Proven Patterns for Building Production-Grade Vue Apps

Ben Hong

https://www.bencodezen.io

VueConfUS 2025

guest.wireless.net.usf.edu

📡 **WiFi: USF-Guest**

🔑 **PASSWORD**: **Use SMS option (if international, let me know!)**

👋 Introductions

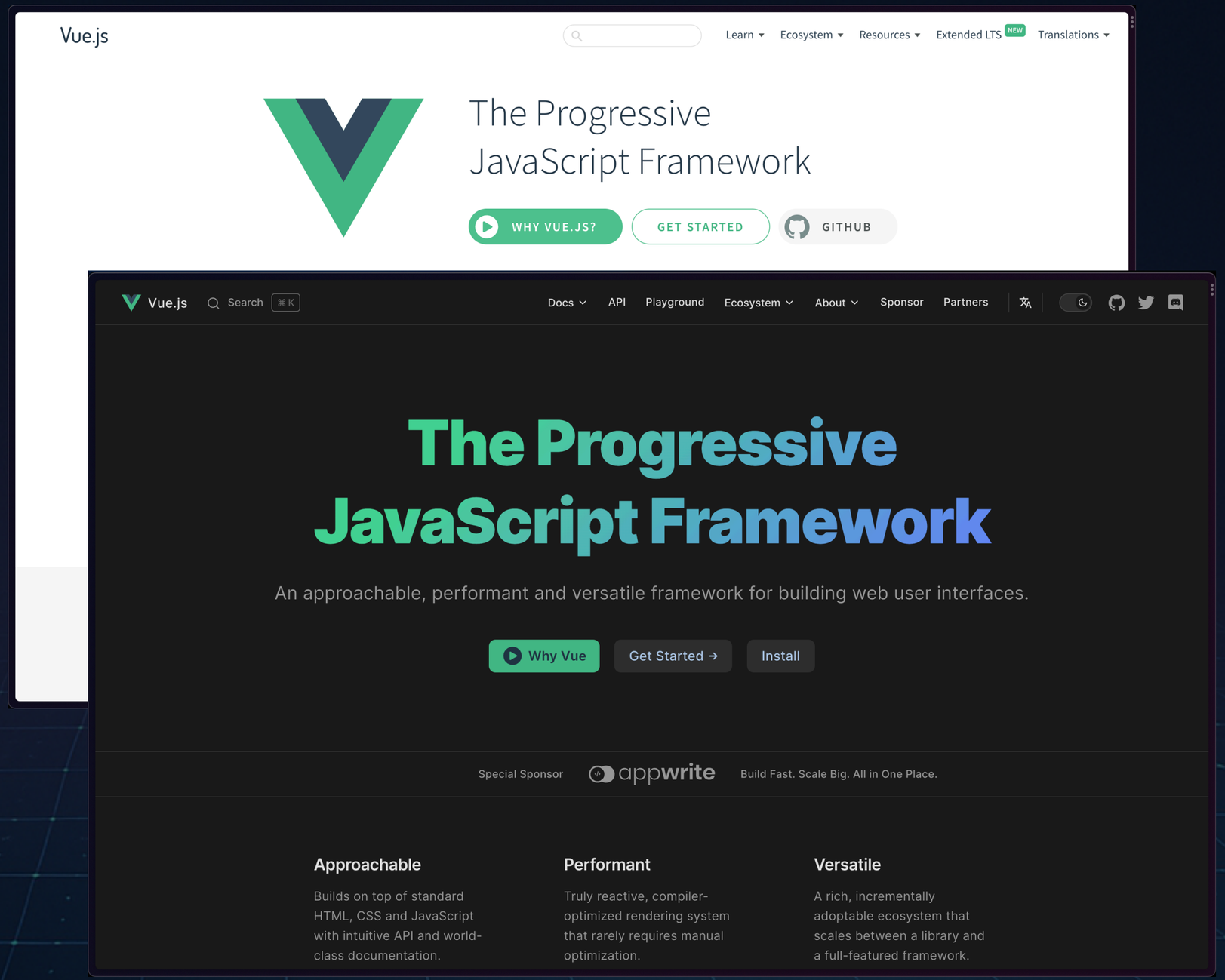
Who am I?


UNIVERSAL'S
ISLANDS OF ADVENTURE
QUKSILVER
JURASSIC PARK
RIVER ADVENTURE
UNIVERSAL'S
ISLANDS OF ADVENTURE™
November 12, 2017
TM & © Universal Studios and Amblin

Ben Hong

Vue Core Team

@bencodezen



Who are you?

Who are you?

Fill out the following survey:

https://form.typeform.com/to/VC762sX9

SCAN ME

Get to know your neighbors

Name

Job / Title

What is their top goal for this workshop?

Get to know your neighbors

📊 Survey results! 📈

Setup

Why are we here?

What will we cover?

What will we cover?

- Languages
- Components
- Routing
- Reusability & Composition
- State Management
- Testing
- Tooling
- Fostering Change

🗓️ Schedule

Time Slot	Event
9:00AM - 9:30AM	Introductions
9:30AM - 10:30AM	Languages
10:30AM - 10:40AM	☕ Short Break 🍵
10:40AM - 12:00PM	Components
12:00PM - 1:00PM	🥪🥗 Lunch 🍟🥤
1:00PM - 2:30PM	Routing + Reusability & Composition
2:30PM - 2:40PM	🍪 Short Break 🍫
2:40PM - 4:00PM	R&C, State Management & Testing
4:00M - 4:10PM	🧁 Short Break 🍰
4:10PM - 4:45PM	Tooling & Fostering Change
4:45PM - 5:00PM	Final Thoughts + Q&A

📚 Resources

📚 Resources

Proven Patterns Workshop Repo

https://github.com/bencodezen/proven-patterns-workshop

SCAN ME

📚 Resources

Your Projects

I encourage you to pull from your experience
to make the most of this workshop.

📚 Resources

Me

I’m here to talk about your codebase
and any ideas and questions you’re having.

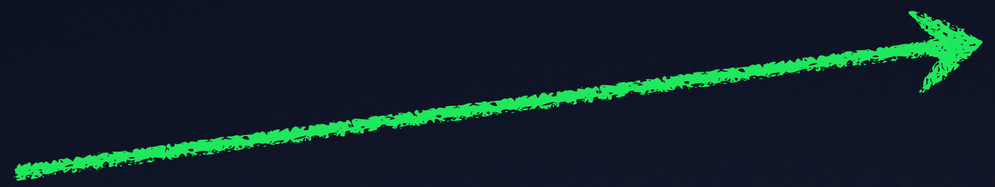
🔀 Workshop Format

🔀 Workshop Format

1. Learn
2. Question
3. Apply

🔀 Workshop Format

1. Learn



Explanations
Examples
Stories

2. Question

Clarifications
What-ifs

3. Apply

Code
Experiment
Pairing

🙋‍♂️ Participation Tips 🙋‍♀️

🙋‍♂️ Participation Tips 🙋‍♀️

This is a **safe place** to learn from one another

https://vuejs.org/about/coc.html

🙋‍♂️ Participation Tips 🙋‍♀️

Questions are **welcome**!

Discussions are how we can make the most of our time together.

🙋‍♂️ Participation Tips 🙋‍♀️

All slides and examples will be **public**.

No need to hurriedly copy down everything!

🙋‍♂️ Participation Tips 🙋‍♀️

Please **no recording**.

Out of respect for you and your participants' privacy

🙋‍♂️ Participation Tips 🙋‍♀️

If you need a **break**, please take one!

Or if you need to leave for any other reason, feel free to.

🙋‍♂️ Participation Tips 🙋‍♀️

All code is **compromise**.

I encourage you to question and/or disagree.

Your opinion and experience matter.

Choose what works best for you and your team.

🙋‍♂️ Participation Tips 🙋‍♀️

Raise your hand for **questions** at any time.

Discussions are how we can make the most of our time together.

Languages

HelloVueConfUS.vue

```vue
<template>
  <h1>Hello VueConfUS 2025!</h1>
</template>

<script>
export default {
  data: () => ({
    location: 'Tampa, FL'
  }),
}
</script>

<style>
h1 {
  background: white;
  color: blue;
}
</style>
```

LANGUAGES

HTML

TECHNIQUE

Template vs Render Function

TECHNIQUE

Template

MyFirstComponent.vue

```vue
<template>
  <h1 class="title">Hello VueConfUS!</h1>
</template>

<script>
export default {
  // ...
}
</script>

<style>
/* My Custom Styles */
</style>
```

TECHNIQUE

Render Function

MyFirstComponent.vue

```
<template>
  <h1 class="title">Hello VueConfUS!</h1>
</template>

<script>
export default {
  // ...
}
</script>

<style>
/* My Custom Styles */
</style>
```

TECHNIQUE

Render Function

MyFirstComponent.vue

```
<template>
  <h1 class="title">Hello VueConfUS!</h1>
</template>

<script>
import { h } from 'vue'

export default {
  // ...
}
</script>

<style>
/* My Custom Styles */
```

TECHNIQUE

Render Function

MyFirstComponent.vue

```vue
<template>
  <h1 class="title">Hello VueConfUS!</h1>
</template>

<script>
import { h } from 'vue'

export default {
  render() {
    return h()
  }
}
</script>

<style>
```

TECHNIQUE

Render Function

MyFirstComponent.vue

```vue
<script>
import { h } from 'vue'

export default {
  render() {
    return h('h1', { class: 'title' }, 'Hello VueConfUS!')
  }
}
</script>

<style>
/* My Custom Styles */
</style>
```

TECHNIQUE

Template vs Render Function

- Templates are the most declarative way to write HTML and is recommended as the default
- Render functions are a valid alternative to templates that are great for programmatic generation of markup (e.g., libraries, etc.)

BEST PRACTICE

All HTML should exist in .vue files as a <template> or render function.

BEST PRACTICE

All HTML should exist in .vue files as a <template> or render function.

- A benefit of doing this is that Vue has an opportunity to parse it before the browser does

BEST PRACTICE

All HTML should exist in .vue files as a <template> or render function.

- A benefit of doing this is that Vue has an opportunity to parse it before the browser does
- This allows for developer experience improvements such as:
  - Self-closing tags

BEST PRACTICE

All HTML should exist in .vue files as a <template> or render function.

- Self-closing tags

```
<span class="fa fa-info"></span>
```

BEST PRACTICE

All HTML should exist in .vue files as a <template> or render function.

- Self-closing tags

```
<span class="fa fa-info" />
```

BEST PRACTICE

All HTML should exist in .vue files as a <template> or render function.

- Self-closing tags

```
<BaseIcon
  name="info"
  size="large"
  text="Information Icon"
></BaseIcon>
```

BEST PRACTICE

All HTML should exist in .vue files as a <template> or render function.

- Self-closing tags

```
<BaseIcon
  name="info"
  size="large"
  text="Information Icon"
/>
```

BEST PRACTICE

All HTML should exist in .vue files as a <template> or render function.

- A benefit of doing this is that Vue has an opportunity to parse it before the browser does
- This allows for developer experience improvements such as:
  - Self-closing tags
  - Easy enhancement path if needed

TECHNIQUE

Inspect complex data in the UI

TECHNIQUE

Inspect complex data in the UI

MyFirstComponent.vue

```vue
<script setup lang="ts">
import { ref } from 'vue'

const data = ref({
  postId: 1,
  id: '64456ca6-8927-46d6-9ee0-6afa99e508d2',
  name: 'Chidi Anagonye',
  email: 'mystomachhurts@thegoodplace.com',
  body: 'Principles aren’t principles when you pick and choose
  when you’re gonna follow them.',
})
</script>

<template>
  <pre>{{ data }}</pre>
</template>
```

TECHNIQUE

```
v-bind="{ … }"
v-on="{ … }"
```

When working with props and/or events consider...

```
<VueMultiselect
  v-bind:options="options"
  v-bind:value="value"
  v-bind:label="label"
  v-on:change="parseSelection"
  v-on:keyup="registerSelection"
  v-on:mouseover="registerHover"
/>
```

When working with props and/or events consider...

```
<VueMultiselect
  v-bind="{
    options: options,
    value: value,
    label: label
  }"
  v-on:change="parseSelection"
  v-on:keyup="registerSelection"
  v-on:mouseover="registerHover"
/>
```

When working with props and/or events consider...

```
<VueMultiselect
  v-bind="{
    options: options,
    value: value,
    label: label
  }"
  v-on="{
    change: parseSelection,
    keyup: registerSelection,
    mouseover: registerHover
  }"
/>
```

When working with props and/or events consider...

```
<VueMultiselect
  v-bind="{
    options,
    value,
    label
  }"
  v-on="{
    change: parseSelection,
    keyup: registerSelection,
    mouseover: registerHover
  }"
/>
```

When working with props and/or events consider...

```
<VueMultiselect
  v-bind="vmsProps"
  v-on="{
    change: parseSelection,
    keyup: registerSelection,
    mouseover: registerHover
  }"
/>
```

When working with props and/or events consider...

```
<VueMultiselect
  v-bind="vmsProps"
  v-on="vmsEvents"
/>
```

BEST PRACTICE

Don’t use v-if with v-for.

Questions?

LANGUAGE

CSS

BEST PRACTICE

Limit global styles to App component.

BEST PRACTICE

Scope all component styles.

MyRedText.vue

```vue
<template>
  <p class="red">
    This should be red!
  </p>
</template>

<style>
.red {
  color: red;
}
.bold {
  font-weight: bold;
}
</style>
```

TECHNIQUE

Scoped Styles

MyRedText.vue

```vue
<template>
  <p class="red">
    This should be red!
  </p>
</template>

<style>
.red {
  color: red;
}
.bold {
  font-weight: bold;
}
</style>
```

MyRedText.vue

```vue
<template>
  <p class="red">
    This should be red!
  </p>
</template>

<style scoped>
.red {
  color: red;
}
.bold {
  font-weight: bold;
}
</style>
```

MyRedText.vue

```vue
<template>
  <p class="red" data-v57s8>
    This should be red!
  </p>
</template>
```

```vue
<style>
.red[data-v57s8] {
  color: red;
}
.bold[data-v57s8] {
  font-weight: bold;
}
</style>
```

TECHNIQUE

CSS Modules

MyRedText.vue

```vue
<template>
  <p class="red">
    This should be red!
  </p>
</template>

<style>
.red {
  color: red;
}
.bold {
  font-weight: bold;
}
</style>
```

MyRedText.vue

```vue
<template>
  <p class="red">
    This should be red!
  </p>
</template>

<style module>
.red {
  color: red;
}
.bold {
  font-weight: bold;
}
</style>
```

MyRedText.vue

```vue
<template>
  <p :class="$style.red">
    This should be red!
  </p>
</template>

<style module>
.red {
  color: red;
}
.bold {
  font-weight: bold;
}
</style>
```

MyRedText.vue

```vue
<template>
  <p :class="$style.red">
    This should be red!
  </p>
</template>

<style>
.MyRedText_red_3fj4x {
  color: red;
}
.MyRedText_bold_8fn3s {
  font-weight: bold;
}
</style>
```

MyRedText.vue

```vue
<template>
  <p class="MyRedText_red_3fj4x">
    This should be red!
  </p>
</template>

<style>
.MyRedText_red_3fj4x {
  color: red;
}
.MyRedText_bold_8fn3s {
  font-weight: bold;
}
</style>
```

TECHNIQUE

CSS Modules Exports

TECHNIQUE

CSS Modules Exports

```
<template>
  <p>Grid Padding: {{ $style.gridPadding }}</p>
</template>

<style module>
:export {
  gridPadding: 1.5rem;
}
</style>
```

TECHNIQUE

State Driven CSS

TECHNIQUE

State Driven CSS

```
<template>
  <h1 class=“title">Make my color dynamic!</h1>
</template>

<style scoped>
.title {
  color: red;
}
</style>
```

TECHNIQUE

State Driven CSS

```
<script>
export default {
  data: () => ({
    textColor: 'papayawhip'
  })
}
</script>

<template>
  <h1 class="title">Color me whatever you want!</h1>
  <input v-model="textColor" type="color" />
</template>

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

Questions?

LANGUAGE

JavaScript

DISCUSSION

JavaScript vs TypeScript

DISCUSSION

JavaScript vs TypeScript

- Majority of bugs encountered are **not** due to type violations
- TypeScript does **not** inherently guarantee type safety - it requires discipline
- Full type safety in a codebase can be a significant cost to a team in terms of productivity
- Most applications would benefit from better tests and code reviews

DISCUSSION

JavaScript vs TypeScript

- Progressive types can be added to a codebase with JSDoc comments
- If the application is in Vue.js 2, probably not worth upgrading to TypeScript
- Starting a new project with Vue.js 3 and the team is interested in trying it out TypeScript? Go for it!

Questions?

👩‍💻 CODE / EXPERIMENT 👨‍💻

In the playground

- Create a component using the render function
- Experiment with different CSS techniques (scoped, modules, state driven, etc.)
- Try writing some TypeScript!

In your app

- Refactor a template to use the render function instead.
- Refactor a component to use CSS modules
- Setup TypeScript in your project and add it to a component.

SHORT BREAK

Be back at 10:45AM!

COMPONENTS

BEST PRACTICE

Naming Components

Actual programming

Debating for 30 minutes on how to name a variable

BEST PRACTICE

Naming Components

Avoid single word components

~~Header.vue~~

~~Button.vue~~

~~Container.vue~~

BEST PRACTICE

Naming Components

AppPrefixedName.vue / **Base**PrefixedName.vue

Reusable, globally registered UI components.

`AppButton, AppModal, BaseDropdown, BaseInput`

ThePrefixedName.vue

Single-instance components where only 1 can be active at the same time.

`TheShoppingCart, TheSidebar, TheNavbar`

BEST PRACTICE

Naming Components

Tightly coupled/related components

TodoList.vue

TodoListItem.vue

TodoListItemName.vue

1. Easy to spot relation
2. Stay next to each other in the file tree
3. Name starts with the highest-level words

BEST PRACTICE

Naming Component Methods

BEST PRACTICE

Naming Component. Methods

Use descriptive names

❌ `onInput` ✅ `updateUserName`

Don’t assume where it will be called

```
updateUserName($event) {
  this.user.name = $event.target.value
}
```

❌ Wrong

```
updateUserName (newName) {
  this.user.name = newName
}
```

✅ Correct

BEST PRACTICE

Naming Component. Methods

Prefer destructuring over multiple arguments

```
updateUser (userList, index, value, isOnline) {
  if (isOnline) {
    userList[index] = value
  } else {
    this.removeUser(userList, index)
  }
}
```

❌ Not recommended

BEST PRACTICE

Naming Component. Methods

Prefer destructuring over multiple arguments

```
updateUser ({ userList, index, value, isOnline }) {
  if (isOnline) {
    userList[index] = value
  } else {
    this.removeUser(userList, index)
  }
}
```

✅ Recommended

BEST PRACTICE

When to Refactor Your Components

Premature optimization is the root of all evil (or at least most of it) in programming.

- Donald Knuth

PRINCIPLE

Data Driven Refactoring

PRINCIPLE

Data Driven Refactoring

Signs you need more components

- When your components are hard to understand
- You feel a fragment of a component could use its own state
- Hard to describe what what the component is actually responsible for

PRINCIPLE

Data Driven Refactoring

How to find reusable components?

- Look for v-for loops
- Look for large components
- Look for similar visual designs
- Look for repeating design interface fragments
- Look for multiple/mixed responsibilities
- Look for complicated data paths

Questions?

PRO TIP

SFC Code Block Order

MyFirstComponent.vue

```vue
<template>
  <h1>Hello VueConfUS 2024!</h1>
</template>


<script>
export default {
  // ...
}
</script>


<style>
/* My Custom Styles */
</style>
```

MyFirstComponent.vue

```vue
<template>
  <h1>Hello VueConfUS 2025!</h1>
</template>


<script>
export default {
  // ...
}
</script>


<style>
/* My Custom Styles */
</style>
```

MyFirstComponent.vue

```vue
<template>
  <h1>Hello VueConfUS 2025!</h1>
</template>


<script>
export default {
  // ...
}
</script>


<style>
/* My Custom Styles */
</style>
```

MyFirstComponent.vue

```vue
<template>
  <h1>Hello VueConfUS 2025!</h1>
</template>


<script>
export default {
  // ...
}
</script>


<style>
/* My Custom Styles */
</style>
```

MyFirstComponent.vue

```vue
<script>
export default {
  // ...
}
</script>

<template>
  <h1>Hello VueConfUS 2025!</h1>
</template>


<style>
/* My Custom Styles */
</style>
```

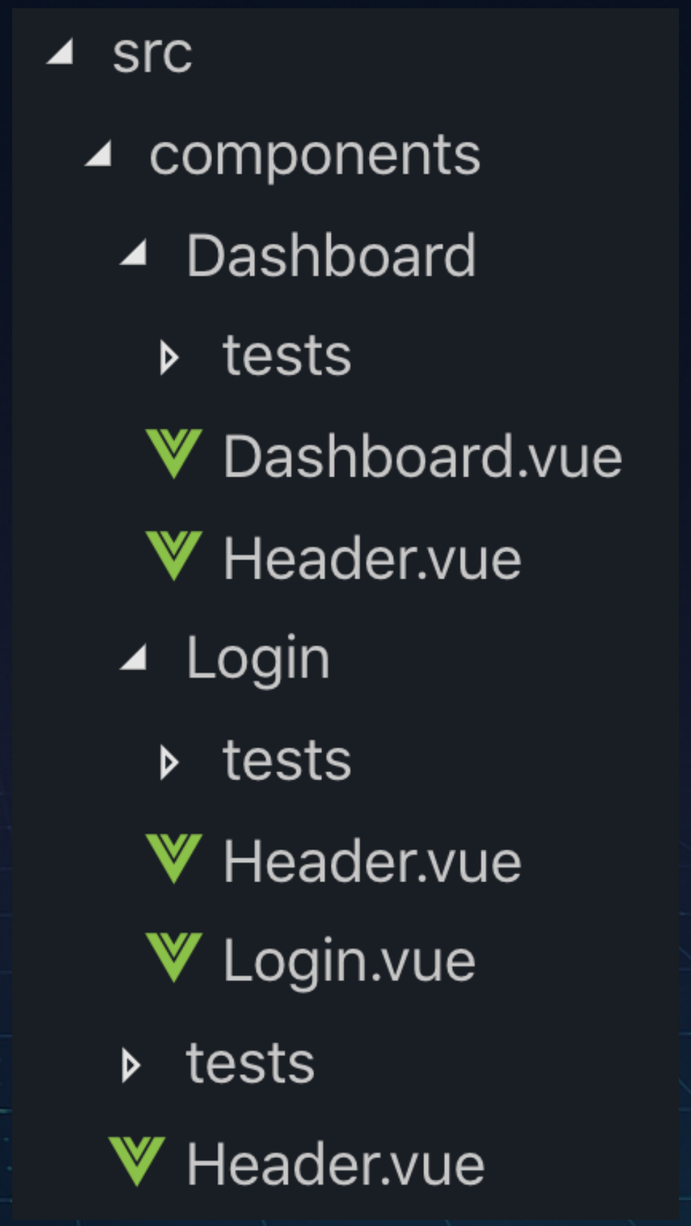
MyFirstComponent.vue

```vue
<script>
export default {
  // ...
}
</script>

<template>
  <h1>Hello VueConfUS 2025!</h1>
</template>

<style>
/* My Custom Styles */
</style>
```

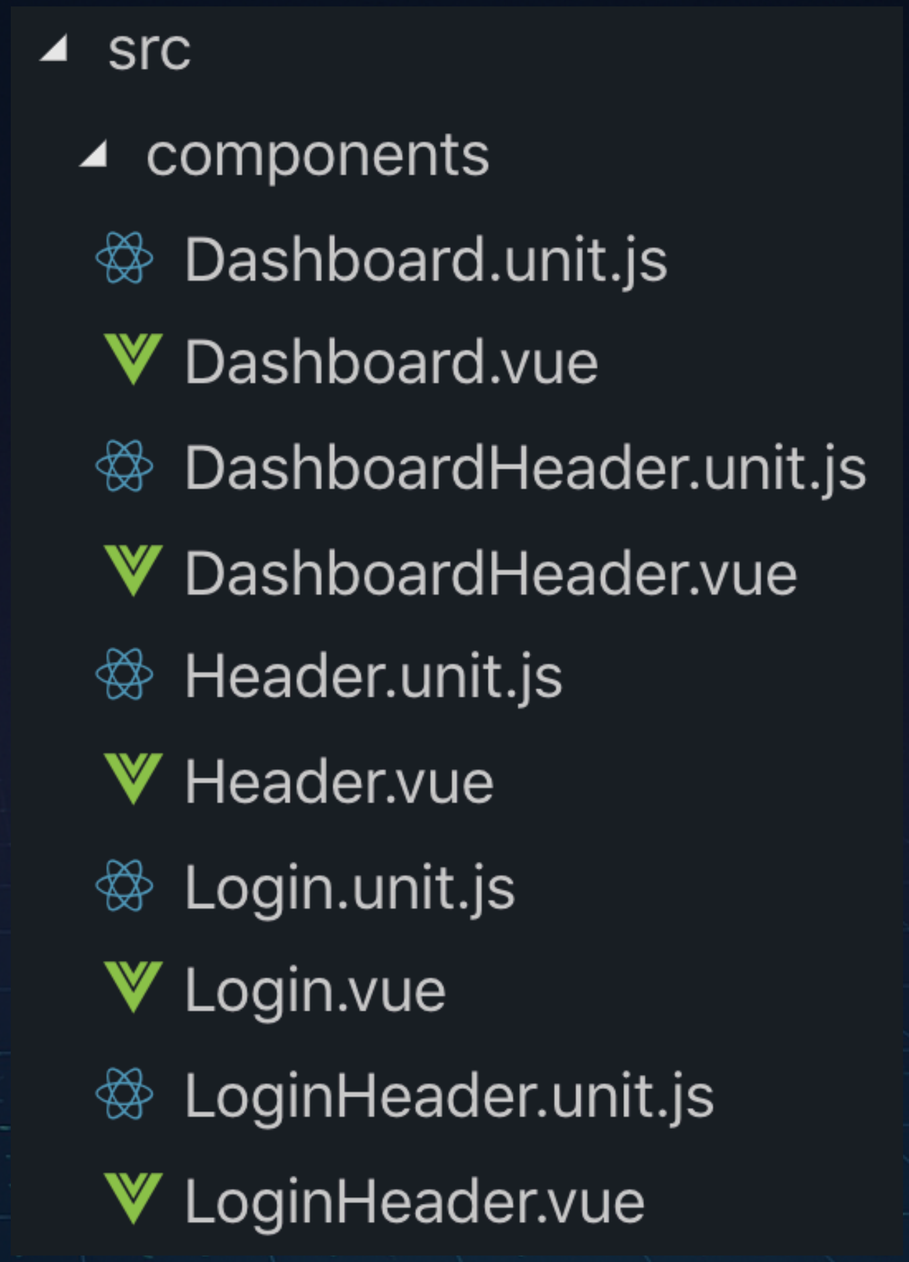
PRO TIP

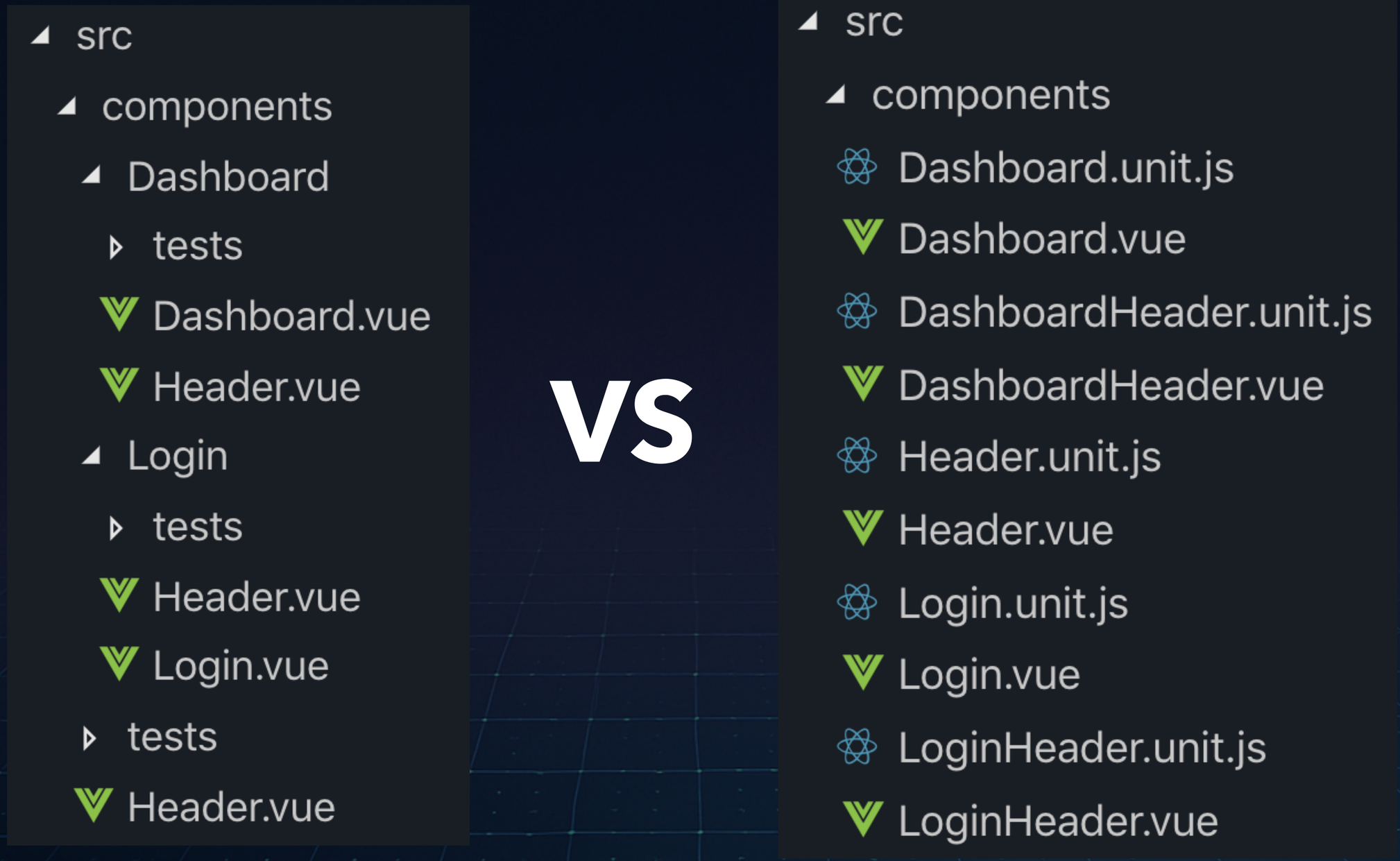
Component File Organization

Nested Structure



Flat Structure




src
components
Dashboard
tests
Dashboard.vue
Header.vue
Login
tests
Header.vue
Login.vue
tests
Header.vue
VS
src
components
Dashboard.unit.js
Dashboard.vue
DashboardHeader.unit.js
DashboardHeader.vue
Header.unit.js
Header.vue
Login.unit.js
Login.vue
LoginHeader.unit.js
LoginHeader.vue

PRO TIP

Component File Organization

Flat makes refactoring easier

No need to update imports if components move

Flat makes finding files easier

Folders often leads to lazily named files

because they don’t have to be unique

TECHNIQUE

Namespaced Components

```ts
// @/components/index.ts
export { default as List } from './TodoList.vue'
export { default as Item } from './TodoItem.vue'


// @/App.vue
<script setup lang="ts">
import * as Todo from '@/components/todo'
</script>

<template>
  <div>
    <Todo.List>
      <Todo.Item v-for="i in 5" :key="i"> Todo {{ i }} </Todo.Item>
    </Todo.List>
  </div>
</template>
```

PRO TIP

Auto import components

PRO TIP

Auto import components

No more multi-line component import statements…

```
import BaseButton from './components/BaseButton.vue'
import BaseIcon from './components/BaseIcon.vue'
import BaseInput from './components/BaseInput.vue'
```

Do this in a performative way with:

https://github.com/antfu/unplugin-vue-components

Questions?

TECHNIQUE

Props

NavItem.vue

```
<script>
export default {
  props: ['label']
}
</script>

<template>
  <li class="nav-item">
    <a :href="`/${label}`">
      {{ label }}
    </a>
  </li>
</template>
```

NavItem.vue

```
<script>
export default {
  props: ['label']
}
</script>

<template>
  <li class="nav-item">
    <a :href="`/${label}`">
      {{ label }}
    </a>
  </li>
</template>
```

Navbar.vue

```
<template>
  <ul>
    <NavItem label="Home" />
    <NavItem label="About" />
    <NavItem label="Contact" />
  </ul>
</template>
```

NavItem.vue

```
<script>
export default {
  props: ['label']
}
</script>

<template>
  <li class="nav-item">
    <a :href="`/${label}`">
      {{ label }}
    </a>
  </li>
</template>
```

Navbar.vue

```
<template>
  <ul>
    <NavItem label="Home" />
    <NavItem label="About" />
    <NavItem label="Contact" />
  </ul>
</template>
```

NavItem.vue

```
<script>
export default {
  props: ['label']
}
</script>
```

NavItem.vue

```
<script>
export default {
  props: {
    label: {
      type: String,
      required: true,
      default: 'Home'
    }
  }
}
</script>
```

NavItem.vue

```
<script>
export default {
  props: {
    label: {
      type: String,
      default: 'Home'
    }
  }
}
</script>
```

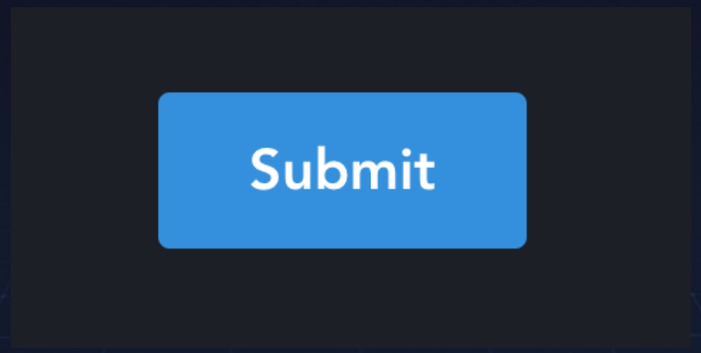
NavItem.vue

```
<script>
export default {
  props: {
    label: {
      type: String,
      default: 'Home',
      validator: (value) => {
        return ['Home', 'About'].indexOf(value) !== -1
      }
    }
  }
}
</script>
```

Let’s do a Coding Experiment

Task 1

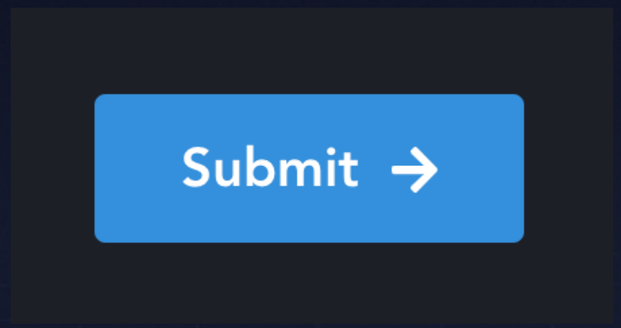
Create a button component that can display text specified in the parent component



Task 2

Allow the button to display an icon of choice

on the right side of the text

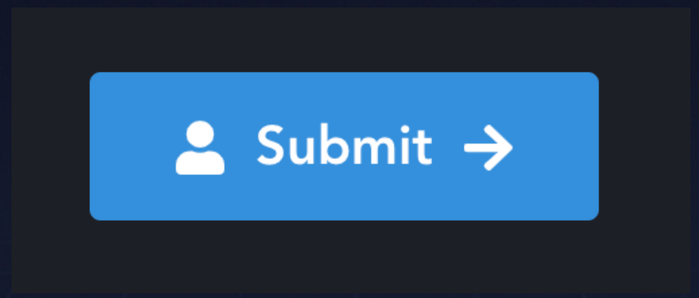



```
<AppIcon icon="arrow-right" class="ml-3"/>
```

This is the code responsible for displaying an arrow.

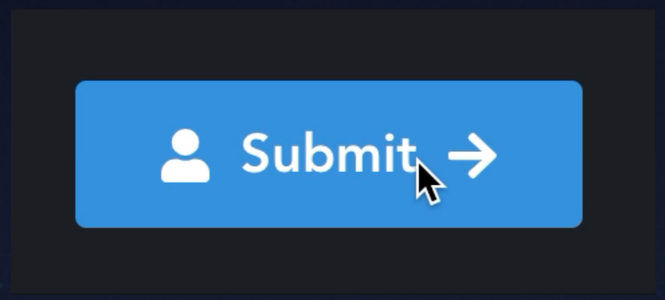
Task 3

Make it possible to have icons on either side or even both sides



Task 4

Make it possible to replace the content with a loading spinner

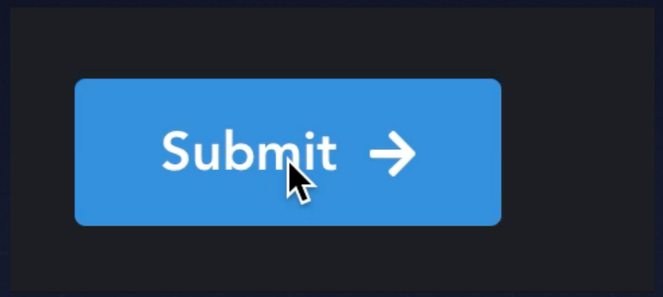



```
<PulseLoader color="#fff" size="12px"/>
```

This is the code responsible for displaying a spinner.

Task 5

Make it possible to replace an icon with a loading spinner



Possible solution

```
<template>
  <button type=“button" class=“nice-button“>
    {{ text }}
  </button>
</template>

<script>
export default {
  props: ['text']
}
</script>
```

```
<template>
  <button type=“button" class=“nice-button“>
    <PulseLoader v-if="isLoading" color="#fff" size="12px">
    <template v-else>
      <template v-if="iconLeftName">
        <PulseLoader v-if="isLoadingLeft" color="#fff" size="6px">
        <AppIcon v-else :icon=“iconLeftName"/>
      </template>
      {{ text }}
      <template v-if="iconRightName">
        <PulseLoader v-if="isLoadingRight" color="#fff" size="6px">
        <AppIcon v-else :icon=“iconRightName”/>
      </template>
    </template>
  </button>
</template>

<script>
export default {
  props: ['text', 'iconLeftName', 'iconRightName', 'isLoading',
'isLoadingLeft', 'isLoadingRight']
}
</script>
```


OMG
PROPS EVERYWHERE!

```
<template>
  <button type=“button" class=“nice-button“>
    <PulseLoader v-if="isLoading" color="#fff" size="12px">
    <template v-else>
      <template v-if="iconLeftName">
        <PulseLoader v-if="isLoadingLeft" color="#fff"
size="6px">
        <AppIcon v-else :icon=“iconLeftName"/>
      </template>
      {{ text }}
      <template v-if="iconRightName">
        <PulseLoader v-if="isLoadingRight" color="#fff"
size="6px">
        <AppIcon v-else :icon=“iconRightName”/>
      </template>
    </template>
  </button>
</template>

<script>
export default {
  props: ['text', 'iconLeftName', 'iconRightName', ‘isLoading’,
'isLoadingLeft', 'isLoadingRight']
}
</script>
```

Let’s call it the props-based solution

props-based solution

Is it wrong?

props-based solution

Is it wrong?

No.

It does the job.

props-based solution

Is it good, then?

props-based solution

Is it good, then?

Not exactly.

props-based solution

Problems

props-based solution

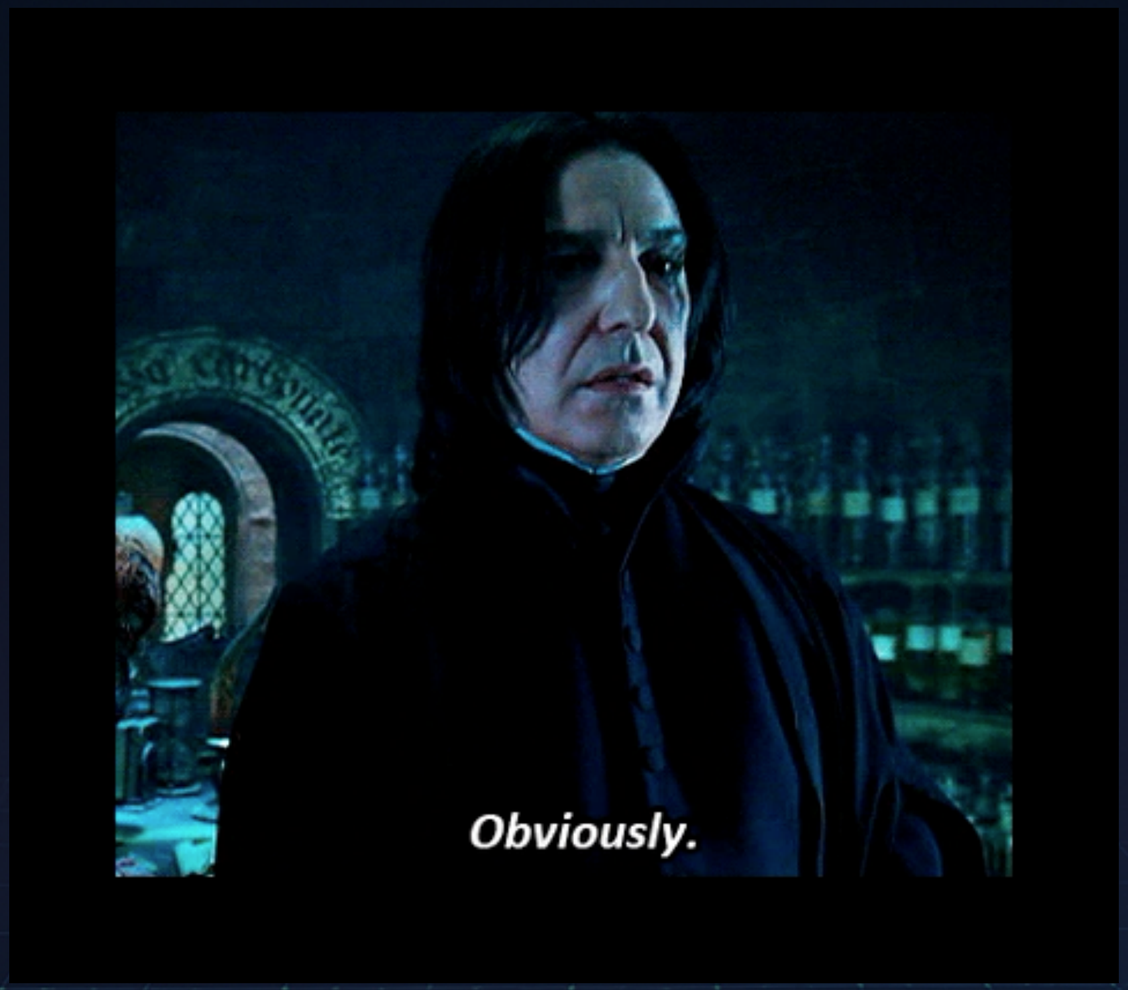
Problems

- New requirements increase complexity
- Multiple responsibilities
- Lots of conditionals in the template
- Low flexibility
- Hard to maintain

Is it good, then?

Not exactly.

Is there a better another alternative?



Is there a better another alternative?

Recommended solution

```
<template>
  <button type="button" class="nice-button">
    <slot />
  </button>
</template>
```

TECHNIQUE

Slots

```
<template>
  <button type=“button" class=“nice-button“>
    <PulseLoader v-if="isLoading" color="#fff" size="12px">
    <template v-else>
      <template v-if="iconLeftName">
        <PulseLoader v-if="isLoadingLeft" color="#fff"
size="6px">
        <AppIcon v-else :icon=“iconLeftName"/>
      </template>
      {{ text }}
      <template v-if="iconRightName">
        <PulseLoader v-if="isLoadingRight" color="#fff"
size="6px">
        <AppIcon v-else :icon=“iconRightName”/>
      </template>
    </template>
  </button>
</template>

<script>
export default {
  props: ['text', 'iconLeftName', 'iconRightName', ‘isLoading’,
'isLoadingLeft', 'isLoadingRight']
}
</script>
```

BaseButton.vue

```
<template>
  <button type="button" class="nice-button">
    <slot/>
  </button>
</template>
```

BaseButton.vue

```vue
<template>
  <button type="button" class="nice-button">
    <slot/>
  </button>
</template>
```

App.vue

```vue
<template>
  <BaseButton>
    Submit
  </BaseButton>
</template>
```

BaseButton.vue

```
<template>
  <button type="button" class="nice-button">
    <slot/>
  </button>
</template>
```

App.vue

```
<template>
  <BaseButton>
    Submit
    <PulseLoader v-if="isLoading" color="#fff" size="6px"/>
    <AppIcon v-else icon=“arrow-right”/>
  </BaseButton>
</template>
```

PROBLEM

What if you want to define multiple slots?

Default Slot

NavigationLink.vue

```
<a
  :href="url"
  class="nav-link"
>
  <slot></slot>
</a>
```

```
<navigation-link url="/profile">
  <span class="fa fa-user”/>
  Your Profile
</navigation-link>
```

Named Slots

BaseLayout.vue

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

```
<base-layout>
  <template v-slot:header>
    <h1>Here might be a page title</h1>
  </template>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <template v-slot:footer>
    Here's some contact info
  </template>
</base-layout>
```

Named Slots

BaseLayout.vue

```vue
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

```vue
<base-layout>
  <template #header>
    <h1>Here might be a page title</h1>
  </template>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <template #footer>
    Here's some contact info
  </template>
</base-layout>
```

Dynamic Slot Names

```
<base-layout>
  <template v-slot:[dynamicSlotName]>
    ...
  </template>
</base-layout>
```

PROBLEM

How do you access component data from the slot?

Scoped Slots (i.e., Slot Props)

```
// todo-list.vue
<ul>
  <li v-for="todo in todos" :key="todo.id">
    <slot name="header" :header="todo.header" >
      <!-- Fallback content -->
      {{ todo.text }}
    </slot>
    <slot :todo="todo" :newData="newData" >
      <!-- Fallback content -->
      {{ todo.text }}
    </slot>
  </li>
</ul>
```

```
<todo-list :todos="todos">
  <template v-slot="slotProps">
    {{ slotProps.todo }}
  </template>
  <template v-slot:header="slotProps">
     {{ slotProps.header }}
  </template>
</todo-list>
```

Destructuring slot-scope

```
<todo-list :todos="todos">
  <template v-slot="slotProps">
    <AppIcon
      v-if="slotProps.todo.completed"
      icon="checked"
    />
    {{ slotProps.todo.text }}
  </template>
</todo-list>
```

```
<todo-list :todos="todos">
  <template v-slot="{ todo }">
    <AppIcon
      v-if="todo.completed"
      icon="checked"
    />
    {{ todo.text }}
  </template>
</todo-list>
```

Use slots for:

- Content distribution (like layouts)
- Creating larger components by combining smaller components
- Default content in Multi-page Apps
- Providing a wrapper for other components
- Replace default component fragments

Use scoped slots for:

- Applying custom formatting/template to fragments of a component
- Creating wrapper components
- Exposing its own data and methods to child components

Pros

- Great for creating reusable and *composable* components
- Receiving properties from slot-scope is explicit

Cons

- Properties received through slot-scope can't be easily used in component script
- However, you can pass those to methods inside the template as arguments

Questions?

Slots > Props

Composition > Configuration

With composition, you're less restricted by what you were building at first.

With configuration, you have to document everything and new requirements means new configuration.

PROBLEM

How to dynamically switch components based on data?

TECHNIQUE

<Component :is="name">

```
<template>
  <div>
    <Component :is=“clockType" v-bind="clockProps"/>
  </div>
</template>

<script>
export default {
  components: { DigitalClock },
  computed: {
    clockType () {
      if (this.selectedClock === 'analog') {
         this.clockProps = {
            …analogProps
          }
        return () => import(`./components/${this.compName}`)
      } else {
        return 'DigitalClock'
      }
    }
  }
  // ...
}
```

<Component :is>

Becomes the component specified by the **:is** prop.

```
<template>
  <div>
    <Component :is="clockType" v-bind="clockProps"/>
  </div>
</template>

<script>
export default {
  components: { DigitalClock },
  computed: {
    clockType () {
      if (this.selectedClock === 'analog') {
         this.clockProps = {
            …analogProps
          }
        return () => import(`./components/${this.compName}`)
      } else {
        return 'DigitalClock'
      }
    }
  }
  // ...
}
```

Pros

- Extremely powerful and flexible
- Easy to use
- Can accept props
- Can accept asynchronous components
- Can change into different components
- You can make a router-view out of it

Cons

- Got to handle props carefully

LUNCH BREAK

Be back at 12:50PM!