Vue.js Component Communication Patterns

alligator.io/vuejs/component-communication

There are several methods that can be used for inter-component communication in Vue.js. Normal *props* and *events* should be sufficient for most cases, but there are other methods available at your disposal as well.

Props & Events

Of course, the normal method for communication involves props and events. This common pattern provides a powerful way of communicating between components without introducing any dependency or limitations on which components are involved.

Props:

Props allow you to pass any data type to a child component, and allow you to control what sort of data your component receives. Prop updates are also reactive, allowing a child component to update whenever parent data changes.

Template Usage:

```
<my-component v-bind:prop1="parentValue"></my-component>
```

```
<my-component :prop1="parentValue"></my-component>
```

Events:

Events provide a way to inform your parent components of changes in children.

Template Usage:

```
<my-component v-on:myEvent="parentHandler"></my-component>
```

```
<my-component @myEvent="parentHandler"></my-component>
```

Firing an Event:

```
...
export default {
  methods: {
    fireEvent() {
      this.$emit('myEvent', eventValueOne, eventValueTwo);
    }
  }
}
```

Additionally, you can create global event buses to pass events anywhere in your app. We've got an article on that.

Combined:

Using v-model allows for combining props with events for two-way binding. This is often used for input components. *v-model* assumes the *value* prop and *input* event, but this can be customized.

Template Usage:

```
<my-component v-model="prop1"></my-component>
```

A v-model compatible component:

```
<template>
  <div>
    <input type="text" :value="value" @input="triggerEvent"/>
  </div>
</template>

<script>
  export default {
    props: {
      value: String
    },

    methods: {
      triggerEvent(event) {
        this.$emit('input', event.target.value);
      }
    }
  }
</script>
```

Use When: You need to do pretty much any sort of data passing and messaging between components.

Provide / Inject

A much newer addition to *Vue* is the provide / inject mechanism. It allows for selective exposition of data or methods from an ancestor component to all of its descendants. While *provide / inject* is not itself reactive, it can be used to pass reactive objects.

provide / inject is probably not a good idea to develop an app with, but it can come in quite handy when writing whole custom-rendered component libraries.

Ancestor Component:

```
const SomethingAllDescendantsNeed = 'Air, probably.';

export default {
  provide: {
    SomethingAllDescendantsNeed
  }
}
```

Descendant Component(s):

```
export default {
  inject: ['SomethingAllDescendantsNeed'],

  mounted() {
    console.log(this.SomethingAllDescendantsNeed);
  }
}
```

Template Usage:

```
<ancestor-component>
  <div>
    <descendant-component>
      <p>
        <descendant-component></descendant-component>
      </p>
    </descendant-component>
  </div>
</ancestor-component>
```

(All descendant components, no matter how deep in the tree, have access to *SomethingAllDescendantsNeed*.)

Use When: Child components need access to an instance of something that's only instantiated once per component tree. (Perhaps another library or an event bus.)

Direct Access

CAUTION: HERE BE SHARKS!

If you **really, really, really, neeeeed** to access a property or method directly on a parent or child component, you can use every component's *this.$parent* and *this.$children* properties to have full access to everything on parent and children components. This is, however, and **absolutely, horribly, despicably, terrible idea**. If you find yourself in a situation where you need to do this, there's a *99.99958%* chance you did something wrong and should refactor.

Use When: DON'T. JUST DON'T.

Why not? Because you are introducing a direct coupling between both the implementation and structure in markup between parent and children components, making them inflexible and ridiculously easy to break.