BLOGS

How to Emit Data in Vue: Beyond the Vue.js Documentation

by Sunil Sandhu | September 10, 2018 | JavaScript | 0 Comments

In this blog, we learn how to emit events from child components in Vue, as well as how to emit from nested child components. We'll do all this while avoiding a common anti-pattern that new Vue developers often make.

A core concept behind a lot of modern JavaScript frameworks and libraries is the ability to encapsulate data and UI inside modular, reusable components. This is great when it comes to helping a developer avoid repeating chunks of code throughout an application (or even across apps). However, while the ability to contain functionality inside of a component is great, a component will often need ways to be able to communicate with the outside world or, more specifically, with other components.

We're able to send data down from a parent component via props (short for properties). This is *usually* a fairly straightforward concept to grasp. But what about sending data from a child component back up to its parent?

In Vue, I initially had a bit of difficulty figuring out how to do this—mainly because I feel that Vue's documentation doesn't cover this as well or as thoroughly as it could (which is a shame, because Vue's docs often excel in most other areas).

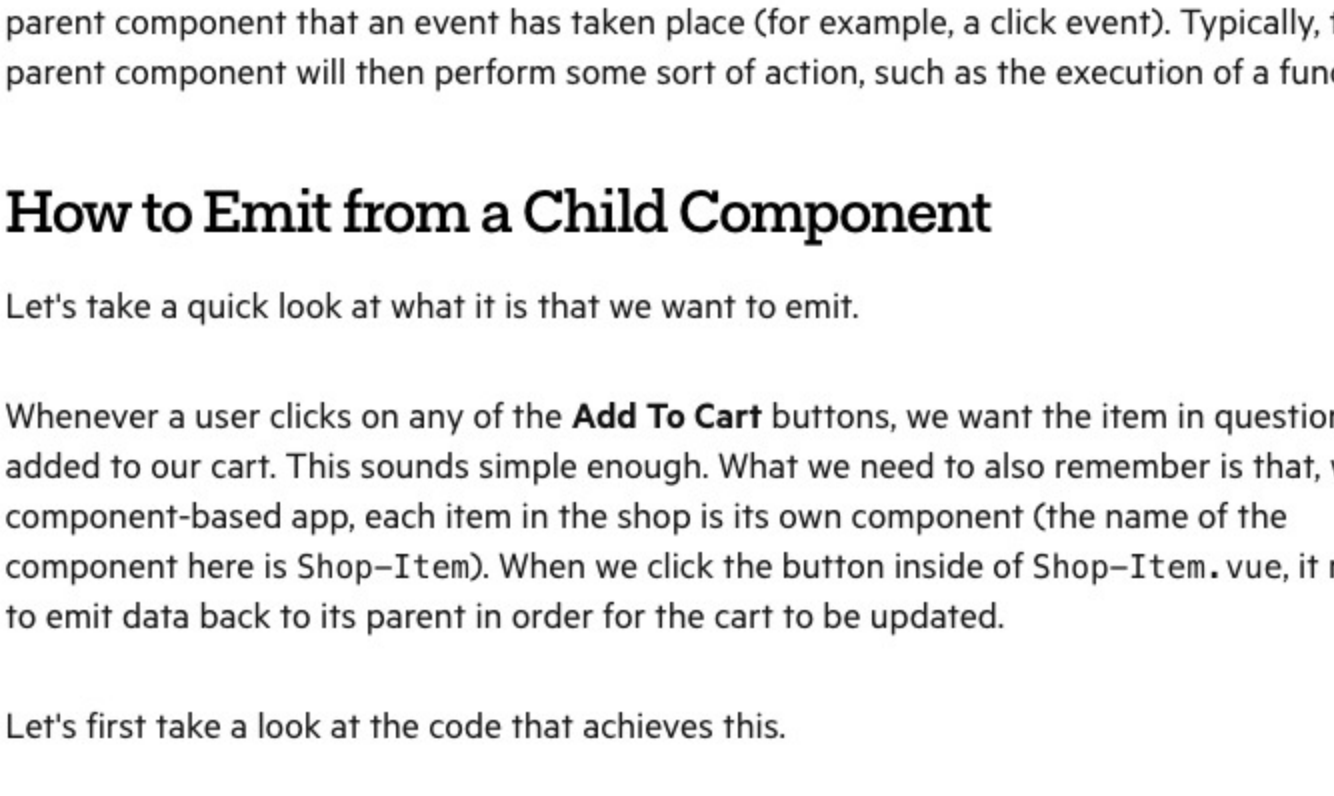
After a lot of Googling (and trial and error), I ended up figuring out a way to send data upwards from child to parent, but after a while, I found out from a fellow developer that I had been doing this in completely the wrong way—it worked, but I was committing a cardinal sin in the world of anti-patterns.

With all of this in mind, we're going to write an article that hopefully helps future Vue developers find a clear answer for "how to emit data in Vue" a little easier, while building a neat little Shopping Cart app along the way.

Setup

We'll be using the Vue CLI to quickly get some boilerplate code set up, as well as all of the other nice things it brings, such as hot-module-reloading, auto-compiling, etc. Don't worry if any of this goes over your head, just get used to using it because it's great! 💪

We'll try not to spend too much time going through any further set-up, as the focus here is to show you how to emit data, rather than showing you a step-by-step set-up of our Shopping Cart app. By all means, feel free to try and build one yourself though with the code examples littered throughout the article.



Visit cli.vuejs.org for more info on how to install and use Vue CLI.

The finished Shopping Cart app built for this article can also be found here: github.com/sunil-sandhu/vue-cart-emit-example.

What Even Is Emit?

A definition from Cambridge Dictionary tells us that the formal definition of "emit" is "to send out a beam, noise, smell or gas." Don't worry, our app won't be emitting any strange smells or gases! In our case, the aim is to "emit" a signal—a signal from a child component to notify a parent component that an event has taken place (for example, a click event). Typically, the parent component will then perform some sort of action, such as the execution of a function.

How to Emit from a Child Component

Let's take a quick look at what it is that we want to emit.

Whenever a user clicks on any of the **Add To Cart** buttons, we want the item in question to be added to our cart. This sounds simple enough. What we need to also remember is that, with a component-based app, each item in the shop is its own component (the name of the component here is `Shop-Item`). When we click the button inside of `Shop-Item.vue`, it needs to emit data back to its parent in order for the cart to be updated.

Let's first take a look at the code that achieves this.

```
<!-- Shop-Item.vue -->

<template>
  <div class="Item">
    <img :src="item.imageSrc" :alt="item.name" class="ItemImage">
    <div class="ItemDetails">
      <p><strong>{{item.name}}</strong></p>
      <p>Price: <strong>${{item.price}}</strong></p>
    </div>
    <button class="Button" @click="addToCart(item)">Add To Cart</button>
  </div>
</template>
<script>
  export default {
    name: 'Shop-Item',
    props: ['item'],
    data() {
      return {}
    },
    methods: {
      addToCart(item) {
        this.$emit('update-cart', item)
      }
    }
  }
</script>
<style>
</style>
```

```
<!-- App-Item.vue -->

<template>
  <div id="app">
    <section class="Header">
      <h1 id="Fruiticious!">Fruiticious!</h1>
      <!-- Cart component -->
      <shop-cart :cart="this.cart" :total="this.total" @empty-cart="empty
      </shop-cart>
    </section>
    <!-- Item component -->
    <shop-item v-for="item in this.items" :item="item" :key="item.id" @u
    </shop-item>
  </div>
</template>
<script>
  export default {
    name: 'app',
    data() {
      return {
        items: [
          { id: 205, name: 'Banana', price: 1, imageSrc: Banana },
          { id: 148, name: 'Orange', price: 2, imageSrc: Orange },
          { id: 248, name: 'Apple', price: 1, imageSrc: Apple }
        ],
        cart: [],
        total: 0
      }
    },
    methods: {
      updateCart(e) {
        this.cart.push(e);
        this.total = this.shoppingCartTotal;
      },
      emptyCart() {
        this.cart = [];
        this.total = 0;
      }
    },
  }
</script>
```

Let's break this down further and just show the highlighted parts and explain how the click of a button sets off a chain of events.

First, we have a button in `Shop-Item.vue`:

```
<button class="Button" @click="addToCart(item)">Add To Cart</button>
```

Each item in the shop (Banana, Orange, Apple) has one of these buttons. When it gets clicked, our `@click="addToCart(item)` event listener is triggered. You can see that it takes the item in as a parameter (this is the entire item object which has been passed into `<Shop-Item>` as a prop.) When the button is clicked, it triggers the `addToCart` function:

```
addToCart(item) {
  this.$emit('update-cart', item)
}
```

We see that this function fires `this.$emit`. What does that even mean? Well, emit simply sends a signal. In this case, the signal is 'update cart', which is sent in the form of a string. So essentially, `this.$emit` takes a `string` as its first parameter. It can also accept a second parameter, which will usually take the form of some piece of data that we want to send along with it. This could be another `string`, an `integer`, a variable, an `array`, or, in our case, an `object`.

But then how does sending that `string` of `"update-cart"` notify our parent component that the Shopping Cart needs to be updated? Well, let's look at the third piece of the jigsaw.

When we add our `<shop-item>` tag in `App.vue`, we also add a custom event listener onto it that listens out for `update-cart`. In fact, it actually looks similar to our `@click` event listener that was on the '**Add To Cart**' buttons.

```
<shop-item v-for="item in this.items"
           :item="item"
           :key="item.id"
           @update-cart="updateCart">
</shop-item>
```

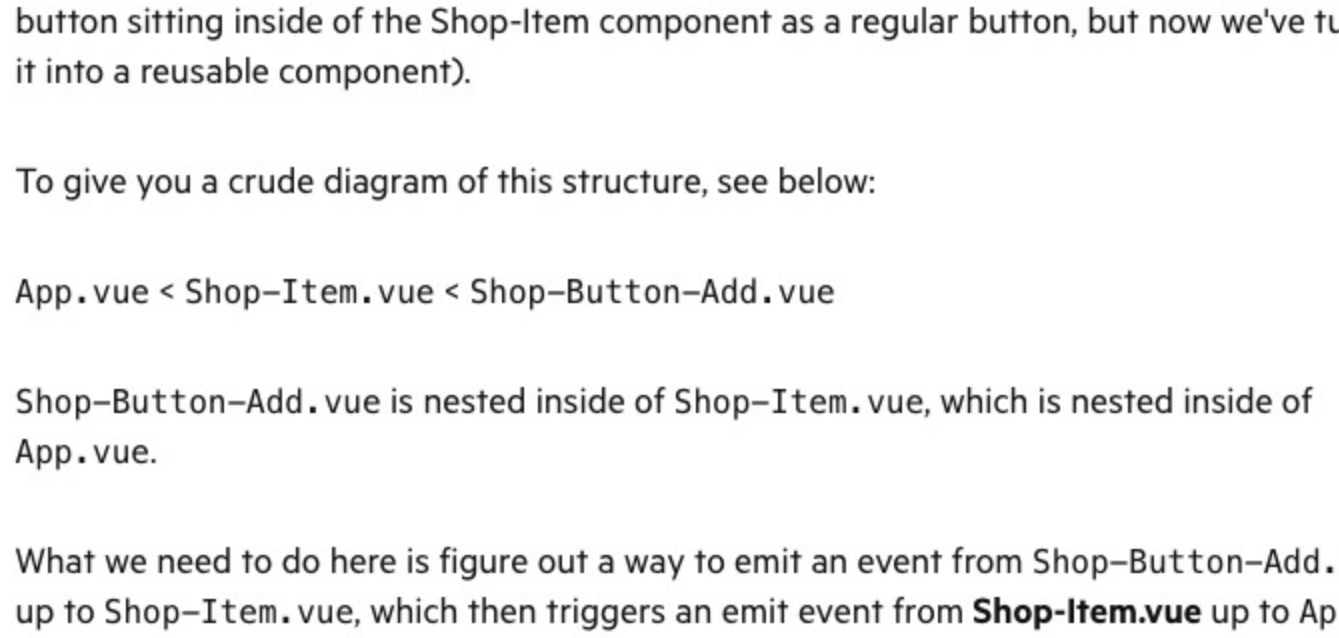
We see here that our custom event listener is waiting for the `update-cart` event to be fired. And how does it know when this happens? When the string 'update-cart' is emitted from inside the `Shop-Item.vue`!

The final bit is to now see what happens when this `@update-cart` event listener triggers the `updateCart` function:

```
updateCart(e) {
  this.cart.push(e);
  this.total = this.shoppingCartTotal;
}
```

This simply takes an event parameter and pushes it into the `this.cart` array. The event that it takes is simply the item that we initially put in as the second parameter when we called `this.$emit`. You can also see that `this.total` is also updated to return the result of the `this.shoppingCartTotal` function (check out the Github repository for more info on how it does this).

And that is how we emit from a child component back to the parent component. We can even see this take place inside of Vue Developer Tools (an essential piece of kit if you use Chrome and you're working with Vue components). When the "Add To Cart" button is pressed for the Banana, all of the info in the screenshot below is rendered:



This is the output in Vue DevTools after we click the Banana's "Add To Cart" button.

Awesome, we now know how to correctly emit from a child component back to the parent!

But what if we have lots of sub components? What if we have child components sitting inside of other child components? How do we emit a message all the way back up to the parent (or grandparent, if that makes it easier for you to visualize)? Let's tackle that next!

How to Emit From a Nested Child Component (i.e. Grandchild to Grandparent)

Okay, so taking the same example that we used when emitting from child up to parent, we're going to take this one step further. In our finished code, we actually had the "Add To Cart" button as its own component, which sits inside of `Shop-Item.vue` (before we just had the button sitting inside of the Shop-Item component as a regular button, but now we've turned it into a reusable component).

To give you a crude diagram of this structure, see below:

`App.vue` < `Shop-Item.vue` < `Shop-Button-Add.vue`

`Shop-Button-Add.vue` is nested inside of `Shop-Item.vue`, which is nested inside of `App.vue`.

What we need to do here is figure out a way to emit an event from `Shop-Button-Add.vue` up to `Shop-Item.vue`, which then triggers an emit event from **Shop-Item.vue** up to `App.vue`. Sounds a bit complicated, but it's actually easier than you'd think.

Here are the code blocks that make it happen.

In `Shop-Button-Add.vue`:

```
<button class="Button" @click="buttonClicked">
```

Which triggers this method in the same file:

```
methods: {
  buttonClicked() {
    this.$emit('button-clicked')
  }
}
```

Inside of `Shop-Item.vue`, we attach a `@button-clicked` listener onto the `<shop-button-add>` tag:

```
<shop-button-add
  @button-clicked="addToCart(item)"
  :item="item">
  <p>Add To Cart</p>
</shop-button-add>
```

We see here that we're also passing in the item object as a parameter (exactly the same as what we did in our previous example). This `@button-clicked` event listener fires the following function in the same file:

```
methods: {
  addToCart(item) {
    this.$emit('update-cart', item)
  }
}
```

Inside of `App.vue`, we attach an `@update-cart` listener onto the `<shop-item>` tag:

```
<shop-item v-for="item in this.items"
           :item="item"
           :key="item.id"
           @update-cart="updateCart">
</shop-item>
```

Finally, this triggers the `updateCart` function which sits in `App.vue`, as such:

```
methods: {
  updateCart(e) {
    this.cart.push(e);
    this.total = this.shoppingCartTotal;
  }
}
```

Which pushes the item object into the cart. And that's how we emit from nested components!

But what about super-deeply nested components (eg. Great-Great-Great-Great-Grandchild to Great-Great-Great-Great-Grandparent)?

Well, we have three options here:

1. You could **emit your event all the way back up the chain** (although this can start to get quite messy if you're having to emit any further than grandchild to grandparent).
2. You could use a dedicated state management system such as **Vuex**, which can help to simplify the process of emitting from deeply nested components. I'd certainly recommend this route and we'll be definitely look to cover this in a future post!
3. Or you could use something known as a **Global Event Bus**. You can think of this as implementing your own simplified version of a state management system such as Vuex. However, it is worth noting that Vue's core team generally advises against the use of Global Event Buses in favor of something more robust, such as Vuex. We won't go into the reasons for this any further here, but it's certainly worth researching further if this is something you are considering in your app.

Anti-Pattern Culprits

The reason why it's really important to get our emit event listeners set up properly is because we are ultimately trying to encapsulate our components as best as possible. In the case of a button, the more reusable we can make it, the more transportable it becomes. If our button emits a simple 'button-clicked' string, we can then decide what we want that emit event to trigger on a per application basis—we could even have it trigger different things inside of the same application, based on where we decide to use it.

As mentioned at the start of this article, when I first figured out how to emit events, I ended up using the following two syntaxes: `this.$parent.$emit` and `this.$root.$emit`.

Although they look similar to `this.$emit`, they're different in the sense that the `this.$parent.$emit` emits the event inside of the parent component, while `this.$root.$emit` emits the event inside of the root component *(which in our example would have been `App.vue`).*

So to expand on this a little, if we take our **Shop-Button-Add** component, this emits a signal upward to **Shop-Item**, through the use of `this.$emit`. However, if we opted to use **this.$parent.$emit,** this will actually be telling **Shop-Item** to emit an event instead. Effectively, the **Shop-Button-Add** is now telling its parent **Shop-Item** to emit an event, rather than following the proper pattern of event emitting.

It can seem a little bit confusing to wrap your head around sometimes, and, to be fair, in our example it may actually make sense to skip a step and go for `this.$parent.$emit`. However, the problem here is that our `Shop-Button-Add` is no longer truly encapsulated, because it now relies on always being inside of `Shop-Item` for it to work. Again, this may seem okay in the case of our simple Shopping Cart application, but what if we wanted to generalize our button a bit and simply make it a `Shop-Button` that gets used across our application for lots of different things, such as increasing/decreasing quantities, emptying our cart, etc. It would get very messy and very confusing very fast!

To quickly summarise `this.$parent` and `this.$root`:

- `this.$emit` dispatches an event to its parent component
- `this.$parent` gives you a reference to the parent component
- `this.$root` gives you a reference to the root component
- `this.$parent.$emit` will make the parent dispatch the event to its parent
- `this.$root.$emit` will make the root dispatch the event to itself

Conclusion

And there we have it! We now know how to successfully emit events and data from child components, and even nested child components, all the way back to the parent. We have also learned about the existence of this.$parent and this.$root, but why they should be avoided and are considered to cause an anti-pattern. Following on, I highly recommend listening to this episode of Full Stack Radio, where Chris Fritz, a member of the Vue Core Team, talks further about common anti-patterns he has noticed being used out in the wild.

If you found this useful, be sure to share, and feel free to reach out to me on Twitter to discuss further.

For More Vue

Want to learn more about Vue? Check out the video series on Getting Started with Kendo UI and Vue to learn how to create a great UI in Vue, or just take a look at the Kendo UI for Vue component library.

guide, components, app dev, Vue, JavaScript

ABOUT THE AUTHOR

Sunil Sandhu

Sunil Sandhu is a Full Stack Web Developer and Editor of Javascript In Plain English. When not building or writing about interactive experiences with the latest JS frameworks, Sunil acts as an advisor to start-ups and helps to train new developers.