Vue.js + Event Pub/Sub

Introduction

Vue (pronounced /vjuː/, like view) is a progressive framework for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects. On the other hand, Vue is also perfectly capable of powering sophisticated Single-Page Applications when used in combination with modern tooling and supporting libraries.

Event Pub/Sub is an application architecture that was designed to break down communication into events. Everything can be handled via a publish to send or a subscribe to listen. Each component is isolated and does not need to pass information directly or know about another component.

Virtual DOM

The virtual DOM is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM by a library such as ReactDOM. This process is called reconciliation. Vue.js has a a (VDOM) as well though is considered to be faster and is declarative in comparison. For a comparison with React as well as other frameworks you can read about it here.

Declarative Rendering

At the core of Vue.js is a system that enables us to declaratively render data to the DOM using straightforward template syntax.

Every Vue application starts by creating a new Vue instance with the Vue function:

```
<div class="v-counter">
  Counter: {{ counter }}
</div>
```

```
var vCounter = new Vue({
  el: '.v-counter,
  data: {
    counter: 0
  }
})
```

We have already created our very first Vue instance! This looks pretty similar to rendering a counter, but Vue has done a lot of work under the hood. The data and the DOM are now linked, and everything is now reactive.

Note that we no longer have to interact with the HTML directly. A Vue instance attaches itself to a single DOM element (.v-counter in our case) then fully controls it. The HTML is our entry point, but everything else happens within the newly created Vue instance.

Although not strictly associated with the MVVM pattern, Vue's design was partly inspired by it. As a convention, we often use the variable vm (short for ViewModel) to refer to our Vue instance. When you create a Vue instance, you pass in an options object.

Conditionals and Loops

Vue includes conditionals and looping to help wih updating view dynamically based on data passed to it. Like the following.

```
<div class="v-instance">
  <span v-if="seen">Now you see me</span>
</div>
```

```
var vInstance = new Vue({
  el: '.v-instance',
  data: {
    seen: true
  }
})
```

This example demonstrates that we can bind data to not only text and attributes, but also the structure of the DOM. Moreover, Vue also provides a powerful transition effect system that can automatically apply transition effects when elements are inserted/updated/removed by Vue.

There are quite a few other directives, each with its own special functionality. For example, the v-for directive can be used for displaying a list of items using the data from an Array:

```
<div class="v-instance">
  <ol>
    <li v-for="todo in todos">
      {{ todo.text }}
    </li>
  </ol>
</div>

var vInstance = new Vue({
  el: '.v-instance',
  data: {
    todos: [
      { text: 'Learn JavaScript' },
      { text: 'Learn Vue' },
      { text: 'Build something awesome' }
    ]
  }
})
```

Input

To let users interact with your app, we can use the `v-on` directive to attach event listeners that invoke methods on our Vue instances:

```
<div class="v-instance">
  <p>{{ message }}</p>
  <button v-on:click="reverseMessage">Reverse Message</button>
</div>

var vInstance = new Vue({
  el: '.v-instance',
  data: {
    message: 'Hello Vue.js!'
  },
  methods: {
    reverseMessage: function () {
      this.message = this.message.split('').reverse().join('')
    }
  }
})
```

Data and Methods

When a Vue instance is created, it adds all the properties found in its data object to Vue’s reactivity system. When the values of those properties change, the view will “react”, updating to match the new values.

```
// Our data object
var data = { a: 1 }

// The object is added to a Vue instance
var vm = new Vue({
  data: data
})

// Getting the property on the instance
// returns the one from the original data
vm.a == data.a // => true

// Setting the property on the instance
// also affects the original data
vm.a = 2
data.a // => 2

// ... and vice-versa
data.a = 3
vm.a // => 3
```

When this data changes, the view will re-render. It should be noted that properties in data are only reactive if they existed when the instance was created. That means if you add a new property, like:

```
vm.b = 'hi'
```

Instance Lifecycle Hooks

Each Vue instance goes through a series of initialization steps when it's created - for example, it needs to set up data observation, compile the template, mount the instance to the DOM, and update the DOM when data changes. Along the way, it also runs functions called lifecycle hooks, giving users the opportunity to add their own code at specific stages.

For example, the created hook can be used to run code after an instance is created:

```
new Vue({
  data: {
    a: 1
  },
  created: function () {
    // `this` points to the vm instance
    console.log('a is: ' + this.a)
  }
})
```

For reference please use this diagram to see the complete lifecycle of a Vue instance here.

Template Syntax

Vue.js uses an HTML-based template syntax that allows you to declaratively bind the rendered DOM to the underlying Vue instance's data. All Vue.js templates are valid HTML that can be parsed by spec-compliant browsers and HTML parsers.

Under the hood, Vue compiles the templates into Virtual DOM render functions. Combined with the reactivity system, Vue is able to intelligently figure out the minimal number of components to re-render and apply the minimal amount of DOM manipulations when the app state changes.

This list includes a vast assortment like Interpolation and directives. For reference you can see them all here.

Partials/Instances

For each part of the page, developers create separate partial Vue instances scoped to a subset of data from the Central Store Vue.js instance. These partials should be relatively simple implementation wise:

- Tied to a specific DOM element (specific section of HTML markup)
- Should only one-way bind and use/present the data that they are concerned with
- Publish and subscribe to events

The following is an example of a partial Vue instance that can show message indicators:

```
const el = '.v-snackbar';
const $el = $(el);
if (!$el.length) {
return;
}
const vSnackbar = new Vue({
el: el,
data: {
      data: {
            msg: ''
      },
      state: {
            show: false,
            error: false
      }
},
methods: {

},
mounted: function () {
      vApp.$on('snackbar:show', ({ msg, err }) => {
            this.$data.data.msg = msg || _.get(snackbarCfg, 'fallbackMsg');
            this.$data.state.err = err;
            this.$data.state.show = true;
            _.delay(() => this.$data.state.show = false, 3000);
      });
},
beforeDestroyed: function () {
      vApp.$off('snackbar:show');
}
});

window.vSnackbar = vSnackbar;

<div class="v-snackbar">
    <div class="snackbar" v-if="state.show" v-bind:class="{show: state.show,
error: state.err}">
        <div v-text="data.msg"></div>
    </div>
</div>
```

The Vue instance here manages a global message indicator for that app.

State Management

It is often overlooked that the source of truth in Vue applications is the raw data object - a Vue instance only proxies access to it. Therefore, if you have a piece of state that should be shared by multiple instances, you can share it by identity:

```
var sourceOfTruth = {}

var vmA = new Vue({
  data: sourceOfTruth
})

var vmB = new Vue({
  data: sourceOfTruth
})
```

Now whenever sourceOfTruth is mutated, both vmA and vmB will update their views automatically. Subcomponents within each of these instances would also have access via this.$root.$data. We have a single source of truth now, but debugging would be a nightmare. Any piece of data could be changed by any part of our app at any time, without leaving a trace.

To help solve this problem, we can adopt a store pattern:

```
var store = {
  debug: true,
  state: {
    message: 'Hello!'
  },
  setMessageAction (newValue) {
    if (this.debug) console.log('setMessageAction triggered with', newValue)
    this.state.message = newValue
  },
  clearMessageAction () {
    if (this.debug) console.log('clearMessageAction triggered')
    this.state.message = ''
  }
}
```

Notice all actions that mutate the store’s state are put inside the store itself. This type of centralized state management makes it easier to understand what type of mutations could happen and how they are triggered. Now when something goes wrong, we'll also have a log of what happened leading up to the bug.

In addition, each instance/component can still own and manage its own private state:

```
var vmA = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})

var vmB = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})
```

Event Pub/Sub

This allows different instances to communicate it without needing to have a direct reference or share data. This is made possible via an event bus that allows messages to be sent and received via a publish and subscribe architecture.

```
mounted: function () {
      vApp.$on('snackbar:show', ({ msg, err }) => {
            this.$data.data.msg = msg || _.get(snackbarCfg, 'fallbackMsg');
            this.$data.state.err = err;
            this.$data.state.show = true;
            _.delay(() => this.$data.state.show = false, 3000);
      });
},
beforeDestroyed: function () {
      vApp.$off('snackbar:show');
}
```

The example snippet from our snackbar above creates a subscript that the vue instance can listen to. This allows it to react to message received from the bus. For example if we have a need to show a error message the following would trigger the snackbar:

```
_.delay(() => vApp.$emit('snackbar:show', {
    msg: _.get(snackbarCfg, `addToCartCatch.${cartName === 'Default' ? 'cart'
: 'wishlist'}`),
    err: true
}));
```

Services

Allows for data communication and business logic to be handled in a event bus i.e pub/sub approach. The following create a global Vue instance that can be accessed anywhere. In addition and more importantly it allows service to subscribe to this global event bus and listen to and react to incoming messages.

```
const vApp = new Vue({});
window.vApp = vApp;
const vApp = window.vApp;
vApp.$on('cart:getCart', function () {
    const state = { active: false };
    return (req) => {
        const cartName = req && req.cartName;
        if (state.active) { return; }
        // state.active = true;
        console.log('cart:getCart');
        vApp.$emit('loadingBar:show');
        return $.ajax({
            url: `/api/cart/${cartName ? cartName : 'Default'}`,
            type: 'GET',
            contentType: "application/json",
            dataType: 'json',
        }).then((res) => {
            state.active = false;
            state.cart = res;
            console.log(`cart:getCart${cartName ? (':' + cartName) :
''}:then`, res);
            vApp.$emit(`cart:getCart${cartName ? (':' + cartName) : ''}:then`,
res);
            vApp.$emit('loadingBar:hide');
        });
    }
}());
```

This allows vue instances / partials to emit/trigger requests to this service and the service can internally manage it's business logic and how it reacts. In addition this service can now be subscribed to anywhere in the application and each subscriber can handle the data being emitted from this service however it needs to. Examples include a cart view and a cart icon in the header.

References

- https://vuejs.org/
- https://github.com/vuejs/awesome-vue
- https://www.digitalocean.com/community/tutorials/vuejs-global-event-bus