Ridiculously Reusable Components

Damian Dulisz

GitHub: @shentao

Twitter: @damiandulisz

Dariusz “Gusto” Wędrychowski

GitHub: @gustojs

Twitter: @gustojs

Our background

Format

Lecture

Design Techniques

Open Practice

Lecture

1. Components tips and conventions
2. Application tips and conventions
3. Questions

Design Techniques

1. Problem
2. Solution
3. Try it out
4. Discuss
5. Repeat

Open Practice

1. Analyse
2. Refactor
3. 1on1 help

Example workshop application or your project!

Participation tips

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

There are no wrong questions here.

Feel free to disagree. **Share your perspective**.

It's all about experience and understanding when applying a pattern is worth it.

Questions?

So… Components?

Why components?

- Organise and scale the application
- Make development easier and faster
- Make it easy to test and reason about

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

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:
`AppButton, 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 code (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

Questions?

Suggested **Naming Convention**

AppPrefixedName.vue

Reusable, globally registered UI components.

`AppButton, AppModal, AppDropdown, AppInput`

ThePrefixedName.vue

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

`TheShoppingCart, TheSidebar, TheNavbar`

Suggested **Naming Convention**

Coupled/related components

TodoList.vue
TodoListItem.vue
TodoListItemName.vue

Easy to spot relation

Stay next to each other in the file tree

Name starts with the highest-level words

More conventions:

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

Automatic global components registration:

https://github.com/chrisvfritz/vue-enterprise-boilerplate/blob/master/src/components/_globals.js

Tips for **designing 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

Tips for **designing methods**

Prefer destructuring over multiple arguments

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

❌ Wrong

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

✅ Correct

Tips for **passing props and listeners**

When working with multiple props consider

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

<!-- is the same -->
<VueMultiselect
  :options="options"
  :value="value"
  :key=“”
  label="name"
/>
```

Tips for **passing props and listeners**

```
<template>
  <WithErrorMessage>
    <input
      type="text"
      v-bind="$attrs"
      v-on="$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

Questions?

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 a computed property 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** need to 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 methods that connect with a remote API (or browser API) and manage data stored in Vuex.

Tips for **using Vuex**

Use modules

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

Questions?

Let’s do a

Let’s do a

Coding Experiment

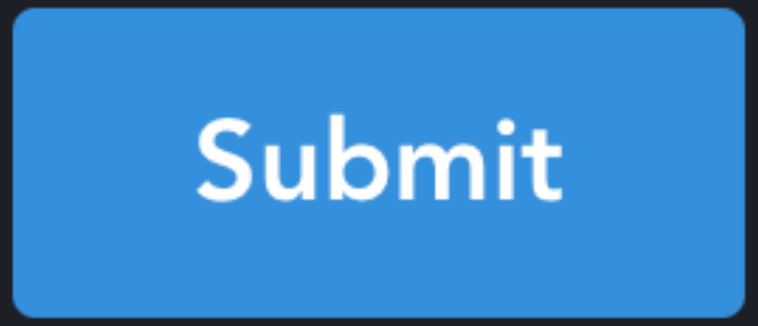
<AppButton>

https://codesandbox.io/s/z3x3zoz413

Just the component template and script.
Don’t worry about the parent component or styles.

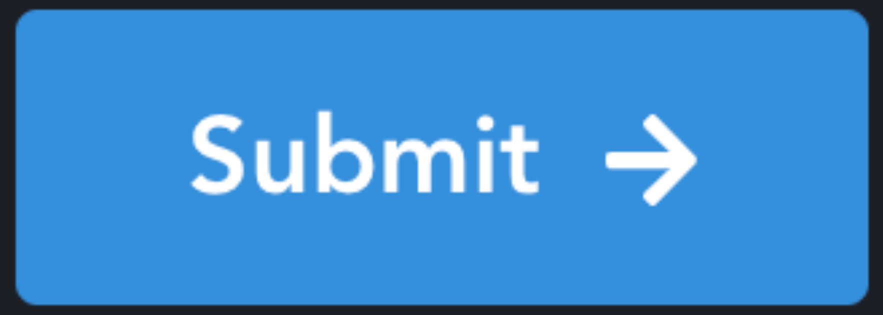
Task 1

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

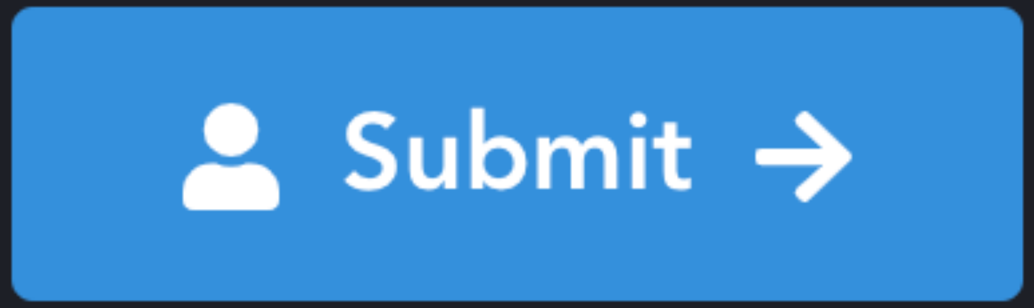


```
<AppIcon icon="arrow-right" class="ml-3"/>
```

This is the code responsible for displaying an arrow.

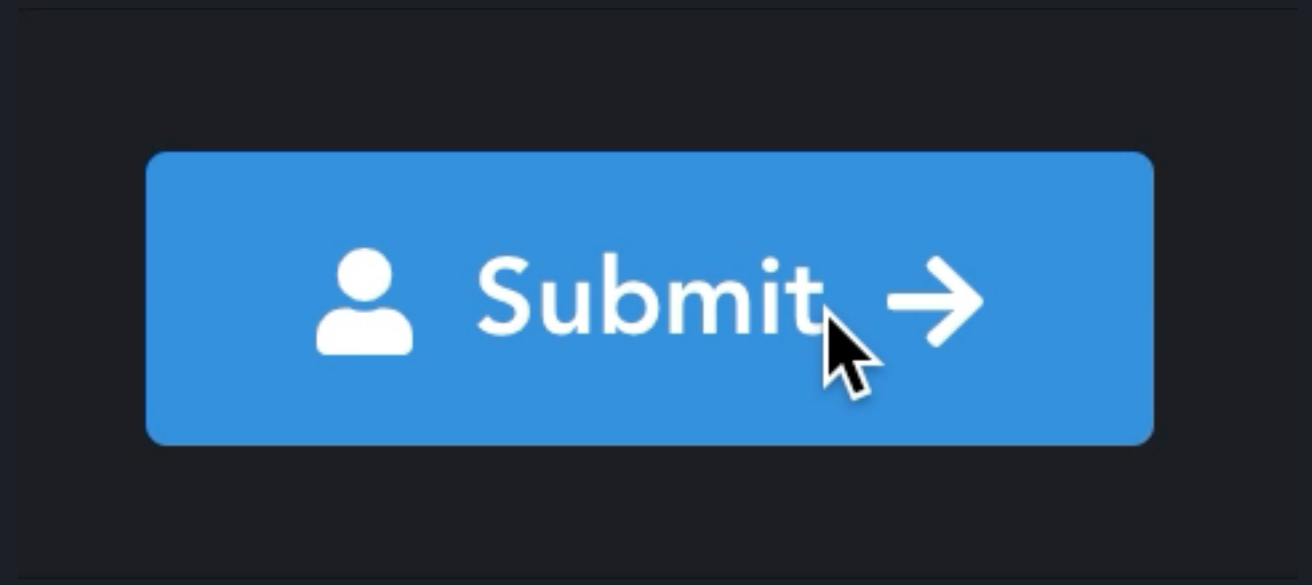
Task 3

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

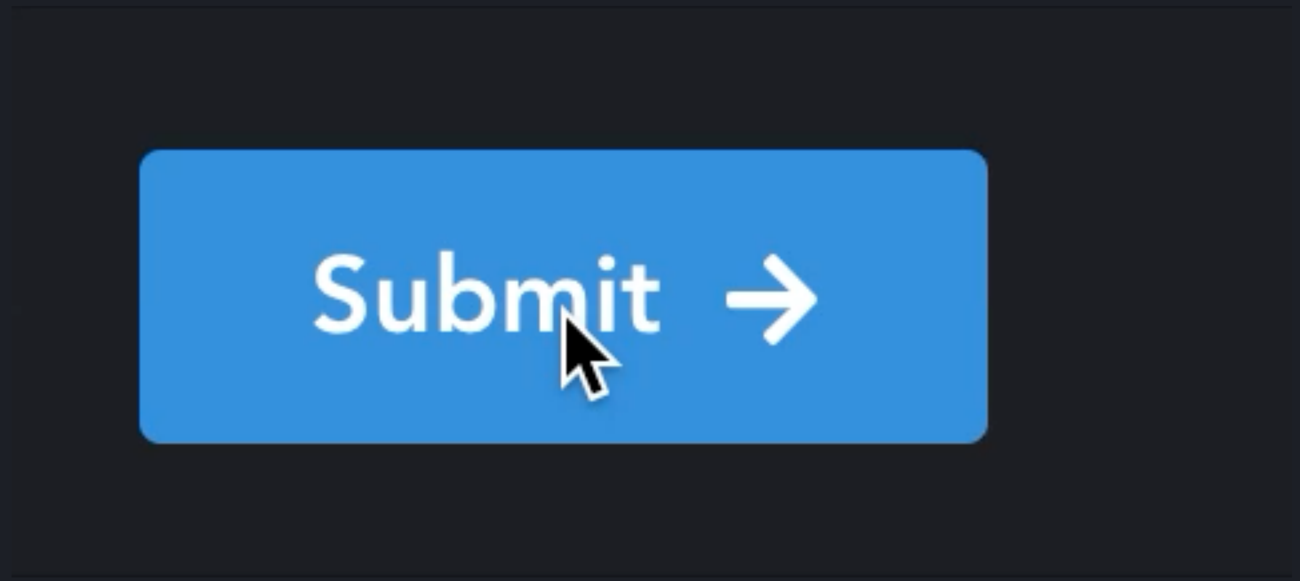


```
<PulseLoader color="#fff" size="12px"/>
```

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“>
    <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>
```

WELL
THAT ESCALATED QUICKLY
quickmeme.com

My recommended solution

My 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>
```

Composition > Configuration

Slots > Props

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>
```

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

Example

```
// AppModal.vue
<template>
  <div class="modal">
    <AppIcon
      @click="$emit('close')"
      icon=“times"
      class="pin-r pin-t m-4 text-lg"
    />
    <slot/>
  </div>
</template>
```

Example

```
// TaskPreview.vue
<template>
  <AppModal @close=“closeTaskView”>
    <h1>{{ task.title }}
    <p>{{ task.description }}</p>

    <AppButton @click="closeTaskView">Close</AppButton>
  </AppModal>
</template>
```

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

Example

```
// AppModal.vue
<template>
  <div class="modal">
    <AppIcon
      @click="$emit('close')"
      icon="times"
      class="pin-r pin-t m-4 text-lg"
    />
    <slot :expand-to-full-screen="expandToFullScreen"/>
  </div>
</template>
```

Example

```
// TaskPreview.vue
<template>
  <AppModal @close="closeTaskView">
    <template slot-scope="{ expandToFullScreen }">
      <h1>{{ task.title }}
      <p>{{ task.description }}</p>

      <AppButton @click="closeTaskView">Close</AppButton>
      <AppButton @click="expandToFullScreen">Full screen</AppButton>
    </template>
  </AppModal>
</template>
```

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

Questions?

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>

What if you only want to expose data and methods, no UI at all?

```
// SelectProvider.vue
<template>
  <div>
    <slot v-bind="{
      value,
      options,
      select,
      deselect,
      open,
      close,
      isOpen
    }"/>
  </div>
</template>
```

Renderless Components

```
// SelectProvider.vue
export default {
  props: ['value', 'options'],
  data () {
    isOpen: false
  },
  render () {
    return this.$scopedSlots.default({
      value: this.value,
      options: this.options,
      select: this.select,
      deselect: this.deselect,
      isOpen: this.isOpen,
      // and more
    })
  },
  methods: {
    // methods
  }
}
```

```
// SelectProvider.vue
export default {
  props: ['value', 'options'],
  data () {
    isOpen: false
  },
  render () {
    // expose everything
    return this.$scopedSlots.default(this)
  },
  methods: {
    // methods
  }
}
```

```
// SelectDropdown.vue
<SelectProvider v-bind="$attrs" v-on="$listeners">
  <template slot-scope="{
    value,
    options,
    select,
    deselect,
    isOpen,
    open,
    close
  }">
    <AppButton @click="open">
      {{ value || 'Pick one' }}
    </AppButton>
    <AppList v-if="isOpen" :options="options" @select="select"/>
  </template>
</SelectProvider>
```

Vue-Multiselect v3.0

Vue-Multiselect v3.0

MultiselectCore.js

Renderless component managing the core functionality. Exposes state, methods and computed properties through the default scoped slot.
No UI.

Vue-Multiselect v3.0

Multiselect.vue

The default composition that implements all features from Vue-Multiselect v2.x.
Proxies all props and event listeners to MultiselectCore.

MultiselectCore.js

Renderless component managing the core functionality. Exposes state, methods and computed properties through the default scoped slot.
No UI.

Works exactly like Vue-Multiselect v2.x

Questions?

Problem

How to switch components based on data?

<Component :is=“name”>

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

<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

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

Questions?

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
}
```

But, aren’t mixins considered harmful in React?

They are.

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

- Create implicit dependencies, where your component is no longer self-contained.
- Possible name clashes.
- Concern separation is a lie.
- Gets harder to track where things are coming from once there are more mixins

Questions?

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

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?

Practice

https://codesandbox.io/s/1zqzn1yl6j

Tasks:

1. Remove **translate** props everywhere
2. Provide the **translate** method in **App.vue**
3. Inject the **translate** method inside **SocialBar.vue**

Done

Open Practice Time

Open Practice Time

Clone and install

https://github.com/shentao/reusable-components-workshop

While installing, read about **Tailwind CSS**:

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