screenshot-adamwathan.me-2020.11.16-12_11_11
https://adamwathan.me/renderless-components-in-vuejs/
16.11.2020

Adam Wathan
Software developer, author, and host of Full Stack Radio.
March 22, 2018

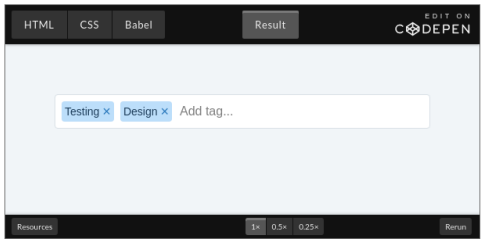
Renderless Components in Vue.js

Have you ever pulled in a third-party UI component only to discover that because of one small tweak you need to make, you have to throw out the whole package?

Custom controls like dropdowns, date pickers, or autocomplete fields can be very complex to build with a lot of unexpected edge cases to deal with.

There are a lot of libraries out there that do a great job handling this complexity, but they often come with a deal-breaking downside: **it's hard or impossible to customize how they look.**

Take this tags input control for example:



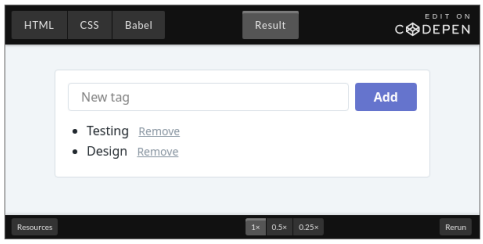
This component wraps up a few interesting behaviors:

- It doesn't let you add duplicates
- It doesn't let you add empty tags
- It trims whitespace from tags
- Tags are added when the user presses enter
- Tags are removed when the user clicks the `×` icon

If you needed a component like this in your project, pulling this in as a package and offloading that logic would certainly save you some time and effort.

But what if you needed it to look a little different?

This component has all of the same behavior as the previous component, but with a significantly different layout:



You could try and support both of these layouts with a single component through a combination of whacky CSS and component configuration options, but *(thankfully)* there's a better way.

Scoped Slots

In Vue.js, slots are placeholder elements in a component that are replaced by content passed in by the parent/consumer:

```
<!-- Card.vue -->
<template>
  <div class="card">
    <div class="card-header">
      <slot name="header"></slot>
    </div>
    <div class="card-body">
      <slot name="body"></slot>
    </div>
  </div>
</template>

<!-- Parent/Consumer -->
<card>
  <h1 slot="header">Special Features</h1>
  <div slot="body">
    <h5>Fish and Chips</h5>
    <p>Super delicious tbh.</p>
  </div>
</card>

<!-- Renders: -->
<div class="card">
  <div class="card-header">
    <h1>Special Features</h1>
  </div>
  <div class="card-body">
    <div>
      <h5>Fish and Chips</h5>
      <p>Super delicious tbh.</p>
    </div>
  </div>
</div>
```

Scoped slots are just like regular slots **but with the ability to pass parameters from the child component up to the parent/consumer**.

Regular slots are like passing HTML to a component; scoped slots are like passing a *callback* that accepts data and *returns* HTML.

Parameters are passed up to the parent by adding props to the slot element in the child component, and the parent accesses these parameters by destructuring them out of the special `slot-scope` attribute.

Here's an example of a `LinksList` component that exposes a scoped slot for each list item, and passes the data for each item back to the parent through a `:link` prop:

```
  <!-- LinksList.vue -->
  <template>
    <!-- ... -->
      <li v-for="link in links">
        <slot name="link"
+         :link="link"
        ></slot>
      </li>
    <!-- ... -->
  </template>

  <!-- Parent/Consumer -->
  <links-list>
    <a slot="link"
+     slot-scope="{ link }"
      :href="link.href"
    >{{ link.title }}</a>
  </links-list>
```

By adding the `:link` prop to the slot element in the `LinksList` component, the parent can now access it through the `slot-scope` and make use of it in its slot template.

Types of Slot Props

You can pass anything to a slot, but I find it useful to think of every slot prop as belonging to one of three categories.

Data

The simplest type of slot prop is just data: strings, numbers, boolean values, arrays, objects, etc.

In our links example, `link` is an example of a data prop; it's just an object with some properties:

```
<!-- LinksList.vue -->
<template>
  <!-- ... -->
    <li v-for="link in links">
      <slot name="link"
        :link="link"
      ></slot>
    </li>
  <!-- ... -->
</template>

<script>
export default {
  data() {
    return {
      links: [
        { href: 'http://...', title: 'First Link', bookmarked: true },
        { href: 'http://...', title: 'Second Link', bookmarked: false },
        // ...
      ]
    }
  }
}
</script>
```

The parent can then render that data or use it to make decisions about what to render:

```
<!-- Parent/Consumer -->
<links-list>
  <div slot="link" slot-scope="{ link }">
    <star-icon v-show="link.bookmarked"></star-icon>
    <a :href="link.href">
      {{ link.title }}
    </a>
  </div>
</links-list>
```

Actions

Action props are *functions* provided by the child component that the parent can call to invoke some behavior in the child component.

For example, we could pass a `bookmark` action to the parent that bookmarks a given link:

```
  <!-- LinksList.vue -->
  <template>
    <!-- ... -->
      <li v-for="link in links">
        <slot name="link"
          :link="link"
+         :bookmark="bookmark"
        ></slot>
      </li>
    <!-- ... -->
  </template>

  <script>
  export default {
    data() {
      // ...
    },
+   methods: {
```

```
+     bookmark(link) {
+       link.bookmarked = true
+     }
+   }
  }
  </script>
```

The parent could invoke this action when the user clicks a button next to an un-bookmarked link:

```
  <!-- Parent/Consumer -->
  <links-list>
-   <div slot="link" slot-scope="{ link }">
+   <div slot="link" slot-scope="{ link, bookmark }">
      <star-icon v-show="link.bookmarked"></star-icon>
      <a :href="link.href">{{ link.title }}</a>
+     <button v-show="!link.bookmarked" @click="bookmark(link)">Bookmark</button>
    </div>
  </links-list>
```

Bindings

Bindings are collections of attributes or event handlers that should be bound to a specific element using `v-bind` or `v-on`.

These are useful when you want to encapsulate implementation details about how interacting with a provided element should work.

For example, instead of making the consumer handle the `v-show` and `@click` behaviors for the bookmark button themselves, we could provide a `bookmarkButtonAttrs` binding and a `bookmarkButtonEvents` binding that move those details into the component itself:

```
  <!-- LinksList.vue -->
  <template>
    <!-- ... -->
    <li v-for="link in links">
      <slot name="link"
        :link="link"
        :bookmark="bookmark"
+       :bookmarkButtonAttrs="{
+         style: [ link.bookmarked ? { display: none } : {} ]
+       }"
+       :bookmarkButtonEvents="{
+         click: () => bookmark(link)
+       }"
      ></slot>
    </li>
    <!-- ... -->
  </template>
```

Now if the consumer prefers, they can apply these bindings to the bookmark button blindly without having to know what they actually do:

```
  <!-- Parent/Consumer -->
  <links-list>
-   <div slot="link" slot-scope="{ link, bookmark }">
+   <div slot="link" slot-scope="{ link, bookmarkButtonAttrs, bookmarkButtonEvents
      <star-icon v-show="link.bookmarked"></star-icon>
      <a :href="link.href">{{ link.title }}</a>
-     <button v-show="!link.bookmarked" @click="bookmark(link)">Bookmark</button>
+     <button
+       v-bind="bookmarkButtonAttrs"
+       v-on="bookmarkButtonEvents"
+     >Bookmark</button>
    </div>
  </links-list>
```

Renderless Components

A *renderless component* is a component that **doesn't render any of its own HTML**.

Instead it only manages state and behavior, exposing **a single scoped slot** that gives the parent/consumer complete control over what should actually be rendered.

A renderless component renders exactly what you pass into it, without any extra elements:

```
<!-- Parent/Consumer -->
<renderless-component-example>
  <h1 slot-scope="{}">
    Hello world!
  </h1>
</renderless-component-example>

<!-- Renders: -->
<h1>Hello world!</h1>
```

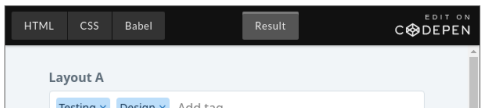
So why is this useful?

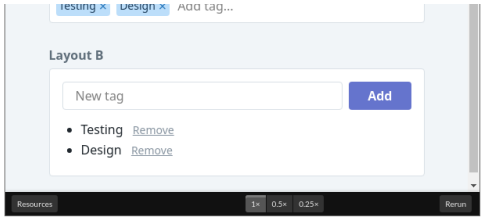
Separating Presentation and Behavior

Since renderless components only deal with state and behavior, they don't impose any decisions about design or layout.

That means that if you can figure out a way to move all of the interesting behavior out of a UI component like our tags input control and into a renderless component, **you can reuse the renderless component to implement any tags input control layout**.

Here's both tag input controls, but this time backed by a single renderless component:





So how does this work?

Renderless Component Structure

A renderless component exposes **a single scoped slot** where the consumer can provide the entire template they want to render.

The basic skeleton of a renderless component looks like this:

```
Vue.component('renderless-component-example', {
  // Props, data, methods, etc.
  render() {
    return this.$scopedSlots.default({
      exampleProp: 'universe',
    })
  },
})
```

It doesn't have a template or render any HTML of its own; instead it uses a render function that invokes the default scoped slot passing through any slot props, then returns the result.

Any parent/consumer of this component can destructure `exampleProp` out of the slot-scope and use it in its template:

```
<!-- Parent/Consumer -->
<renderless-component-example>
  <h1 slot-scope="{ exampleProp }">
    Hello {{ exampleProp }}!
  </h1>
</renderless-component-example>

<!-- Renders: -->
<h1>Hello universe!</h1>
```

A Worked Example

Let's walkthrough building a renderless version of the tags input control from scratch.

We'll start with a blank renderless component that passes no slot props:

```
/* Renderless Tags Input Component */
export default {
  render() {
    return this.$scopedSlots.default({})
  },
}
```

...and a parent component with a static, non-interactive UI that we pass in to the child component's slot:

```
<!-- Parent component -->
<template>
  <renderless-tags-input>
    <div slot-scope="{}" class="tags-input">
      <span class="tags-input-tag">
        <span>Testing</span>
        <span>Design</span>
        <button type="button" class="tags-input-remove">&times;</button>
      </span>
      <input class="tags-input-text" placeholder="Add tag...">
    </div>
  </renderless-tags-input>
</template>

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

Piece by piece, we'll make this component work by adding state and behavior to the renderless component and exposing it to our layout through the slot-scope.

Listing tags

First let's replace the static list of tags with a dynamic list.

The tags input component is a custom form control so like in the original example, the tags should live in the parent and be bound to the component using `v-model`.

We'll start by adding a `value` prop to the component and passing it up as a slot prop named `tags`:

```
  /* Renderless Tags Input Component */
  export default {
+   props: ['value'],
    render() {
-     return this.$scopedSlots.default({})
+     return this.$scopedSlots.default({
+       tags: this.value,
+     })
    },
  }
```

Next, we'll add the `v-model` binding in the parent, fetch the tags out of the slot-scope, and iterate over them with `v-for`:

```
<!-- Parent component -->
<template>
-  <renderless-tags-input>
+  <renderless-tags-input v-model="tags">
-    <div slot-scope="{}" class="tags-input">
+    <div slot-scope="{ tags }" class="tags-input">
      <span class="tags-input-tag">
-       <span>Testing</span>
-       <span>Design</span>
+       <span v-for="tag in tags">{{ tag }}</span>
        <button type="button" class="tags-input-remove">&times;</button>
      </span>
      <input class="tags-input-text" placeholder="Add tag...">
    </div>
  </renderless-tags-input>
</template>

<script>
  export default {
+   data() {
+     return {
+       tags: ['Testing', 'Design']
+     }
+   }
  }
</script>
```

This slot prop is a great example of a simple data prop.

Removing tags

Next let's remove a tag when clicking the `×` button.

We'll add a new `removeTag` method to our component, and pass a reference to that method up to the parent as a slot prop:

```
/* Renderless Tags Input Component */
export default {
  props: ['value'],
+ methods: {
+   removeTag(tag) {
+     this.$emit('input', this.value.filter(t => t !== tag))
+   }
+ },
  render() {
    return this.$scopedSlots.default({
      tags: this.value,
+     removeTag: this.removeTag,
    })
  },
})
```

Then we'll add a `@click` handler to the button in the parent that calls `removeTag` with the current tag:

```
<!-- Parent component -->
<template>
  <renderless-tags-input>
-   <div slot-scope="{ tags }" class="tags-input">
+   <div slot-scope="{ tags, removeTag }" class="tags-input">
      <span class="tags-input-tag">
        <span v-for="tag in tags">{{ tag }}</span>
-       <button type="button" class="tags-input-remove">&times;</button>
+       <button type="button" class="tags-input-remove"
+         @click="removeTag(tag)"
+       >&times;</button>
      </span>
      <input class="tags-input-text" placeholder="Add tag...">
    </div>
  </renderless-tags-input>
</template>

<script>
  export default {
    data() {
      return {
        tags: ['Testing', 'Design']
      }
    }
  }
</script>
```

This slot prop is an example of an action prop.

Adding new tags on enter

Adding new tags is a bit trickier than the last two examples.

To understand why, let's look at how it would be implement in a more traditional component:

```
<template>
  <div class="tags-input">
    <!-- ... -->
    <input class="tags-input-text" placeholder="Add tag..."
      @keydown.enter.prevent="addTag"
      v-model="newTag"
    >
  </div>
</template>

<script>
  export default {
    props: ['value'],
    data() {
      return {
        newTag: '',
      }
    },
    methods: {
      addTag() {
        if (this.newTag.trim().length === 0 || this.value.includes(this.newTag.tri
          return
        }
        this.$emit('input', [...this.value, this.newTag.trim()])
        this.newTag = ''
      },
      // ...
    },
    // ...
  }
```

```
</script>
```

We keep track of the new tag *(before it's been added)* in a `newTag` property, and we bind that property to the input using `v-model`.

Once the user presses enter, we make sure the tag is valid, add it to the list, then clear out the input field.

The question here is **how do we pass a `v-model` binding through a scoped slot?**

Well if you've dug into Vue deep enough, you might know that `v-model` is really just syntax sugar for a `:value` attribute binding, and an `@input` event binding:

```
  <input class="tags-input-text" placeholder="Add tag..."
    @keydown.enter.prevent="addTag"
-   v-model="newTag"
+   :value="newTag"
+   @input="(e) => newTag = e.target.value"
  >
```

That means we can handle this behavior in our renderless component by making a few changes:

- Add a local `newTag` data property to the component
- Pass back an attribute binding prop that binds `:value` to `newTag`
- Pass back an event binding prop that binds `@keydown.enter` to `addTag` and `@input` to update `newTag`

```
  /* Renderless Tags Input Component */
  export default {
    props: ['value'],
+   data() {
+     return {
+       newTag: '',
+     }
+   },
    methods: {
+     addTag() {
+       if (this.newTag.trim().length === 0 || this.value.includes(this.newTag.tri
+         return
+       }
+       this.$emit('input', [...this.value, this.newTag.trim()])
+       this.newTag = ''
+     },
      removeTag(tag) {
        this.$emit('input', this.value.filter(t => t !== tag))
      }
    },
    render() {
      return this.$scopedSlots.default({
        tags: this.value,
        removeTag: this.removeTag,
+       inputAttrs: {
+         value: this.newTag,
+       },
+       inputEvents: {
+         input: (e) => { this.newTag = e.target.value },
+         keydown: (e) => {
+           if (e.keyCode === 13) {
+             e.preventDefault()
+             this.addTag()
+           }
+         }
+       }
      })
    },
  }
```

Now we just need to bind those props to the input element in the parent:

```
  <template>
    <renderless-tags-input>
-     <div slot-scope="{ tags, removeTag }" class="tags-input">
+     <div slot-scope="{ tags, removeTag, inputAttrs, inputEvents }" class="tags-i
        <span class="tags-input-tag">
          <span v-for="tag in tags">{{ tag }}</span>
          <button type="button" class="tags-input-remove"
            @click="removeTag(tag)"
          >&times;</button>
        </span>
-       <input class="tags-input-text" placeholder="Add tag...">
+       <input class="tags-input-text" placeholder="Add tag..."
+         v-bind="inputAttrs"
+         v-on="inputEvents"
+       >
      </div>
    </renderless-tags-input>
  </template>

  <script>
    export default {
      data() {
        return {
          tags: ['Testing', 'Design']
        }
      }
    }
  </script>
```

Adding new tags explicitly

In our current layout, the user adds a new tag by typing it in the field and hitting the enter key. But it's easy to imagine a scenario where someone might want to provide a button that the user can click to add the new tag as well.

Making this possible is easy, all we need to do is pass a reference to our `addTag` method to the slot scope as well:

```
  /* Renderless Tags Input Component */
  export default {
    // ...
    methods: {
      addTag() {
```

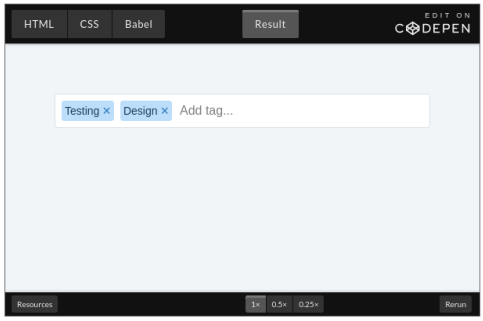
```
        return
      }
      this.$emit('input', [...this.value, this.newTag.trim()])
      this.newTag = ''
    },
    // ...
  },
  render() {
    return this.$scopedSlots.default({
      tags: this.value,
+     addTag: this.addTag,
      removeTag: this.removeTag,
      inputAttrs: {
        // ...
      },
      inputEvents: {
        // ...
      }
    })
  },
}
```

When designing renderless components like this, **it's better to err on the side of "too many slot props" than too few**.

The consumer only needs to destructure out the props they actually need, so there's no cost to them if you give them a prop they aren't going to use.

Working Demo

Here's a working demo of the renderless tags input component that we've built so far:

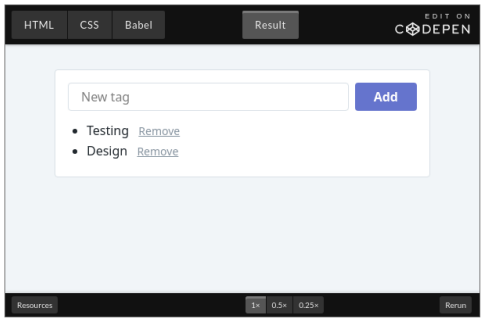


The actual component contains no HTML, and the parent where we define the template contains no behavior. Pretty neat right?

An Alternate Layout

Now that we have a renderless version of the tags input control, we can easily implement alternative layouts by writing whatever HTML we want and applying the provided slot props to the right places.

Here's what it would look like to implement the stacked layout from the beginning of the article using our new renderless component:



Creating Opinionated Wrapper Components

You might look some of these examples and think, *"wow, that sure is a lot of HTML to write every time I need to add another instance of this tags component!"* and you'd be right.

It's definitely a lot more work to write this whenever you need a tags input:

```
<renderless-tags-input v-model="tags">
  <div class="tags-input" slot-scope="{ tags, removeTag, inputAttrs, inputEvents }
    <span class="tags-input-tag" v-for="tag in tags">
      <span>{{ tag }}</span>
      <button type="button" class="tags-input-remove" @click="removeTag(tag)">&tim
    </span>

    <input class="tags-input-text" placeholder="Add tag..." v-on="inputEvents" v-b
  </div>
</renderless-tags-input>
```

...than this, which is what we started with in the beginning:

```
<tags-input v-model="tags"></tags-input>
```

There's an easy fix though: *create an opinionated wrapper component!*

Here's what it looks like to write our original `<tags-input>` component *in terms of* the renderless tags input:

```
<!-- InlineTagsInput.vue -->
```

```
<template>
  <renderless-tags-input :value="value" @input="(tags) => { $emit('input', tags) }
    <div class="tags-input" slot-scope="{ tag, removeTag, inputAttrs, inputEvents
      <span class="tags-input-tag" v-for="tag in tags">
        <span>{{ tag }}</span>
        <button type="button" class="tags-input-remove" @click="removeTag(tag)">&t
      </span>

      <input class="tags-input-text" placeholder="Add tag..."
        v-bind="inputAttrs"
        v-on="inputEvents"
      >
    </div>
  </renderless-tags-input>
</template>

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

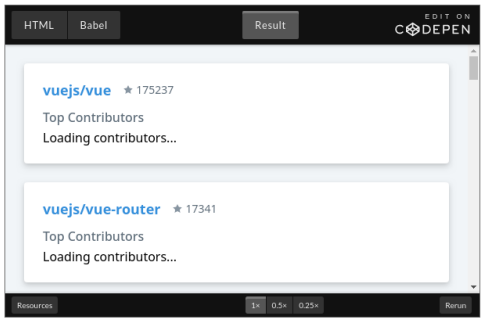
Now you can use that component in one line of code anywhere you need that particular layout:

```
<inline-tags-input v-model="tags"></inline-tags-input>
```

Getting Crazy

Once you realize that a component doesn't have to render anything and can instead be responsible solely for providing data, there's no limit to the type of behavior you can model with a component.

For example, here's a `fetch-data` component that takes a URL as a prop, fetches JSON from that URL, and passes the response back to the parent:



Is this the right way to make every AJAX request? Probably not, but it's certainly interesting!

Conclusion

Splitting a component into a presentational component and a renderless component is an extremely useful pattern to master and can make code reuse a lot easier, but it's not always worth it.

Use this approach if:

- You're building a library and you want to make it easy for users to customize how your component looks
- You have multiple components in your project with very similar behavior but different layouts

Don't go down this path if you're working on a component that's always going to look the same everywhere it's used; it's considerably simpler to keep everything in a single component if that's all you need.

Learning More

If you enjoyed this post, you might be interested in Advanced Vue Component Design, a video series I put together that goes deep into tons of useful component design patterns.

Check out the website to learn more, or sign up below to watch two free preview lessons:

What's your email? Get the course preview