Using Event Bus to Share Props Between Vue Components

BY **KINGSLEY SILAS** ON OCTOBER 10, 2018

VUE, VUE EVENTS, VUE PROPS, VUEX

By default, communication between Vue components happen with the use of props. Props are properties that are passed from a parent component to a child component. For example, here's a component where `title` is a prop:

HTML

```html
<blog-post title="My journey with Vue"></blog-post>
```

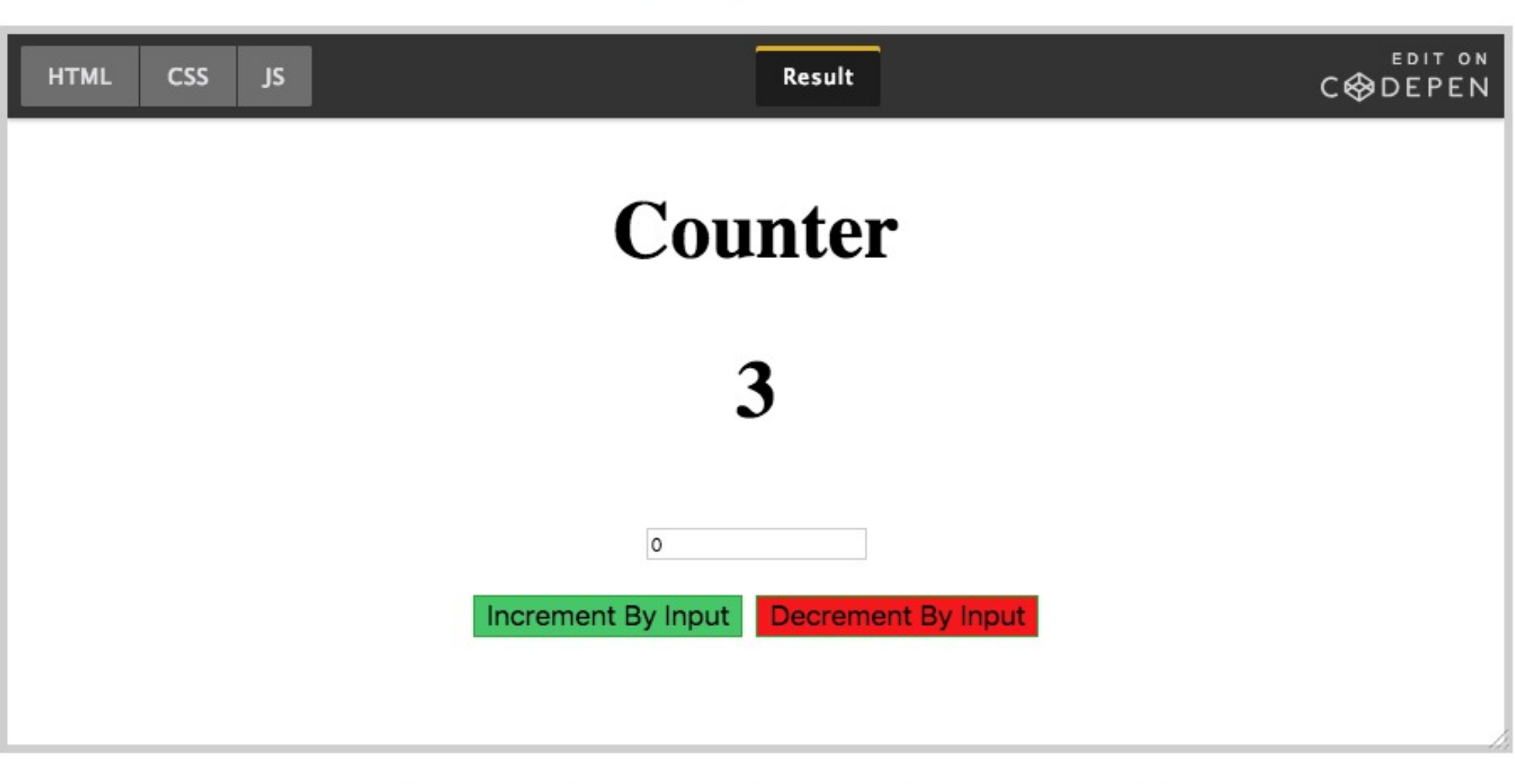
Props are always passed from the parent component to the child component. As your application increases in complexity, you slowly hit what is called prop drilling here's a relate article that is React-focused, but totally applies). **Prop drilling** is the idea of passing props down and down and down to child components — and, as you might imagine, it's generally a tedious process.

So, tedious prop drilling can be one potential problem in a complex. The other has to do with the communication between *unrelated* components. We can tackle all of this by making use of an **Event Bus**.

What is an Event Bus? Well, it's kind of summed up in the name itself. It's a mode of transportation for one component to pass props from one component to another, no matter where those components are located in the tree.

Practice task: Building a counter

Let's build something together to demonstrate the concept of an event bus. A counter that adds or subtracts a submitted value and tallies the overall total is a good place to start:



Counter

3

0

To make use of an event bus, we first need to initialize it like so:

JS

```js
import Vue from 'vue';
const eventBus = new Vue();
```

This sets an instance of Vue to `eventBus`. You can name it anything you'd like, whatsoever. If you are making use of a single-file component, then you should have snippet in a separate file, since you will have to export the Vue instance assigned to `eventBus` anyway:

JS

```js
import Vue from 'vue';
export const eventBus = new Vue();
```

With that done, we can start making use of it in our counter component.

Here's what we want to do:

- We want to have a count with an initial value of 0.
- We want an input field that accepts numeric values.
- We want two buttons: one that will add the submitted numeric value to the count when clicked and the other to subtract that submitted numeric value from the count when clicked.
- We want a confirmation of what happened when the count changes.

This is how the template looks with each of those elements in place:

HTML

```html
<div id="app">
  <h2>Counter</h2>
  <h2>{{ count }}</h2>
  <input type="number" v-model="entry" />
  <div class="div__buttons">
    <button class="incrementButton" @click.prevent="handleIncrement">
      Increment
    </button>
    <button class="decrementButton" @click.prevent="handleDecrement">
      Decrement
    </button>
  </div>
  <p>{{ text }}</p>
</div>
```

We bind the input field to a value called `entry`, which we'll use to either increase or decrease the count, depending on what is entered by the user. When either button is clicked, we trigger a method that should either increase or decrease the value of count. Finally, that `{{ text }}` thing contained in `<p>` tag is the message we'll print that summarizes the change to the count.

Here's how that all comes together in our script:

JS

```js
new Vue({
  el: '#app',
  data() {
    return {
      count: 0,
      text: '',
      entry: 0
    }
  },
  created() {
    eventBus.$on('count-incremented', () => {
      this.text = `Count was increased`
      setTimeout(() => {
        this.text = '';
      }, 3000);
    })
    eventBus.$on('count-decremented', () => {
      this.text = `Count was decreased`
      setTimeout(() => {
        this.text = '';
      }, 3000);
    })
  },
  methods: {
    handleIncrement() {
      this.count += parseInt(this.entry, 10);
      eventBus.$emit('count-incremented')
      this.entry = 0;
    },
    handleDecrement() {
      this.count -= parseInt(this.entry, 10);
      eventBus.$emit('count-decremented')
      this.entry = 0;
    }
  }
})
```

You may have noticed that we're about to hop on the event bus by looking at that code.

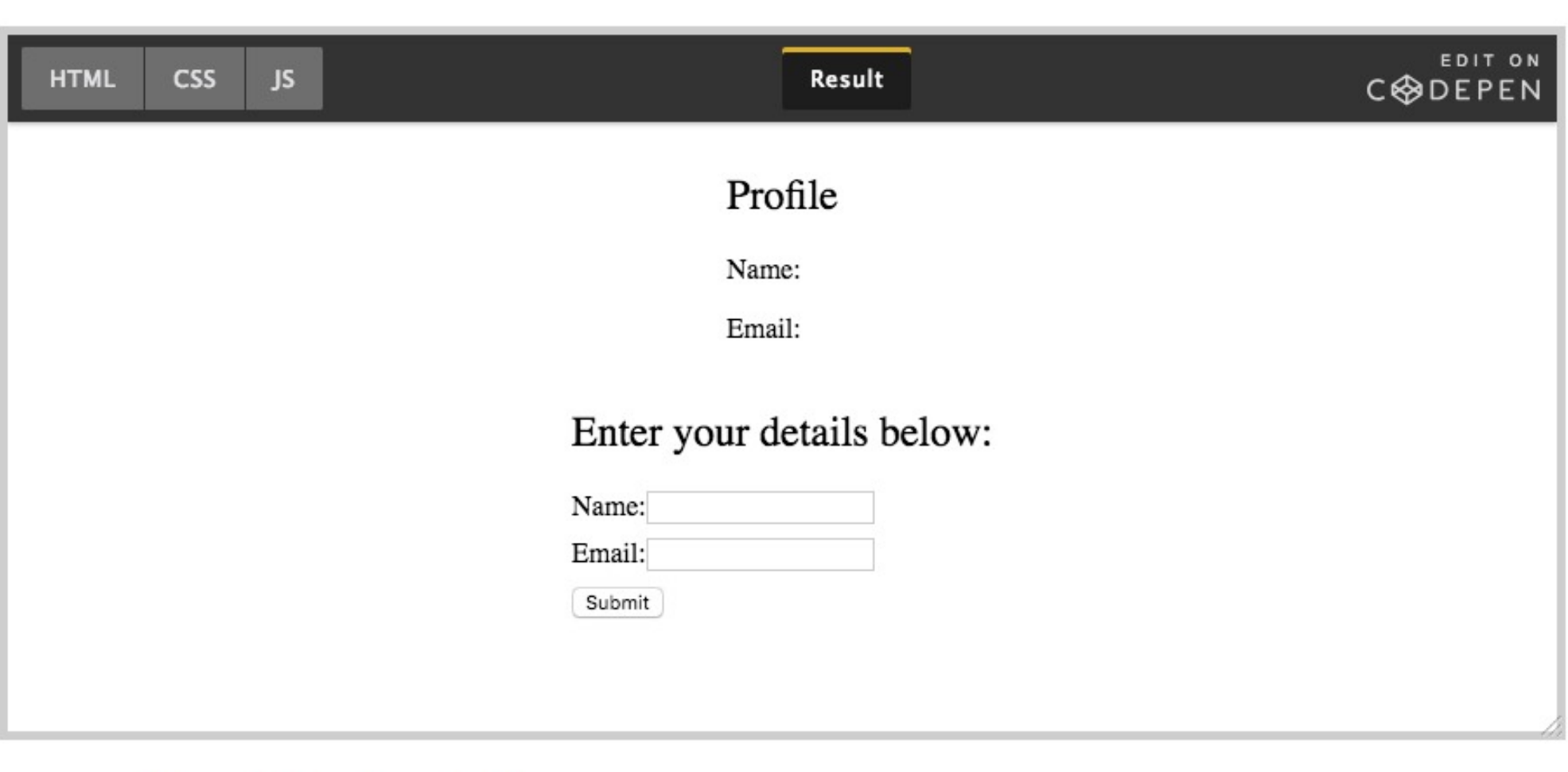
First thing we've got to do is establish a path for sending an event from one component to another. We can pave that path using `eventBus.$emit()` (with emit being a fancy word for sending out). That sending is included in two methods, `handleIncrement` and `handleDecrement`, which is listening for the input submissions. And, once they happen, our event bus races to any component requesting data and sends the props over.

Example: User adds 5 to the initial state of the counter

You may have noticed that we are listening for both events in the `created()` lifecycle hook using `eventBus.$on()`. In both events, we have to pass in the string that corresponds to the event we emitted. This is like an identifier for the particular event and the thing that established a way for a component to receive data. When `eventBus` recognizes a particular event that has been announced, the function that follows is called — and we set a text to display what had happened, and make it it disappear after three seconds.

Practice task: Handling multiple components

Let's say we are working on a profile page where users can update their name and email address for an app and then see the update without refreshing the page. This can be achieved smoothly using event bus, even though we are dealing with two components this time: the user profile and the form that submits profile changes.



Profile

Name:

Email:

Enter your details below:

Name:

Email:

Here is the template:

HTML

```html
<div class="container">
  <div id="profile">
    <h2>Profile</h2>
    <div>
      <p>Name: {{name}}</p>
      <p>Email: {{ email }}</p>
    </div>
  </div>

  <div id="edit__profile">
    <h2>Enter your details below:</h2>
    <form @submit.prevent="handleSubmit">
      <div class="form-field">
        <label>Name:</label>
        <input type="text" v-model="user.name" />
      </div>
      <div class="form-field">
        <label>Email:</label>
        <input type="text" v-model="user.email" />
      </div>
      <button>Submit</button>
    </form>
  </div>
</div>
```

We will pass the `ids` (`user.name` and `user.email`)to the corresponding component. First, let's set up the template for the Edit Profile (`edit__profile`) component, which holds the name and email data we want to pass to the Profile component we'll set up next. Again, we've established an event bus to emit that data after it detects that a submission event has taken place.

JS

```js
new Vue({
  el: "#edit__profile",
  data() {
    return {
      user: {
        name: '',
        email: ''
      }
    }
  },
  methods: {
    handleSubmit() {
      eventHub.$emit('form-submitted', this.user)
      this.user = {}
    }
  }
})
```

This data will be used to reactively update the profile on the user in the Profile (`profile`) component, which looking for `name` and `email` to come in when the bus arrives to its hub.

JS

```js
new Vue({
  el: '#profile',
  data() {
    return {
      name: '',
      email: ''
    }
  },
  created() {
    eventHub.$on('form-submitted', ({ name, email}) => {
      this.name = name;
      this.email = email
    })
  }
})
```

Their bags are packed. Now all they have to do is go home.

Pretty cool, right? Even though the Edit Profile and Profile components are unrelated — or not in a direct parent-child relationship) — it is possible for them to communicate with each other, linked by the same event.

Rollin' right along

I have found Event Bus helpful in cases where I want to enable reactivity in my app — specifically, to update a component based on the response obtained from the server without causing the page to refresh. It is also possible that the event that gets emitted can be listened to by more than one component.

If you have other interesting scenarios of using event bus, I'll love to hear about them in the comments. 🚌