

Ridiculously Reusable Components

Please fill out this quick survey!

<https://bencodezen.typeform.com/to/cWW5XI>

Workshop Repos:

<https://github.com/ridiculously-reusable-components>

Workshop Resources

Kanban Board app

A little about us...



Damian Dulisz
Vue.js Core Team

GitHub: [@shentao](#)
Twitter: [@damiandulisz](#)



Ben Hong
Vue.js Community Partner

GitHub: [@bencodezen](#)

Twitter: [@bencodezen](#)

A little bit about you...

Get to know your neighbors

1. Name
2. Skills
3. Fun Fact

Before we get started...

PARTICIPATION TIPS

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

There are no wrong questions here.

PARTICIPATION TIPS

All slides and examples are **public**.

No need to hurry and copy notes before we switch slides.

PARTICIPATION TIPS

Please **no recording.**

Out of respect for you and your participants' privacy

PARTICIPATION TIPS

These are **guidelines**. Not rules.

Feel free to question and/or disagree.

Your opinion and experience matter too.

Choose what works best for you and your team.

Questions?

What will we cover?

What will we cover?

1. Techniques for building and managing components
2. Pro tips from our experience
3. Best practices for designing components
4. Useful design patterns

There's going to be
a lot of information...

There's going to be
a lot of information...

but we encourage you to stay
engaged and take notes on what
parts interest you.



Components Review

```
<template>
  <ul>
    <li class="nav-item">
      <a href="/Home">Home</a>
    </li>
    <li class="nav-item">
      <a href="/About">About</a>
    </li>
    <li class="nav-item">
      <a href="/Contact">Contact</a>
    </li>
  </ul>
</template>
```

Components Review

```
<template>
  <ul>
    <li class="nav-item">
      <a href="/Home">Home</a>
    </li>
    <li class="nav-item">
      <a href="/About">About</a>
    </li>
    <li class="nav-item">
      <a href="/Contact">Contact</a>
    </li>
  </ul>
</template>
```

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

Components Review

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

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

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

Why components?

- Build things faster
- Stop writing the same code
- Maintenance is much easier
- Less bugs when making changes
- Codebase is much easier to scale
- Testing is more manageable

Questions?

Let's do a

Let's do a
Coding Experiment

Task 1

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

Submit

Task 2

*Allow the button to display an icon of choice
on the right side of the text*

Submit →

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

This is the code responsible for displaying an arrow.

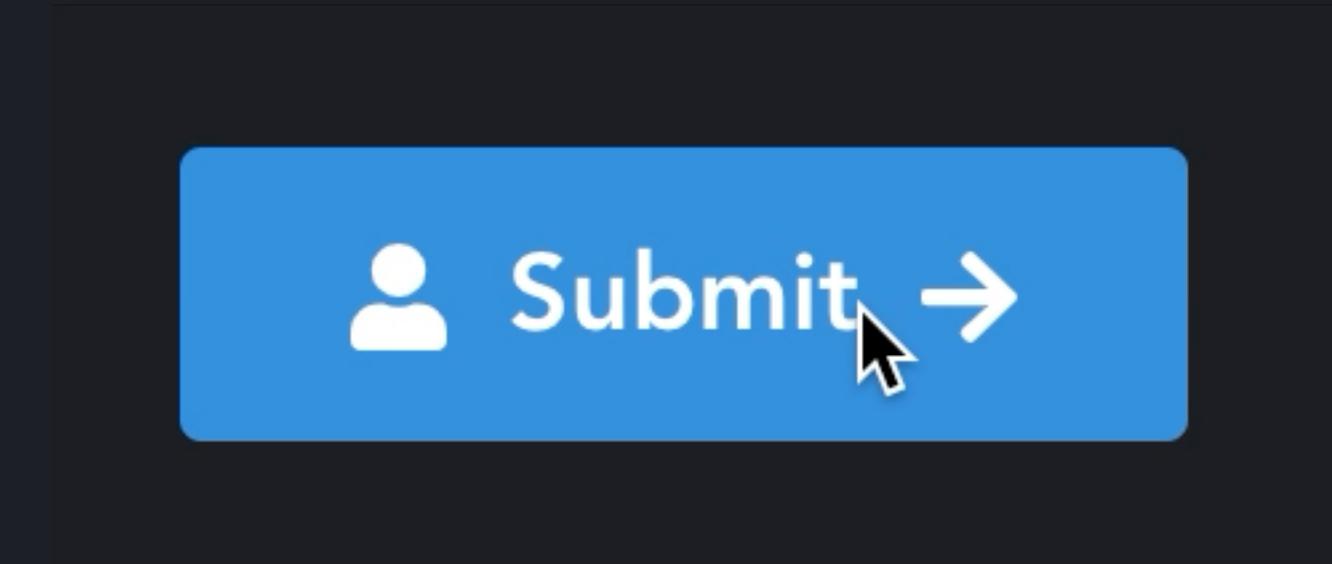
Task 3

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

 Submit →

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

Submit →

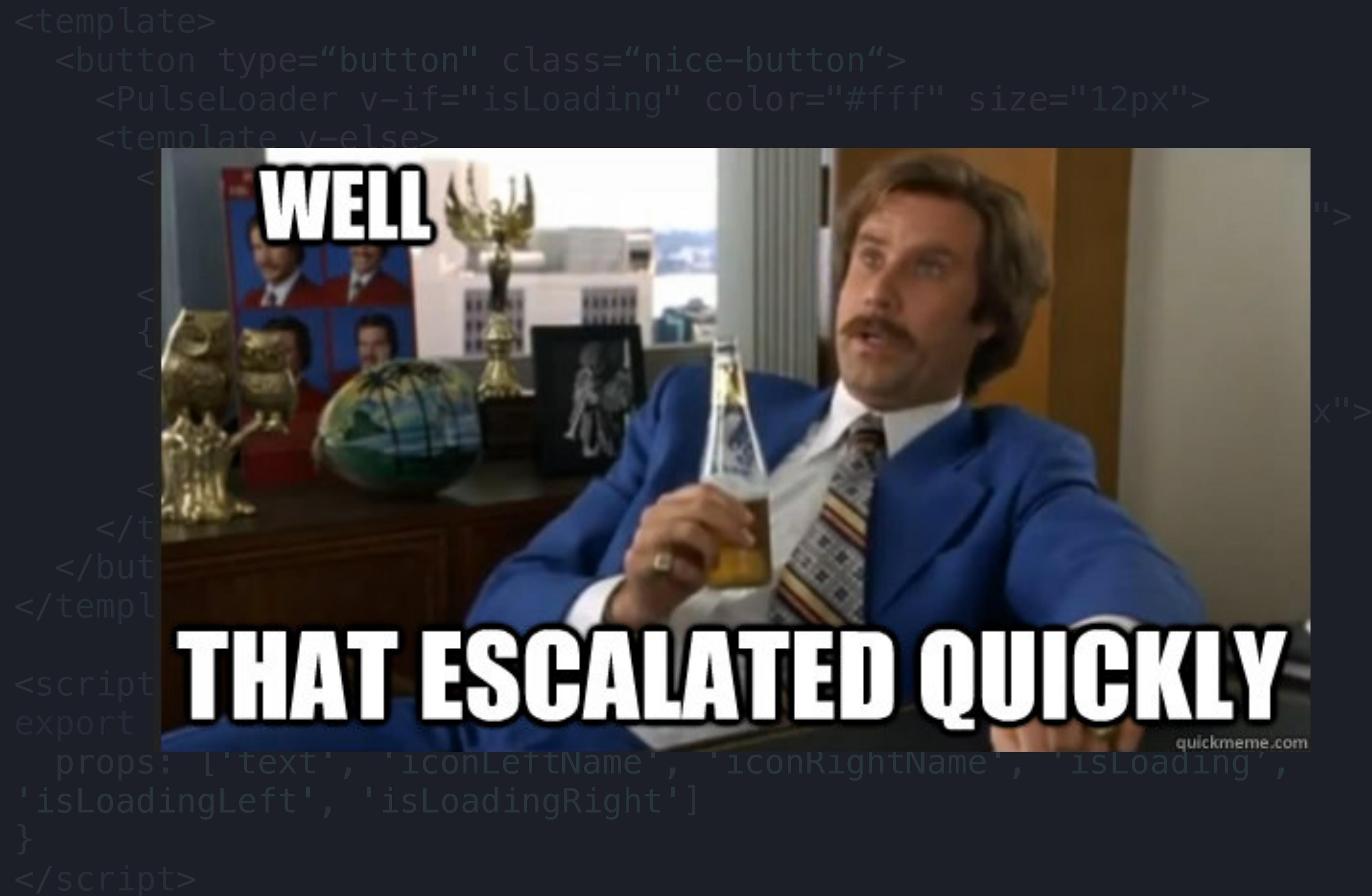
Possible solution

```
<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>
```



```
<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?

Of course!

Recommended solution

Recommended solution

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

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

Usage:

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

Live Demo

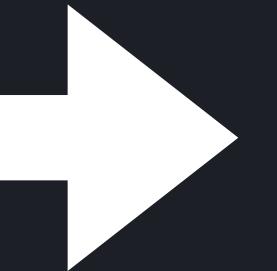
AppTooltip & AppDropdown

Slots

<https://vuejs.org/v2/guide/components-slots.html#ad>

Default Slot

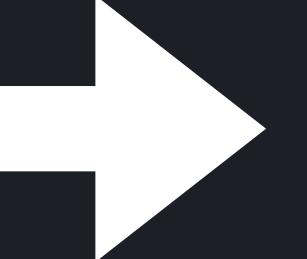
```
// navigation-link.vue
<a
  v-bind:href="url"
  class="nav-link"
>
  <slot></slot>
</a>
```



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

Named Slots

```
// base-layout.vue
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```



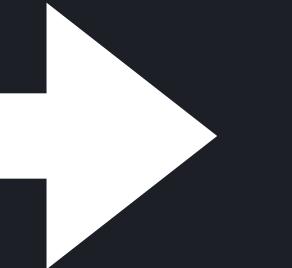
```
<base-layout>
  <template slot="header">
    <h1>Here might be a page title</h1>
  </template>

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

  <p slot="footer">
    Here's some contact info
  </p>
</base-layout>
```

Scoped slots

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



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

Scoped slots

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

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



Scoped slots

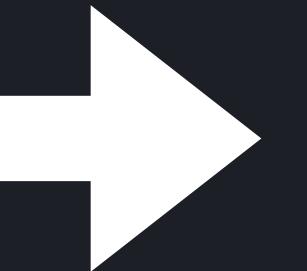
```
// todo-list.vue
<ul>
  <li
    v-for="todo in todos"
    :key="todo.id"
  >
    <slot :todo="todo">
      <!-- Fallback content -->
      {{ todo.text }}
    </slot>
  </li>
</ul>
```

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

The diagram illustrates the data flow between the parent component's slot and the child component's template. A red arrow originates from the slot reference in the parent component's code and points to the slot-scope attribute in the child component's template. Another red arrow originates from the scope object in the child component's template and points to the todo prop used in the AppIcon component.

Destructuring slot-scope

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



```
<todo-list :todos="todos">  
  <template slot-scope="{ 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 changes in `Vue` v2.6

What's new in v2.6

Unified `v-slot` directive

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

  <p>Main content.</p>
  <p>And another one.</p>

  <template slot="footer">
    <p>Here's some contact info</p>
  </template>
</base-layout>
```

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

  <p>Main content.</p>
  <p>And another one.</p>

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

What's new in v2.6

Unified `v-slot` directive

```
<base-layout>
  <template slot="header">—————<base-layout>
    <h1>Here might be a page title</h1>
  </template>

  <p>Main content.</p>
  <p>And another one.</p>

  <template slot="footer">—————<base-layout>
    <p>Here's some contact info</p>
  </template>
</base-layout>
```

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

  <p>Main content.</p>
  <p>And another one.</p>

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

What's new in v2.6

Unified `v-slot` directive

```
<todo-list :todos="todos">
  <template slot="todo" slot-scope="{ todo }">
    {{ todo.text }}
  </template>
</todo-list>
```

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

What's new in v2.6

Unified `v-slot` directive

```
<todo-list :todos="todos">
  <template slot="todo" slot-scope="{ todo }">
    {{ todo.text }}
  </template>
</todo-list>
```



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

What's new in v2.6

v-slot directive shorthand

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

```
<todo-list :todos="todos">
  <template #todo="{ todo }">
    {{ todo.text }}
  </template>
</todo-list>
```

What's new in v2.6

v-slot directive shorthand

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



```
<todo-list :todos="todos">
  <template #todo="{ todo }">
    {{ todo.text }}
  </template>
</todo-list>
```

What's new in v2.6

Dynamic Slot Names

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

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

Advanced Practice

<http://localhost:8080/#/tasks/1>

Tasks:

1. Compose **AppSelect.vue** that uses **AppDropdown.vue**. Let it accept an array of options and the selected value.
2. Expose a default slot for showing the current value
3. Expose an option slot for modifying how the option list would look like.

Questions?

Break (15 min)

PRO TIP

Organizing Component Files

Nested Structure

```
▲ src
  ▲ components
    ▲ Dashboard
      ▶ tests
      ▼ Header.vue
    ▲ Login
      ▶ tests
      ▼ Header.vue
      ▼ Login.vue
      ▶ tests
      ▼ Header.vue
```

Flat Structure

```
▲ 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
```

```
▲ 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
```

Component 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

Component Organization

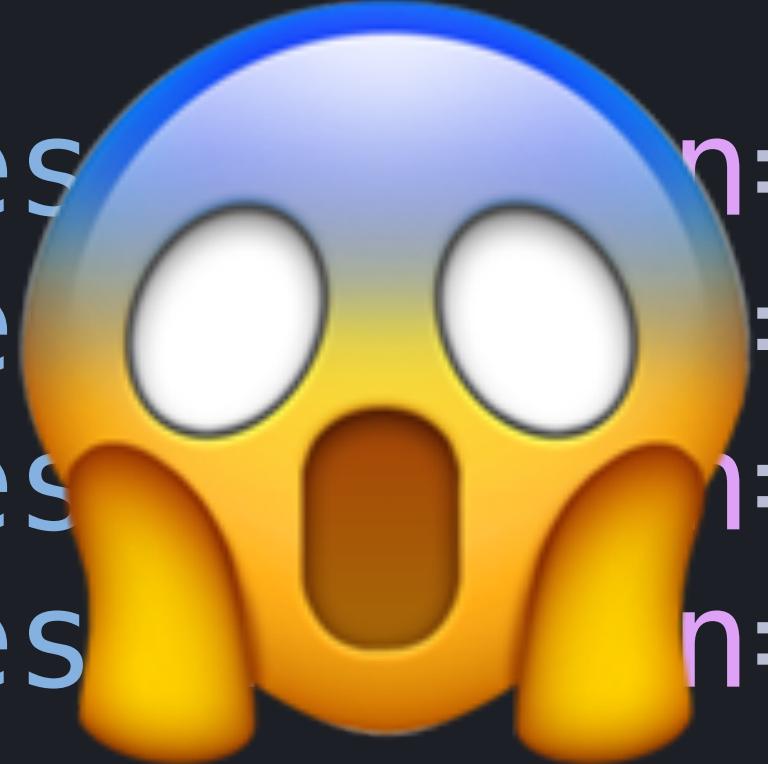
Introduce catalogs once you need them

No need to update imports if components move

DESIGN PATTERN

Wrap Vendor Components

```
<template>
  <p>
    <FontAwe
    <FontAwe
    <FontAwe
    <FontAwe
  </p>
</template>
```



BaseIcon .vue

```
<template>
  <FontAwesomeIcon
    v-if="source === 'font-awesome'"
    :icon="name"
  />
  <span
    v-else
    :class="customIconClass"
  />
</template>
```

```
<template>
  <p>
    <BaseIcon icon="earth" />
    <BaseIcon icon="fire" />
    <BaseIcon icon="water" />
    <BaseIcon icon="water" />
  </p>
</template>
```

BEST PRACTICE

Naming Components



Actual
programming



Debating for
30 minutes on
how to name a
variable

Recommended Naming Conventions

Avoid single word components

~~Header.vue~~

~~Button.vue~~

~~Container.vue~~

Recommended Naming Conventions

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

Recommended **Naming Convention**

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

More conventions:

<https://vuejs.org/v2/style-guide/>

- Single-file component filename casing
- Base component names
- Single-instance component names
- Tightly coupled component names

And more...

PRO TIP

Register base components globally

Instead of every component having:

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

Use this epic script by Chris Fritz:

<https://vuejs.org/v2/guide/components-registration.html#Automatic-Global-Registration-of-Base-Components>

BEST PRACTICE

Designing Component Methods

Use descriptive names

✗ `onInput`

✓ `updateUserName`

Don't assume where it will be called

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

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

✗ `Wrong`

✓ `Correct`

Prefer destructuring over multiple arguments

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

Prefer destructuring over multiple arguments

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

✗ Not recommended

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

✓ Recommended

PRO TIP

v-bind="“{ ... }”

v-on="“{ ... }”

When working with multiple props consider...

```
<VueMultiselect  
  :options="options"  
  :value="value"  
  :key="“0”"  
  label="name"  
/>
```

When working with multiple props consider...

```
<VueMultiselect  
  :options="options"  
  :value="value"  
  :key="0"  
  label="name"  
/>
```

```
<VueMultiselect v-bind="{"  
  options,  
  value,  
  key: 0,  
  label: 'name'  
 }"/>
```

PRO TIP

SFC Code Block Order

```
<template>
<!-- ... -->
</template>
```

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

```
<style>
/* ... */
</style>
```

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

```
<template>
<!-- ... -->
</template>
```

```
<style>
/* ... */
</style>
```

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

```
<template>
<!-- ... -->
</template>
```

```
<style>
/* ... */
</style>
```

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

```
<template>
<!-- ... -->
</template>
```

```
<style>
/* ... */
</style>
```

DESIGN PATTERN

Transparent Components

When passing props, listeners, and attributes...

```
// BaseInput.vue
<template>
  <div>
    <input
      type="text"
      />
  </div>
</template>
```

When passing props, listeners, and attributes...

```
// BaseInput.vue
<template>
  <div>
    <input type="text" />
  </div>
</template>

<BaseInput
  @input="filterData"
  label="Filter results"
  placeholder="Type in here..." />
```



When passing props, listeners, and attributes...

```
// BaseInput.vue
<template>
  <div>
    <input
      type="text"
    />
  </div>
</template>

<BaseInput
  @input="filterData"
  label="Filter results"
  placeholder="Type in here..." />
```

The diagram illustrates the flow of props from the parent component to the child template. It consists of three main parts: a parent component representation, a child template representation, and a connection between them.

- Parent Component:** On the left, there is a code snippet representing a parent component:

```
<BaseInput
  @input="filterData"
  label="Filter results"
  placeholder="Type in here..." />
```
- Child Template:** On the right, there is a code snippet representing the child template:

```
// BaseInput.vue
<template>
  <div>
    <input
      type="text"
    />
  </div>
</template>
```
- Connection:** A red arrow points from the `@input` prop in the parent component to the `input` element in the child template. Another red arrow points from the `label` prop in the parent component to the `div` element in the child template. A third red arrow points from the `placeholder` prop in the parent component to the `input` element in the child template.

When passing props, listeners, and attributes...

```
<template>
  <div>
    <input
      type="text"
      />
  </div>
</template>

<script>
export default {
  inheritAttrs: false,
  // ...
}
</script>
```

Both props and attributes, as well as all listeners will be passed to this element instead.

Prevent Vue from assigning attributes to top-level element

When passing props, listeners, and attributes...

```
<template>
  <div>
    <input
      type="text"
      v-bind="{ ...$attrs, ...$props }"
      v-on="$listeners"
    />
  </div>
</template>

<script>
export default {
  inheritAttrs: false,
  // ...
}
</script>
```

DESIGN PATTERN

Functional Components

Problem

How to avoid repetition when using
component composition through slots?

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

Save the composition as a component.

```
// SubmitButton.vue
<template>
  <AppButton>
    Submit
    <PulseLoader v-if="isLoading" color="#fff" size="6px"/>
    <AppIcon v-else icon="arrow-right"/>
  </AppButton>
</template>
```

It can be a functional component.

```
// SubmitButton.vue
<template functional>
  <AppButton v-on="listeners">
    Submit
    <PulseLoader v-if="props.isLoading" color="#fff" size="6px"/>
    <AppIcon v-else icon="arrow-right"/>
  </AppButton>
</template>
```

Remember to forward props

Vue-CLI Apps support JSX

```
const MyFunctionalComponent = ({ props }) => <div>  
  { props.message }  
</div>
```



This will translate into:

```
const MyFunctionalComponent = {  
  functional: true,  
  render (h, { props }) {  
    return <div>{ props.message }</div>  
  }  
}
```

Questions?

Practice

Workshop Repo Task #2:

1. Create a **ConfirmationModal.vue** that uses **AppModal** to that accepts a question and contains two buttons: "Confirm" and "Cancel"
2. Make the **ConfirmationModal** component emit a 'confirm' event when clicked on 'Confirm' button. And a 'close' event when clicking on 'Cancel'.
3. Display the confirmation question before the button, similar to the example below.
4. Make sure the **ConfirmationModal** looks like the example, this includes the H3 styling and buttons positioning.

Problem

How to dynamically switch
components based on data?

<Component :is="“name”">

<https://vuejs.org/v2/guide/mixins.html>

```
<template>
  <div>
    <Component :is="clockType" :time="time"/>
  </div>
</template>
```

```
<script>
export default {
  components: { AnalogClock, DigitalClock },
  computed: {
    clockType () {
      if (this.selectedClock === 'analog') {
        return 'AnalogClock'
      } else {
        return 'DigitalClock'
      }
    }
  }
  // ...
}
</script>
```

<Component :is>

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

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

Problem

How to share the same functionality
across different components?

Mixins

<https://vuejs.org/v2/guide/mixins.html>

A mixin

```
const myMixin = {  
  data () {  
    return {  
      foo: 'bar'  
    }  
  }  
}
```

```
export default {  
  mixins: [myMixin],  
  // component code  
}
```

Mixin as a function

```
const myMixin = (defaultFoo) => ({  
  data () {  
    return {  
      foo: defaultFoo  
    }  
  }  
})  
  
export default {  
  mixins: [myMixin(10)],  
  // component code  
}
```

Only use mixins when:

You need to share component logic
between multiple components

Unless

You can extract the shared logic to a
component.

You most likely can.

Pros

- Relatively easy to use

Cons

- Possible properties name clashes.
- Can't share template fragments
- Gets harder to track where things are coming from once there are more mixins

Questions?

Instead of Mixins
use...

Composition API

When Vue 3.0 is here*

Unless you need to share the
UI too then use use...

“Composition over inheritance”

“Composition over inheritance”

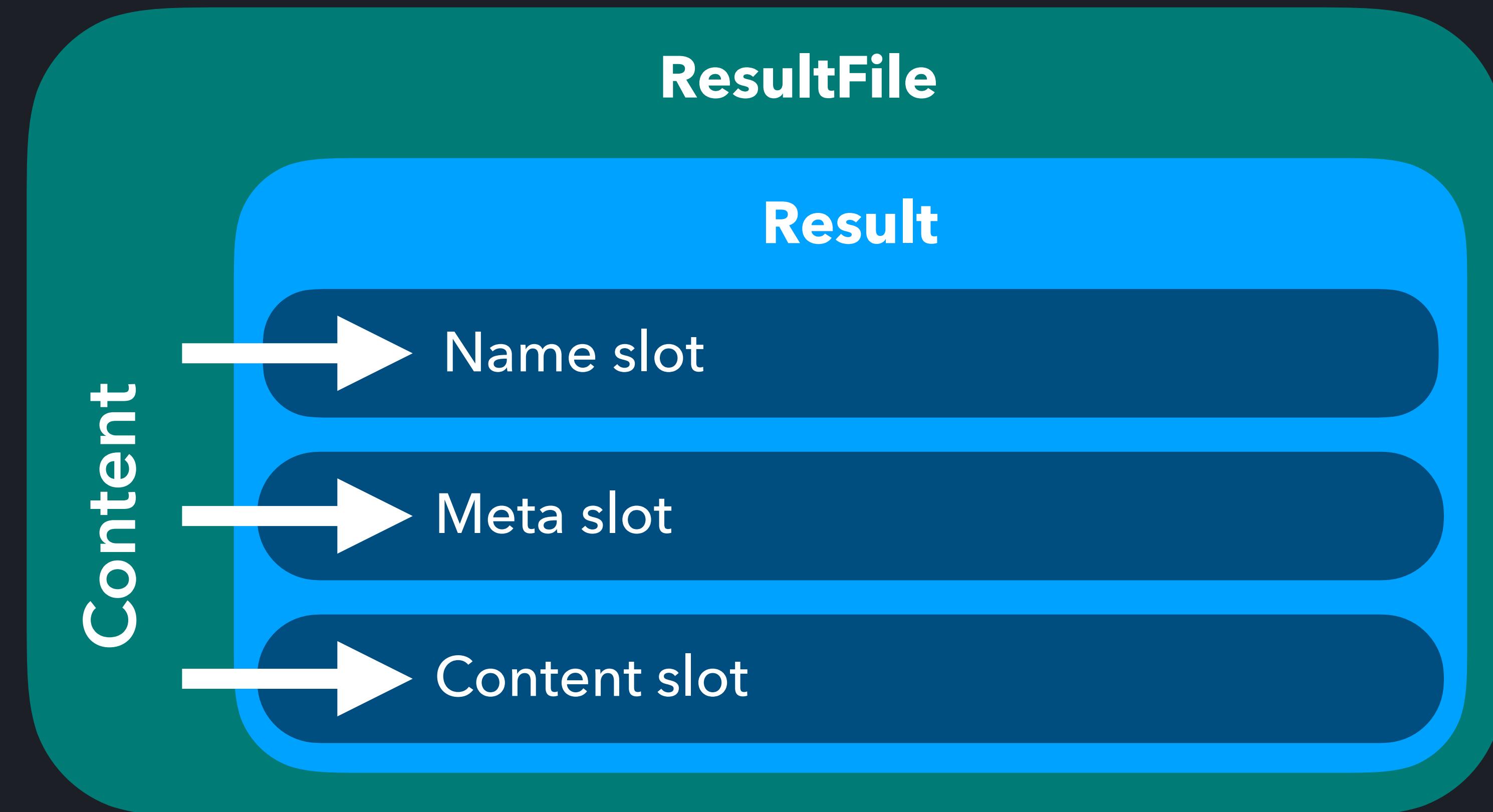
Result

Name slot

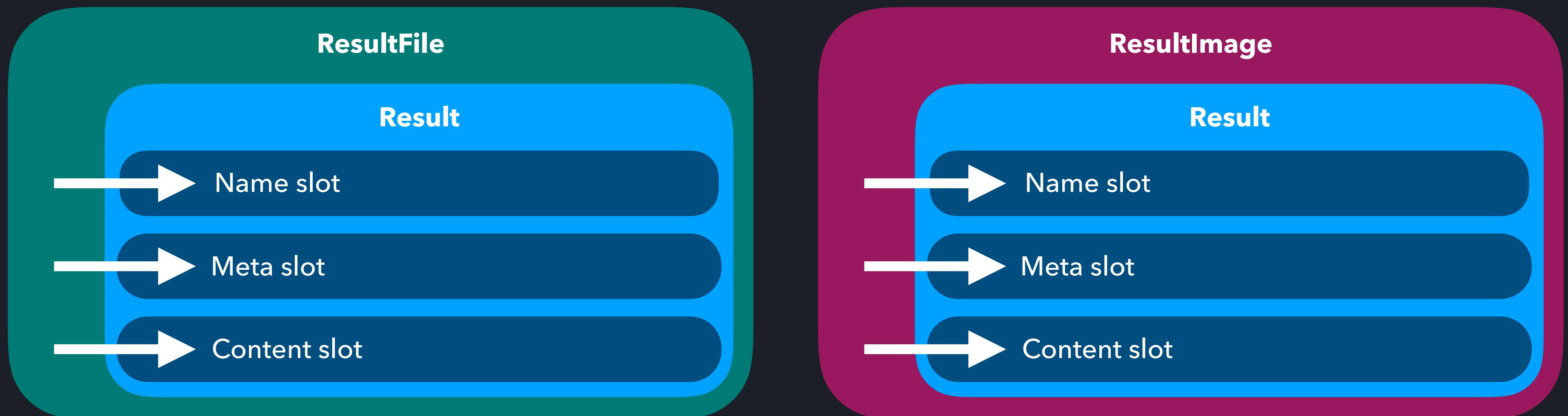
Meta slot

Content slot

“Composition over inheritance”



“Composition over inheritance”



Live Demo

Real life code example

<https://github.com/CodePilotai/codepilot/blob/master/src/renderer/components/search-results-item-issue.vue>

<https://github.com/CodePilotai/codepilot/blob/master/src/renderer/components/search-results-item.vue>

...and other files starting with “search-result-item-”

Questions?

Practice

<http://localhost:8080/#/tasks/3>

Tasks:

1. Create the **NewsEvent.vue** component that will handle the "event" type news from the newsFeed.
2. Similar to **NewsAd** and **NewsPost**, make use of the **BaseNews** component as a method to share code.

Break (15 min)

Problem

What if you only want to expose **Data** and
Methods, but no **User Interface**?

Composition API

When Vue 3.0 is here*

“Provider” components

“Provider” components

```
<ApolloQuery
  :query="require('../graphql/Helloworld.gql')"
  :variables="{ name }"
>
<template slot-scope="{ result: { loading, error, data } }">
  <!-- Loading -->
  <div v-if="loading" class="loading apollo">Loading...</div>
  <!-- Error -->
  <div v-else-if="error" class="error apollo">An error occurred</div>
  <!-- Result -->
  <div v-else-if="data" class="result apollo">{{ data.hello }}</div>
```

“Provider” components

```
<ApolloQuery
  :query="require('../graphql/Helloworld.gql')"
  :variables="{ name }"
>
<template slot-scope="{ result: { loading, error, data } }">
  <!-- Loading -->
  <div v-if="loading" class="loading apollo">Loading...</div>
  <!-- Error -->
  <div v-else-if="error" class="error apollo">An error occurred</div>
  <!-- Result -->
  <div v-else-if="data" class="result apollo">{{ data.hello }}</div>
```

“Provider” components

```
<ApolloQuery
  :query="require('../graphql/Helloworld.gql')"
  :variables="{ name }"
>
<template slot-scope="{ result: { loading, error, data } }">
  <!-- Loading -->
  <div v-if="loading" class="loading apollo">Loading...</div>
  <!-- Error -->
  <div v-else-if="error" class="error apollo">An error occurred</div>
  <!-- Result -->
  <div v-else-if="data" class="result apollo">{{ data.hello }}</div>
```

“Provider” components

```
<ApolloQuery
  :query="require('../graphql/Helloworld.gql')"
  :variables="{ name }"
>
<template slot-scope="{ result: { loading, error, data } }">
  <!-- Loading -->
  <div v-if="loading" class="loading apollo">Loading...</div>
  <!-- Error -->
  <div v-else-if="error" class="error apollo">An error occurred</div>
  <!-- Result -->
  <div v-else-if="data" class="result apollo">{{ data.hello }}</div>
```

Live Coding

MousePosition

Renderless Components

```
export default {
  data () {
    return {
      x: 0,
      y: 0
    }
  },
  render () {
    return this.$scopedSlots.default({ x: this.x, y: this.y })
  },
  mounted () {
    window.addEventListener('mousemove', this.handleMouseMove)
  },
  beforeDestroy () {
    window.removeEventListener('mousemove', this.handleMouseMove)
  },
  methods: {
    handleMouseMove (e) {
      this.x = e.x
      this.y = e.y
    }
  }
}
```

Consuming the WithMousePos Renderless provider component

```
<WithMousePos>
  <template v-slot="{ x, y }">
    {{ x }}, {{ y }}
  </template>
</WithMousePos>
```

Vue 3.0 Alternative

Composition API

```
import { ref, onMounted, onUnmounted } from '@vue/composition-api'

export default function () {
  const x = ref(0)
  const y = ref(0)

  onMounted(() => {
    window.addEventListener('mousemove', handleMouseMove)
  })

  onUnmounted(() => {
    window.removeEventListener('mousemove', handleMouseMove)
  })

  const handleMouseMove = (e) => {
    x.value = e.x
    y.value = e.y
  }

  return { x, y }
}
```

Consuming the useMousePos Composition function

```
setup () {  
  const { x, y } = useMousePos()  
  return { mouseX: x, mouseY: y }  
},
```

<https://vue-composition-api-rfc.netlify.com/>

Questions?

Practice

<http://localhost:8080/#/tasks/4>

Tasks:

1. Create a renderless component called FetchData
2. Make it accept a URL props where you can pass a url to be fetched.
3. Depending on the status: pending, error or resolved (data fetched), make it render different slots to match the use case below.
4. **Advanced:** Create a Composition-API based Solution

Popular convention for classifying components

Container

aka smart components, providers

Presentational

aka dumb components, presenters

Container

- Application logic
- Application state
- Use Vuex
- Usually Router views

Presentational

- Application UI and styles
- UI-related state only
- Receive data from props
- Emit events to containers
- Reusable and *composable*
- Not relying on global state

Container

Examples:

UserProfile, Product,
TheShoppingCart, Login

What is it doing?

Presentational

Examples:

AppBar, AppModal,
TheSidebar, ProductCard

How does it look?

Should I **always** follow this convention?

Should I **always** follow this convention?

NO

Should I **always** follow this convention?

Not when:

- It leads to premature optimisations
- It makes simple things unnecessarily complex
- It requires you to create strongly coupled components (like feature-aware props in otherwise reusable components)
- It forces you to create unnecessary, one-time-use presenter components

Should I **always** follow this convention?

Instead

- Focus on keeping things simple (methods, props, template, Vuex modules, everything)
- Don't be afraid to have UI and styles in your containers
- Split large, complicated containers into several smaller ones

Problem

Pass **data** and **methods** deep into the
component tree

Provide/Inject

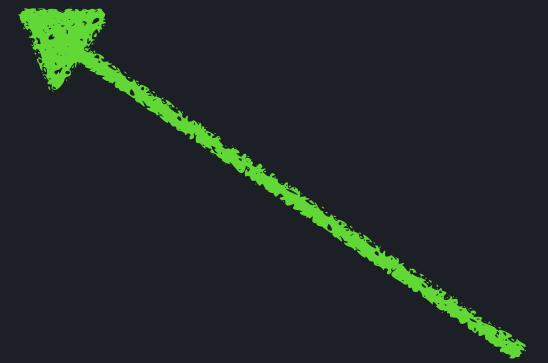
<https://vuejs.org/v2/api/#provide-inject>

Provide/Inject

```
export default {
  provide () {
    return {
      width: this.width, // will stay reactive
      key: 'name', // won't be reactive
      fetchMore: this.fetchMore // methods can be passed
    }
  },
  data() {
    return {
      width: null,
    }
  },
  methods: {
    fetchMore () {
      // ...
    }
  }
}
```

Provide/Inject

```
export default {  
  inject: ['width', 'key', 'fetchMore'],  
  props: {  
    optionKey: {  
      type: String,  
      default () {  
        return this.key  
      }  
    }  
  }  
}
```



Injected values can be used as default props and data values

Pros

- Easy sharing data and methods with descendants
- Helps avoiding unnecessary props
- Components can choose which properties to inject
- Can be used to provide **default props** and **data values**

Cons

- Besides observable objects defined in data, other properties are not reactive
 - Example: computed properties won't update
 - Pretty clumsy usage, due to some properties staying reactive, where other don't
 - Requires complicated setup to make other properties reactive
 - Better suited for plugins and component libraries rather than regular applications

Make provide/inject reactive

<https://github.com/LinusBorg/vue-reactive-provide>

Questions?

TECHNIQUE

EventBus

```
// event-bus.js
import Vue from 'vue';
const EventBus = new Vue();
export default EventBus;
```

```
// component-a.js
import Vue from 'vue';
import EventBus from './event-bus';
Vue.component('component-a', {
  ...
  methods: {
    emitMethod () {
      EventBus.$emit('EVENT_NAME', payLoad);
    }
  }
});
```

```
// component-b.js
import Vue from 'vue';
import EventBus from './event-bus';
Vue.component('component-b', {
  ...
  mounted () {
    EventBus.$on('EVENT_NAME', function (payLoad) {
      ...
    });
  }
});
```

It can be very helpful...

But we **strongly advise against** using
it for global state management.

BEST PRACTICE

Components + Vuex

Tips for using Vuex

State

Tips for **using Vuex**

What data to put into Vuex?

- Data shared between components that might not be in direct parent-child relation
- Data that you want to keep between router views (for example lists of records fetched from the API)
 - Route params are more important though (as a source of truth)
- Any kind of global state
 - Examples: login status, user information, global notifications
- Anything if you feel it will make managing it simpler

Tips for **using** Vuex

What data **NOT** to put into Vuex?

- User Interface variables
 - Examples: `isDropdownOpen`, `isInputFocused`, `isModalVisible`
- Forms data.
- Validation results.
- Single records from the API
 - Think: `currentlyViewedProduct`

Tips for using Vuex

Getters

Tips for **using Vuex**

Do I need to **always** use a **getter** to return a simple fragment of state? **No.**

Feel free to access state directly
`this.$store.state.usersList`

Use computed properties to return computed state

```
activeUsersList () {  
  return this.$store.state.usersList.filter(  
    user => user.isActive  
  )  
}
```

Tips for **using Vuex**

If you need to share derived Vuex state between components, make it a getter.

You should weigh the trade-offs and make decisions that fit the development needs of your app.

Tips for **using Vuex**

Use **mapState** and **mapGetters** helpers

```
computed: {  
  ...mapState({  
    userName: state => state.user.name  
  }),  
  ...mapGetters([  
    'activeUsersList'  
  ]),  
  // local computed properties  
}
```

Tips for using Vuex

Mutations & Actions

Tips for **using Vuex**

Do I need to **always** create an **action** to call a **mutation**?

No.

Feel free to directly commit mutations inside components

```
this.$store.commit('UPDATE_USER', { id, name, isActive })
```

Or use the **mapMutations** helper

```
methods: {
  ...mapMutations({
    updateUser: 'UPDATE_USER'
  })
  // methods
}
```

Tips for **using Vuex**

Think about actions as shared, global methods that connect with a remote API and only affect data stored in Vuex.

If there is no asynchronous part, just use a mutation.

Tips for **using Vuex**

Use modules

<https://vuex.vuejs.org/guide/modules.html>

Questions?

DESIGN PATTERNS

Managing Styles



css

TECHNIQUE

Global Styles

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

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

TECHNIQUE

Inline Styles

```
<template>
  <p :style="`color: ${themeColor}`">
    This should be red
  </p>
</template>
```

TECHNIQUE

Scoped Styles

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

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

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

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

```
<template>
  <p class="red"
      data-f3f3eg9>
    This should be red
  </p>
</template>
```

```
<style scoped>
  .red[data-f3f3eg9] {
    color: red;
  }
  .bold[data-f3f3eg9] {
    font-weight: bold;
  }
</style>
```

TECHNIQUE

CSS Modules

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

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

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

```
<p class="red-xhjd1">
  This should be red
</p>
```

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

```
<style>
  .red-xhdj1 {
    color: red;
  }
</style>
```

Questions?

Problem

Manage **data** deep in the component tree
without mutating it.

Schema Generated Form

Live Coding

SchemaForm

Practice

<http://localhost:8080/#/tasks/5>

Tasks:

1. Create a **FormSelect** component that uses the **AppSelect** component or a regular select element that accepts an array of options.
2. Add a **Country** field to the schema that uses the newly created **FormSelect** and provide a few countries to select from.
3. If the user checks the **isVueFan** checkbox, extend the schema with an additional text input that asks for feedback.

Problem

Extend components template without
having access to slots.

Higher-Order Components

Higher-Order Components

A function that accepts a component as an argument and returns a modified version of that component.

```
function WithMessage (Component) {  
  // code  
  return ModifiedComponent  
}
```

Higher-Order Components

It can also wrap an existing Component within a different component (using slots) or a custom template.

```
function WithMessage (Component) {  
  const ModifiedComponent = ({ props }) => <div>  
    <Component {...props} />  
    With added message { props.message }  
  </div>  
  
  return ModifiedComponent  
}
```

Advanced Practice

<http://localhost:8080/#/tasks/5>

Tasks:

1. Use a Higher-Order Component to wrap the FormComponents to display the `description` prop on the side of the input.
2. Replace all Form Components with a version created by the HoC.

Questions?

BEST PRACTICE

Refactoring

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

- Donald Knuth

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

Components and how to find them?

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

Questions?

Open Practice (90 min)

Open Practice Time

Clone and install

<https://github.com/ridiculously-reusable-components/kanban-board-app>

Or use your own application!

The example app uses **Tailwind CSS**. Read more:

<https://tailwindcss.com/docs/examples/buttons>

The logo consists of the word "Icebox" in a bold, white, sans-serif font. The letter "I" is positioned to the left of a blue snowflake icon, and the letter "X" is positioned to the right of another blue snowflake icon. Both snowflakes have a central grey hexagonal prism and six distinct arms, with blue outlines and highlights.

Icebox

Introduction

1. General tips and best practices
2. Coding experiment

Design Techniques

1. Component Composition
2. Useful design patterns
3. Managing CSS

Exercises inside [workshop resources](#) repo!

Refactoring

1. How to Organize Components
2. Tips and Tricks
3. What to refactor
4. Live refactoring

Useful Component Design Patterns

Resources

1. Slides
2. Repository
3. You

Final Thoughts (15 min)

Things We Didn't Cover

Migration

TypeScript

Hooks

<https://css-tricks.com/what-hooks-mean-for-vue/>

Sarah Drasner

and much more...

Questions?

Thank you!

Damian Dulisz

GitHub: [@shentao](https://github.com/@shentao)

Twitter: [@damiandulisz](https://twitter.com/@damiandulisz)

Ben Hong

GitHub: [@bencodezen](https://github.com/@bencodezen)

Twitter: [@bencodezen](https://twitter.com/@bencodezen)

<https://bencodezen.typeform.com/to/zRaSgV>

<ButtonsLab />

Just the component template and script.

Don't worry about the parent component or styles.

CHALLENGES

**Know how atomic to make them. Handling scope creep.
Knowing what should be a component vs a mix-in.**

- * How to clearly separate app/business logic away from the UI layer.
- * Guidelines around what logic belongs in Vuex vs components.

Reusability and abstraction, when to make two components instead of one big one

Modularity, state management and interacting across components. Not sure the best way to use the event bus.

**How to style instances of components that are nested
inside parent components, if those instances of the
components require style customization**

- Making judgment calls on whether computed properties or call-outs to the Vuex store should take place in that component or the parent components that it's nested in

- Single responsibility principle, also should a component that, say, manipulates objects from an Ajax call - should the objects be passed in as props or should the component be responsible for get/posting these objects.

HOPE TO GAIN

Breaking a system into components

Mostly hoping to get a better sense of a) things to look out for/consider and b) best practices around component design

Mostly hoping to get a better sense of a) things to look out for/consider and b) best practices around component design

Better techniques for making more
performant components

A stronger understanding of how to
compose Vue components together
and make them reusable while still
having sane defaults.

Get a deep understanding of how Vue works, and having a framework to approach handling more complex situations aside from just rendering and reacting to events based on emits, props, computed, methods, etc. I believe that understanding how Vue works more under the hood will help with that, but I'm not sure, so I defer to you.

OTHER QUESTIONS

Currently using Vue-Styleguidist for our component library; is there a better alternative?

I'm particularly interested in writing tests for my Vue components, especially testing containers that compose multiple components together.

I'm currently replacing an AngularJs (v1.5) app w/ Vue.js as part of a piece meal migration, any advice to make it easier for me to port the components would be greatly appreciated.

Would be great to cover typescript

Practice

<https://codesandbox.io/s/2x1zn526v0>

Tasks:

1. Locate repeating component logic
2. Extract it into **InputBaseMixin.js**
3. Add the mixing to all form components

Practice

<https://codesandbox.io/s/40m03r3oqx>

Tasks:

1. Provide the **translate** method in **App.vue**
2. Inject the **translate** method inside **SocialBar.vue**

FORMAT

1. Learn

2. Question

3. Apply

Practice

<https://codesandbox.io/s/xj7mz9x2zp>

Tasks:

1. Introduce a default slot to the **List.vue** component inside the loop.
2. Expose the **option** property to the default slot.
3. Use the **slot-scope** to display the option data in the following format:
FirstName LastName <email>

Problem

Manage **data** deep in the component tree
without mutating it.

Schema Generated Form

Live Coding

SchemaForm

Practice

<http://localhost:8080/#/tasks/5>

Tasks:

1. Create a **FormSelect** component that uses the **AppSelect** component or a regular select element that accepts an array of options.
2. Add a **Country** field to the schema that uses the newly created **FormSelect** and provide a few countries to select from.

Advanced Practice

<http://localhost:8080/#/tasks/5>

Tasks:

1. Create a component that allows to pick a date (without using `input type="date"`). It can be a composition/wrapper around an open-source datepicker or a combination of 3 regular select elements filled with days, months and years.
2. Add the component directly to the **Schema** rather than registering it inside **SchemaForm.vue**
3. If the user checks the `isVueFan` checkbox, extend the schema with an additional text input that asks for feedback.

Questions?

FORMAT

1. Learn

2. Question

3. Apply

Explanations
Examples

Clarifications
What-ifs

Exercises
Experiment

Practice Transparent Components

```
<template>
  <WithErrorMessage>
    <input
      type="text"
      v-bind="{ ...$attrs, ...$props }"
      v-on="$listeners"
    />
  </WithErrorMessage>
</template>
```

```
<script>
export default {
  inheritAttrs: false,
  // ...
}
</script>
```

Transparent components

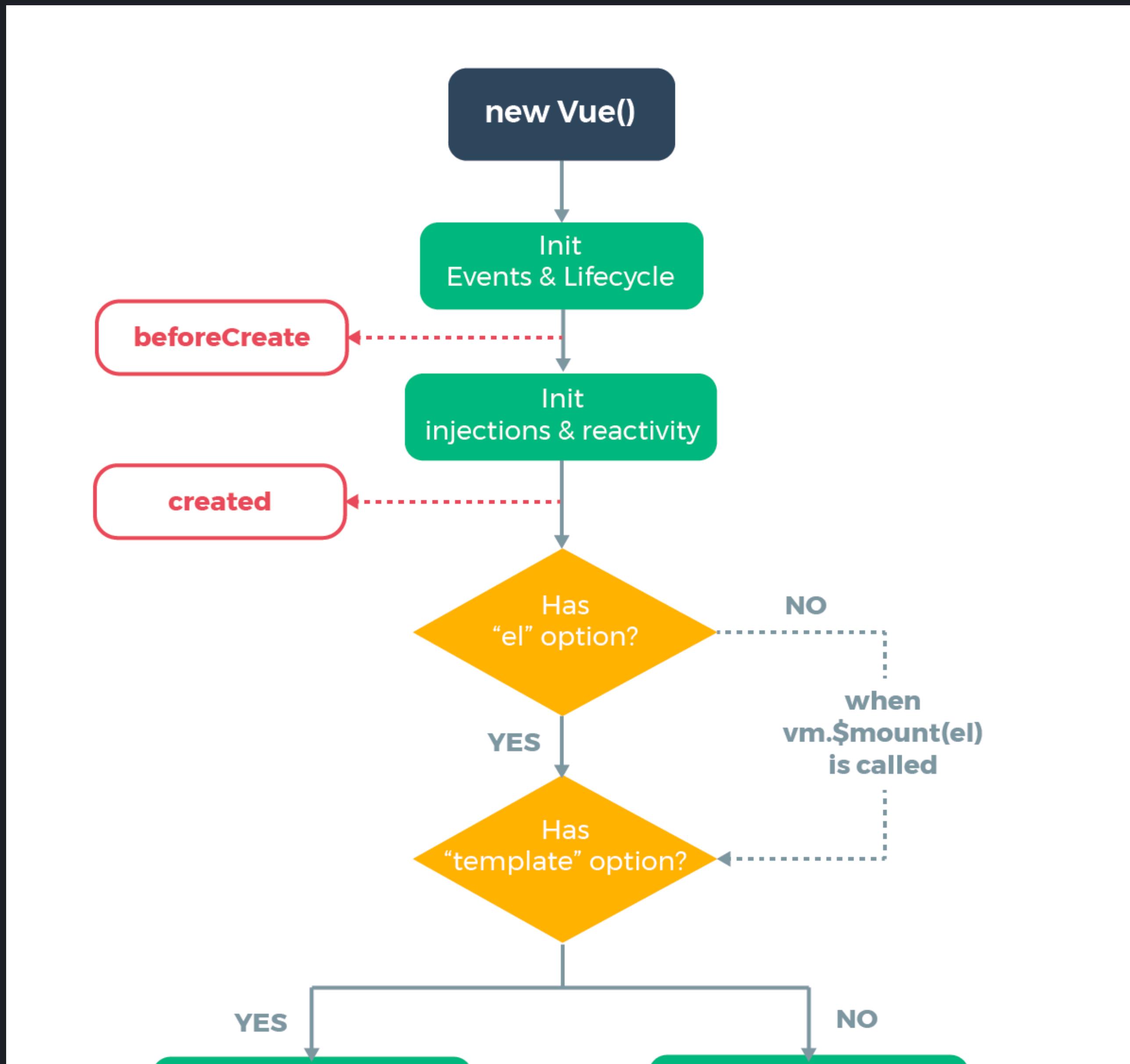
Both props and attributes, as well as all listeners will be passed to this element instead.

Prevent Vue from assigning attributes to top-level element

“PRO” TIP

@hook

<https://vuejs.org/v2/guide/instance.html#Lifecycle-Diagram>



```
<ThirdPartyLibrary  
    @hook:beforeCreate="fetchData"  
    @hook:beforeDestroy="sendNotification"  
/>
```