are always the same price — $1 each!
    </p>

    <div class="rounded p-4 bg-yellow-200 text-black">
      <h2>
        Number of bananas in stock: as many as you need
      </h2>
      <p>
        That's right, we never run out of bananas!
      </p>
    </div>

    <p>
      Some people might say that we're... bananas about bananas!
    </p>
  </div>
</template>
```

These can be helpful performance optimizations if you need them.

With `v-pre`, Vue will treat the element and its children as static HTML and won't do any of its magic on it. The `v-once` directive tells Vue to evaluate it once and never update it again.

Here are the docs for v-pre and v-once.

86. Dynamic directives

Vue lets us use dynamic arguments with directives:

```
<template>
  <WowSoDynamic
    v-bind:[somePropName]="somePropValue"
    v-on:[customEvent]="handleClick"
  />
</template>
```

When the argument evaluates to `null`, the attribute or event is removed.

This makes a really convenient way of making them conditional:

```
<!-- LinkComponent.vue -->
<template>
  <a
    :href="url"
    v-bind:[targetAttr]="'_blank'"
  >
    <slot />
  </a>
</template>
<script setup>
import { computed } from 'vue';

const { newTab } = defineProps({
  newTab: {
    type: Boolean,
    default: false,
  },
  url: {
    type: String,
    required: true,
  },
});

// If newTab === false set this to null
const targetAttr = computed(() => newTab ? 'target' : null);
</script>
```

By using a computed prop, we set `targetAttr` to `null` if `newTab` is `false`.

Either we add the `target` attribute with a value of `_blank`, or we don't add it.

87. Reactive SVG components

SVGs can be reactive components, too.

After all, they're HTML elements just like `div`, `span`, and `button`.

Here's an SVG component that has a prop to change it's fill colour:

```
<template>
  <svg viewBox="0 0 100 100" xmlns="http://www.w3.org/2000/svg">
    <circle cx="50" cy="50" r="50" :fill="color" />
  </svg>
</template>
```

```
<script>
export default {
  name: "SVGComponent",
  props: {
    color: String,
  },
};
</script>
```

I'm sure you can build some pretty wild things if you dig into different SVG elements and attributes.

Scoped slots and SVGs? Why not...

Here's a demo if you want to see this example in action.

88. Teleportation

You can get an element to render *anywhere* in the DOM with the `teleport` component in Vue 3:

```
<template>
  <div>
    <div>
      <div>
        <teleport to="body">
          <footer>
            This is the very last element on the page
          </footer>
        </teleport>
      </div>
    </div>
  </div>
</template>
```

This will render the `footer` at the very end of the document `body`:

```
<html>
  <head><!-- ... --></head>
  <body>
    <div>
      <div>
        <div>
          <!-- Footer element was moved from here... -->
        </div>
      </div>
    </div>
    <!-- ...and placed here -->
    <footer>This is the very last element on the page</footer>
  </body>
</html>
```

This is very useful when the logic and state are in one place, but they should be rendered in a different location.

One typical example is a notification (sometimes called a toast).

We want to be able to display notifications from wherever inside of our app. But the notifications should be placed at the end of the DOM so they can appear on top of the page:

```html
<!-- DogList.vue -->
<template>
  <div>
    <DogCard
      v-if="dogs.length > 0"
      v-for="dog in dogs"
      :key="dog.id"
      v-bind="dog"
    />
    <teleport to="#toasts">
      <!-- Show an error notification if we have an error -->
      <Toast
        v-if="error"
        message="Ah shoot! We couldn't load all the doggos"
      >
    </teleport>
  </div>
</template>
```

This will render this to the DOM:

```html
<html>
  <head><!-- ... --></head>
  <body>
    <div id="#app">
      <!-- Where our Vue app is normally mounted -->
    </div>
    <div id="toasts">
      <!-- All the notifications are rendered here,
           which makes positioning them much easier -->
    </div>
  </body>
</html>
```

Here's the complete documentation: https://v3.vuejs.org/api/built-in-components.html#teleport

89. Template refs

Matt Maribojoc
Founder of LearnVue *<a class="link" href="https://learnvue.co/2021/06/the-beginners-guide-to*

We can use template refs to directly access elements in our template:

```
<template>
  <div>
    <input
      type="text"
      placeholder="Start typing ... "
      ref="input"
    />
  </div>
</template>
```

When using the composition API, we provide the ref, and Vue will set the value to the element of the same name:

```
<script setup>
import { ref, onMounted } from "vue";

onMounted(() ⇒ {
  // Once mounted it's assigned the DOM element
  input.value.focus();
});

// Initially set to null
const input = ref(null);
</script>
```

If we're using the options API it's a bit different. Instead of creating a `ref` ourselves, we have to access the template ref through a special `$refs` object:

```
<script>
export default {
  mounted() {
    // Access our input using template refs, then focus
    this.$refs.input.focus()
  }
}
</script>
```

90. How to get rid of extra template tags

Scoped slots are lots of fun, but you have to use a lot of `template` tags to use them.

Luckily, a shorthand lets us get rid of it, but only if we're using a single scoped slot.

Instead of writing this:

```
<DataTable>
  <template #header="tableAttributes">
    <TableHeader v-bind="tableAttributes" />
  </template>
</DataTable>
```

We can write this:

```
<DataTable #header="tableAttributes">
  <TableHeader v-bind="tableAttributes" />
</DataTable>
```

Simple, straightforward, and marvellous.

(Ok, maybe not *quite* marvellous, but still pretty good)

I actually have a whole course on writing cleaner Vue code like this.

91. Configuration driven templates

One trick I like to simplify my templates is making them *configuration-driven*.

Instead of writing a template that repeats itself like this:

```
<template>
  <div>
    <MenuItem @click="openContact">
      Open Contact
    </MenuItem>
    <MenuItem @click="createContact">
      Create Contact
    </MenuItem>
    <MenuItem
      @click="resetSystem"
      danger
    >
      Reset
    </MenuItem>
  </div>
</template>
```

You can refactor the template to this to use configuration to drive it:

```
<template>
  <div>
    <MenuItem
      v-for="item in menuItems"
      :key="item.text"
      :danger="item.danger"
      @click="item.action"
    >
      {{ item.text }}
    </MenuItem>
  </div>
</template>
```

```
<script setup>
const openContact = () ⇒ { /* ... */ };
const createContact = () ⇒ { /* ... */ };
const resetSystem = () ⇒ { /* ... */ };

const menuItems = [
  {
    text: 'Open Contact',
    action: openContact,
  },
  {
    text: 'Create Contact',
    action: createContact,
  },
  {
    text: 'Reset System',
    action: resetSystem,
    danger: true,
  },
]
</script>
```

We don't have less code, but we have *simpler* code:

- **It's easier to read configuration** — Reading code is complicated because you have to figure out what the logic is doing, but configuration is very straightforward to understand.
- **Less logic means less bugs** — The config above is just a boring array with some objects. It's pretty simple, so it's unlikely that bugs would come from there.
- **Dynamic components are more flexible** — Because we've made this menu component dynamically render out the different menu items, we gain tremendous flexibility.

I've written much more on this idea here.

92. Dynamic slot names

We can dynamically generate slots at runtime, giving us even more flexibility in how we write our components:

```
<!-- Child.vue -->
<template>
  <div v-for="step in steps" :key="step.id">
    <slot :name="step.name" />
  </div>
</template>
```

Each of these slots works like any other named slot. This is how we would provide content to them:

```
<!-- Parent.vue -->
<template>
  <Child :steps="steps">
    <!-- Use a v-for on the template to provide content
         to every single slot -->
    <template v-for="step in steps" v-slot:[step.name]>
      <!-- Slot content in here -->
    </template>
  </Child>
</template>
```

We pass all of our `steps` to the `Child` component so it can generate the slots. Then we use a dynamic directive argument `v-slot:[step.name]` inside a `v-for` to provide all of the slot content.

When might you need something like this?

I can imagine one use case for a complex form generated dynamically. Or a wizard with multiple steps, where each step is a unique component.

93. Computed Props in Your Template: v-memo

Vue 3.2 gives you fine-grained control over template re-rendering using `v-memo`:

```
<div v-memo="[varA, varB, varC]">
  <!-- ... -->
</div>
```

This works much the same as a computed prop does. An element with `v-memo` is only re-rendered when the array changes, but otherwise, it caches (or memoizes) the result.

When it's used with `v-for` you can selectively re-render only the parts of a list that have changed:

```
<div
  v-for="item in list"
  :key="item.id"
  v-memo="[item.id === selected]"
>
  <!-- ... -->
</div>
```

Here, we only update the nodes that go from selected to unselected or vice versa. Much faster if you're dealing with extremely long lists!

But since Vue is already so efficient with re-renders, you shouldn't need to use `v-memo` often.

It's definitely a helpful tool to help you get more performance — when you really need it.

Check out the docs for v-memo.

94. Another use for the template tag

The `template` tag can be used anywhere inside your template to better organize code.

I like to use it to simplify `v-if` logic and sometimes `v-for`, too.

In this example, we have several elements that all use the same `v-if` condition:

```
<template>
  <div class="card">
    <img src="imgPath" />
    <h3>
      {{ title }}
    </h3>
    <h4 v-if="expanded">
      {{ subheading }}
    </h4>
    <div
      v-if="expanded"
      class="card-content"
    >
      <slot />
    </div>
    <SocialShare v-if="expanded" />
  </div>
</template>
```

It's a little clunky and not initially obvious that a bunch of these elements are being shown and hidden together. But, on a larger, more complicated component, this could become a catastrophic nightmare!

But we can fix that.

We can use the `template` tag to group these elements. Then lift the `v-if` on to the `template` tag itself:

```html
<template>
  <div class="card">
    <img src="imgPath" />
    <h3>
      {{ title }}
    </h3>
    <template v-if="expanded">
      <h4>
        {{ subheading }}
      </h4>
      <div class="card-content">
        <slot />
      </div>
      <SocialShare />
    </template>
  </div>
</template>
```

Now we have something much easier to read. And it's much easier to understand what's going on at a glance!

95. When should you use v-if?

Instead of using `v-if`, it's sometimes more performant to use `v-show` instead:

```
<ComplicatedChart v-show="chartEnabled" />
```

When `v-if` is toggled on and off, it will create and destroy the element completely. Instead, `v-show` will create the element and leave it there, hiding it by setting its style to `display: none`.

Doing this can be much more efficient if the component you're toggling is expensive to render.

On the flip side, if you don't need that expensive component immediately, use `v-if` so that it will skip rendering it and load the page just a bit faster.

Chapter 7

All the Others

These didn't fit neatly in another category, but they're still useful!

96. Check Vue's version

Did you know that you can easily check the version of Vue at runtime?

```js
import { version } from 'vue';

if (version.split('.')[0] === '2') {
  console.log('Uh, this app is gonna crash.');
  console.log('Upgrade to Vue 3!');
}
```

97. Using two script blocks

The `<script setup>` sugar in Vue 3 is a really nice feature, but did you know you can use it *and* a regular `<script>` block?

```
<script setup>
  // Composition API
  import { ref } from 'vue';
  console.log('Setting up new component instance');
  const count = ref(0);
</script>

<script>
  // ...and the options API too!
  export default {
    name: 'DoubleScript',
  };
</script>
```

This works because the `<script setup>` block is compiled into the component's `setup()` function.

There are a few reasons why you might want to do this:

- **Use the options API** — not everything has an equivalent in the composition API, like `inheritAttrs`.
- **Run setup code one time** — because `setup()` is run for *every* component, if you have code that should only be executed once, you can't include it in `<script setup>`. You can put it inside the regular `<script>` block, though.
- **Named exports** — sometimes, it's nice to export multiple things from one file, but you can only do that with the regular `<script>` block.

Check out the docs for more info

98. Define multiple components in a single file

Every now and then, you just need a small component, one that's not worth an entirely new file:

```
// A small, secondary component
const SmallComponent = {
  // Create a component like you normally would
  data() {
    //...
  },
  computedProps: {
    //...
  },

  // Use the template property instead of a template block
  template: `
    <div>Hello!</div>
  `
};

// Your main component
export default {
  components: { SmallComponent },
  // ...
};
```

This is perfect for reusing code within a component where a `v-for` doesn't work.

However, if the code is more complex or is likely to be used by other components, making a proper reusable component is the best way to go.

Note: You can get proper syntax highlighting of the HTML string using this VSCode extension.

99. Script setup

Lindsay Wardell
Podcaster and Engineer at NoRedInk

Not only is using `<script setup>` the recommended approach for Vue 3, but it reduces a lot of boilerplate code.

Instead of writing this:

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

export default {
  setup() {
    const name = ref('')
    const isNamePresent = computed(() => name.value.length > 0)

    function submitName() {
      console.log(name.value)
    }

    return {
      name,
      isNamePresent,
      submitName
    }
  }
}
</script>
```

With `<script setup>` it becomes:

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

const name = ref('')
const isNamePresent = computed(() => name.value.length > 0)

function submitName() {
  console.log(name.value)
}
</script>
```

First, we added the word "setup" to our script tag, enabling this new mode for writing Vue components.

Second, we took our code from within the `setup` function and replaced our existing exported object with just our code.

Note that *everything* declared within the script tags is available in your component's template. This includes components, non-reactive variables or constants, and utility functions or other libraries.

100. Performance tracing

Vue allows you to do performance tracing to help you debug any performance issues:

```
const app = createApp({});
app.config.performance = true;
```

Once you do this, you can use the official Vue Devtools to debug your app's performance.

101. Multi-file single-file components

Here's a little-known feature of SFC.

You can import files just like you would with a regular HTML file:

```
<!-- A "single" file component -->
<template src="./template.html"></template>
<script src="./script.js"></script>
<style scoped src="./styles.css"></style>
```

This can come in really handy if you need to share styles, docs, or anything else. Also perfect for that super long component file that's wearing out your finger from all the scrolling...

Here's a working demo of it in action: https://codesandbox.io/s/interesting-rosalind-9wwmr?file=/src/components/HelloWorld.vue

102. Get rid of the double curly braces

You can configure the Vue compiler to use different delimiters instead of the default `{{` and `}}`.

```
<template>
  <span>|| isBlue ? 'Blue!' : 'Not blue' ||</span>
</template>
```

This allows you to avoid any conflicts with server-side frameworks:

```
<template>
  <span>${ isBlue ? 'Blue!' : 'Not blue' }</span>
</template>
```

This can be done through the compiler options. Depending on how your project is set up, these options are passed to either `vue-loader`, `vite`, or the in-browser template compiler.

103. A better way to handle errors (and warnings)

You can provide a custom handler for errors and warnings in Vue:

```
// Vue 3
const app = createApp(App);
app.config.errorHandler = (err) => {
  alert(err);
};

// Vue 2
Vue.config.errorHandler = (err) => {
  alert(err);
};
```

Bug tracking services like Bugsnag and Rollbar hook into these handlers to log errors, but you can also use them to handle errors more gracefully for a better UX.

For example, instead of the application crashing if an error is unhandled, you can show a full-page error screen and get the user to refresh or try something else.

In Vue 3 the error handler only works on template and watcher errors, but the Vue 2 error handler will catch almost everything. The warning handler in both versions only works in development.

I created a demo showing how this works. It uses Vue 3, but Vue 2 works nearly the same:

Error Handler Demo

104. Aria roles you didn't know you needed

Aria roles are used to tell a screenreader what an element is for.

This is really important when the native HTML element just doesn't exist (eg. roles like `toolbar` and `alert`) or when you're using a different HTML element for design or technical reasons (eg. wrapping a `radio` button to style it).

But please, remember that you should always use the semantic element where you can. This is always the best and most effective solution.

There are six different categories of aria roles:

1. Widget - roles like `button`, `checkbox`, `separator`, `tab`, or `scrollbar`
2. Composite - roles like `combobox` and `listbox` (these are for dropdown menus), `radiogroup`, or `tree`
3. Document structure - this includes `article`, `presentation`, `figure`, `feed`, and `directory`
4. Landmark - `banner`, `main`, `navigation`, and `region` are roles in this category
5. Live region - `alert`, `log`, `marquee`, and `status` are roles that might update with real-time information
6. Window - `alertdialog` and `dialog` are the only two roles in this category

You can check out the full list here: https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Techniques#roles

105. What are all these loops for?

I always forget this, so this tip is mostly for me — hopefully, I won't have to keep looking this up!

We have 3 types of `for` loops in Javascript:

1. `for … in`
2. `for … of`
3. `for`

But how do you know which one to use?

For iterating through **properties** of an object, use `for … in`:

```
const numbers = {
  'one': 1,
  'two': 2,
  'three': 3,
};

// We get the properties of the object, not the values
for (const number in numbers) {
  console.log(number);
}

// Prints: 'one' 'two' 'three'
```

Items in a **list** (also called an iterable object) like an Array or Set, we use `for … of`:

```
const numbers = ['one', 'two', 'three'];

// We get each of the elements in the Array
for (const number of numbers) {
  console.log(number);
}

// Prints: 'one' 'two' 'three'
```

You *can* use `for … in` with an Array since all the indices are just the object's properties. But you may not get them in the correct order, and you'll also get any other properties the Array has :/

And you know how to use a regular old `for` loop, which lets you have a lot more control with some extra typing.

106. Reducing objects

The `reduce` function is excellent for converting arrays into objects, but it can be intimidating.

If we have a bunch of items that all have an `id`:

```
{
  id,
  //...
}
```

We can reorganize the array into an object, where the key for each item is the item's `id`:

```
const obj = arr.reduce((prev, next) => {
  // Grab the id from the item
  const { id } = next;

  // Add the item to our object
  prev[id] = next;

  // Return the object so we can add more items
  return prev;
}, {});
```

You get an object that looks like this:

```
{
  'id1': { id: 'id1', ... },
  'id2': { id: 'id2', ... },
  'id3': { id: 'id3', ... },
}
```

If you want to group all objects in an array by a specific property, you can do something very similar:

```
const obj = arr.reduce((prev, next) => {
  // Grab the property from the item that we want to group by
  const { prop } = next;

  // Add a new array to the object if this is the first one
  // with this value
  if (prev[prop] === undefined) {
    prev[prop] = [];
  }

  // Add our item to this array
  prev[prop].push(next);

  // Return the object so we can add more items
  return prev;
}, {});
```

Our final object looks like this:

```
{
  'prop1': [
    { prop: 'prop1', ... },
    { prop: 'prop1', ... },
    { prop: 'prop1', ... },
  ],
  'prop2': [
    { prop: 'prop2', ... },
    { prop: 'prop2', ... },
    { prop: 'prop2', ... },
  ],
}
```

107. Looping over a range in Vue

The `v-for` directive allows us to loop over an Array, but it also let's us loop over a range:

```
<template>
  <ul>
    <li v-for="n in 5">Item #{{ n }}</li>
  </ul>
</template>
```

This will render out:

- Item #1
- Item #2
- Item #3
- Item #4
- Item #5

When we use `v-for` with a range, it will start at 1 and end on the specified number.

108. My favourite git commands

Here are a few of my favourite git commands (is it weird to have favourite git commands?):

I'm often jumping back and forth between different branches, and typing is annoying:

```
# Checkout the previous branch
git checkout -
```

Sometimes I add new files, then realize I don't actually need them:

```
# Remove any files not tracked by git
git clean
```

Or I completely broke everything and need to start over:

```
# Undo all changes to git and the working directory,
# going back to the most recent commit
git reset --hard
```

Github takes all of the commits on your PR branch and combines them into a single one when you merge. But sometimes you want to merge a branch, and you aren't ready for a PR just yet:

```
# Squash all commits from a branch into one commit
git merge --squash <branch>
git commit
```

109. The picture element

The `<picture>` element lets us provide many image options for the browser, which will then decide what the best choice is:

```html
<picture>
  <!-- You can have as many source tags as you want -->
  <!-- (or none at all!) -->
  <source srcset="big-image.png" media="(min-width: 1024px)">
  <source srcset="bigger-image.png" media="(min-width: 1440px)">
  <source srcset="biggest-image.png" media="(min-width: 2048px)">

  <!-- One img tag is required to actually display the image -->
  <!-- and is used as the default choice -->
  <img src="regular-image.png">
</picture>
```

You can provide different options based on screen size, resolution, and supported image formats.

The mdn docs have more info on this element.

110. Stacking contexts

If you've ever run into an issue with `z-index` not working as you expect, there's a good chance it's because of stacking contexts.

The browser will stack elements based on their order in the DOM and their `z-index`. But it also groups elements into *stacking contexts*. These are groups of elements that the browser treats as a single unit.

If two elements are in different stacking contexts, adjusting their `z-index` *will not* change how they stack. You have to adjust how their stacking contexts are stacking:

```html
<body>
  <!-- First stacking context -->
  <div class="stacking-context">
    <div id="a"></div>
    <div id="b"></div>
  </div>
  <!-- Second stacking context -->
  <div class="stacking-context">
    <div id="c"></div>
    <div id="d"></div>
  </div>
</body>
```

```html
<style>
  /* These styles won't change anything */
  #a { z-index: 1; }
  #c { z-index: 2; }
  #b { z-index: 3; }
  #d { z-index: 4; }
</style>
```

Because the second stacking context is above the first, we cannot get `#a` to be above `#c`. The stacking contexts overrule no matter how hard we try with `z-index` or anything else.

However, if we change the `z-index` of the stacking contexts, we can get `# a` and `#b` to appear above `#c` and `# d`.

If we wanted to order them `#a`, `#c`, `#b`, `#d`, we'd have to change the HTML structure and move them all in to the same stacking context:

```
<body>
  <!-- A regular div -->
  <div>
    <div id="a"></div>
    <div id="b"></div>
  </div>
  <!-- Nothing special about this div -->
  <div>
    <div id="c"></div>
    <div id="d"></div>
  </div>
</body>
```

```
<style>
  /* These will change the visual hierarchy */
  #a { z-index: 1; }
  #c { z-index: 2; }
  #b { z-index: 3; }
  #d { z-index: 4; }
</style>
```

You know a bit about stacking contexts, but what causes them, and how can you control them?

Unfortunately, the rules for creating them are not that straightforward, but well worth learning.

111. Blockquotes

This element is used for quotes outside of the main flow of an article.

> **Like this quote. Most browsers will indent this automatically, and most websites will add extra styling.**

While you can use a `div` with some CSS, the `<blockquote>` element is the semantically correct way of doing this.

In Markdown, you can use `>` to get a blockquote.

112. UI states to get right

When building a UI, there are many different states that you need to consider:

- **Normal** — Sometimes called the "happy path," this is when things are working as expected. For example, in an email client, you'd show some read emails, some unread emails, and maybe a few that are in the "spam" folder.
- **Loading** — Your UI has to do something while getting the data, right? A couple tricks:
    1. Use a computed prop to combine multiple loading states — you don't want spinners all over the page.
    2. Wait about `200ms` before showing a spinner. If the data loads before that, it feels faster than if you quickly flash the loading spinner on and then off again.
- **Error** — Things *will* go wrong, and you need to handle that gracefully. Effectively communicating problems to users to help them get unstuck is very tricky (don't make me guess the password requirements!). Hopefully, you have a good UX designer. **Empty** — What happens when you have no emails to read, have completed all your tasks, or haven't uploaded any videos yet? A chart showing the "Last 30 Days" of data will probably look weird with no data.
- **Partial Data** — Often similar to the empty state, but your big table with filtering and sorting also needs to work with only two rows of data. The list of emails shouldn't break with only one email in it.
- **Lots of data** — Okay, now you have 1294 unread emails. Does your UI break? Maybe that infinite scrolling doesn't make as much sense as when there were only 42 emails.

113. Folder Structure: Keep It Flat

Markus Oberlehner
Exploring how to build software that lasts

When starting a new project or refactoring an existing one, the question often arises:

How do you set up the project's directory structure?

My first advice is to keep the folder hierarchy as flat as possible for as long as possible:

```
/my/project/src
├─ components
│  ├─ BaseButton.vue
│  ├─ BaseCard.vue
│  ├─ BaseLink.vue
│  ├─ ...
│  ├─ ProductCart.vue
│  ├─ ProductDetail.vue
│  ├─ ...
│  ├─ TheFooter.vue
│  ├─ TheHeader.vue
│  └─ ...
├─ services
│  ├─ product.js
│  └─ ...
├─ utils
│  ├─ as-array.js
│  ├─ format-price.js
│  └─ ...
└─ ...
```

When things in your app get more complicated, I recommend gradually increasing the complexity of the folder structure as needed.

If the utterly flat directory structure does not work for you anymore, start by adding only one additional `base` directory:

```
/my/project/src
├─ components
│  ├─ base
│  │  ├─ BaseButton.vue
│  │  ├─ BaseCard.vue
│  │  ├─ BaseLink.vue
│  │  └─ ...
│  ├─ ProductCart.vue
│  ├─ ProductDetail.vue
│  ├─ ...
│  ├─ TheFooter.vue
│  ├─ TheHeader.vue
│  └─ ...
├─ services
│  ├─ product.js
│  └─ ...
├─ utils
│  ├─ as-array.js
│  ├─ format-price.js
│  └─ ...
└─ ...
```

The base directory contains all the generic and highly reusable components of your app.

Apart from keeping your component directory a little tidier, moving all base components into a separate directory solidifies their status as generic and reusable components.

114. The smallest possible PR

A very underrated skill that will make you a more productive dev is creating small, concise PRs.

(And by small, I don't just mean lines of code, but also the risk. Updating from Vue 2 to Vue 3 could be a single line, but a huge PR.)

As the size of the PR grows, the time it takes to get it merged grows exponentially. This is because it takes much longer to review:

- **Procrastination** — Bigger changes require more focus and time to go through instead of being able to quickly look through them and approve.
- **Higher risk** — The more that changes, the higher chance that you broke something, so reviewers will be more meticulous.
- **Less focused** — Larger PRs often contain more unrelated changes, making it more difficult to understand what you're actually trying to do.
- **More changes** — The back and forth to get it right can take a lot longer if the PR is big.

I'm not saying I'm great at this yet. But I've noticed that I'm *much* more productive when I create small, focused PRs that can be quickly reviewed.

It turns out that PRs that are easy to review are also easier to create.

The hard part (the *really* hard part) is learning how to break your work down into small pieces that you can deliver one at a time.

115. The best solution is the worst solution

And the "worst" solution is the best.

(as long as it works)

The most valuable thing I've learned as a software developer is to just ship the code. Waiting for perfection only slows you down and doesn't usually make the product (or the code) any better.

If you find yourself thinking:

"This code is kind of hacky"

"I wish I had a more elegant solution to this"

Stop and ask yourself why. Once something is working, refining and perfecting the code too much isn't worth it.

In my experience, the "best" solution can often:

- Take 10x longer for no noticeable improvement
- Be far too clever, which actually makes it *worse*
- Be replaced weeks or months later

So don't worry too much about finding the "best" solution. A good one is usually better.