Vue.JS

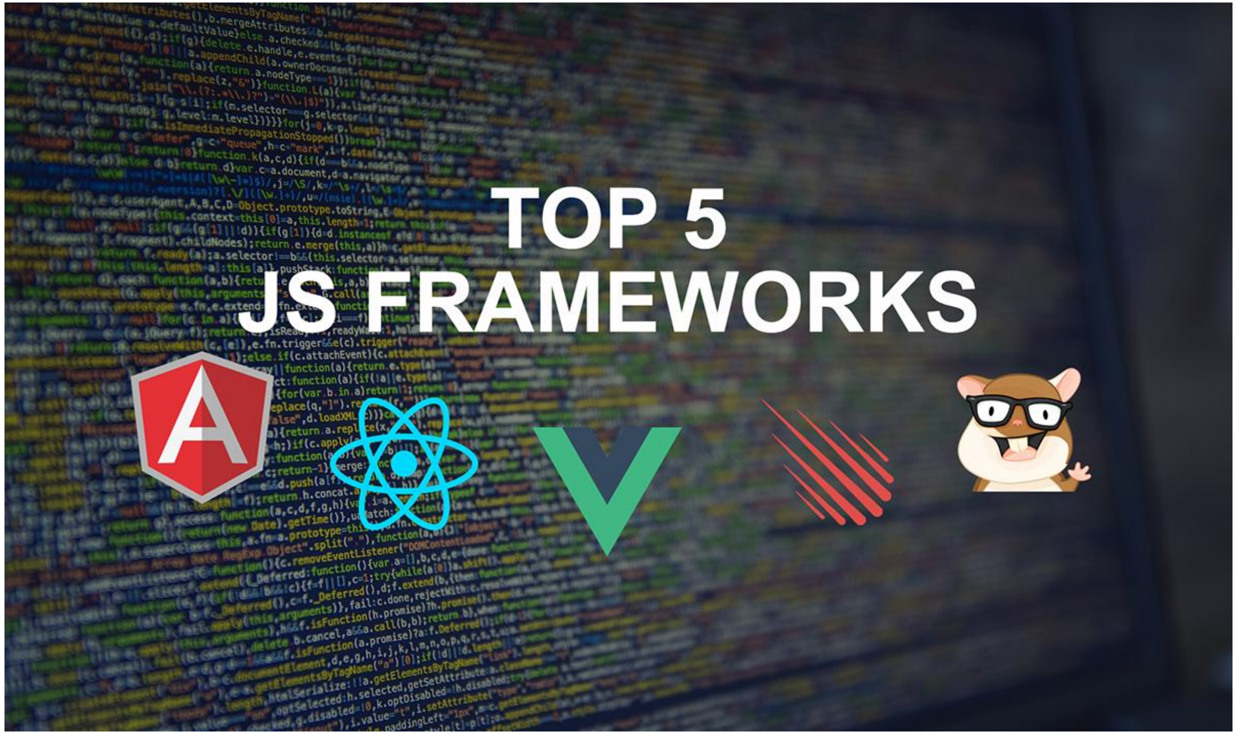
GOCODE

Why we need JS Framework?

- For big projects - vanilla Javascript or Jquery is not enough.

- Keep your code organized with a good structure, easy to extend and test.
- Use tools that implemented many use-cases for SPA (Single Page Application)
- **Declarative** approach over **imperative**:
  Don't talk with the DOM element - just with the model..

What are the JS Frameworks nowadays?



So Why Vue.JS?

- Very easy to learn.
- Good Separation to HTML, CSS and JS.
- Support for using TypeScript, SASS but not must.
- Great ecosystem and 3rd party libraries.
- Powerful reactivity system but with very small core size.

Getting started - The Vue.js Instance

Using CDN (No NPM Build System)

```html
<!-- development version, includes helpful console warnings -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

In HTML:

```html
<div id="app">
 {{ message }}
</div>
```

In JS:

```js
var app = new Vue({
 el: '#app',
 data: {
   message: 'Hello Vue!'
 }
})
```

Output:

Hello Vue!

Declarative render data to DOM

In HTML:

```
<div id="app">
 {{ message }}
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
   message: 'Hello Vue!'
 }
})
```

Every change to "app" will be auto rendered into the DOM.

For Example - Open Console and type:

app.message = 'I Changed the message!'

Data object

In HTML:

```
<div id="app">
 {{ visitCount }}
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
  newTodoText: '',
  visitCount: 0,
  hideCompletedTodos: false,
  todos: [],
  error: null
 }
})
```

Vue instance **created**

and adds all the properties

found in its data object to Vue’s reactivity system.

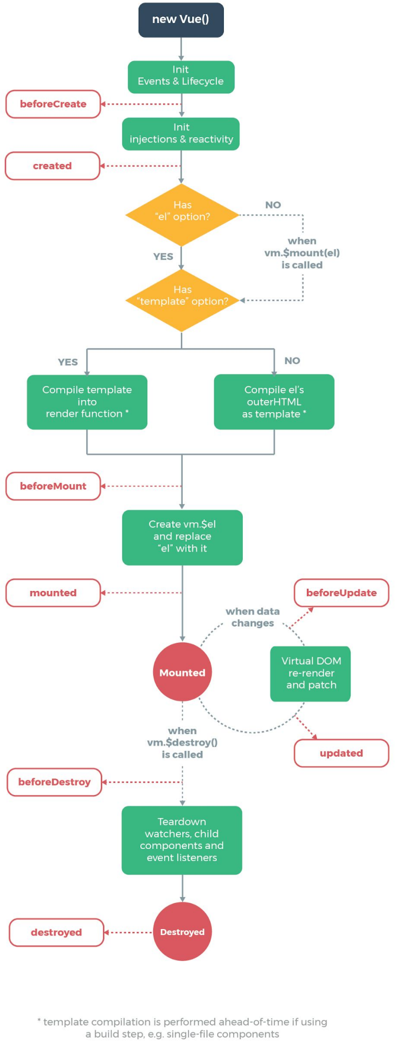
When the values of those properties **change**, the view will “**react**”, updating to

match the new values.

Life Cycle

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

JSFIDDLE



new Vue()
Init Events & Lifecycle
beforeCreate
Init injections & reactivity
created
Has "el" option?
NO
when vm.$mount(el) is called
YES
Has "template" option?
YES
NO
Compile template into render function *
Compile el's outerHTML as template *
beforeMount
Create vm.$el and replace "el" with it
mounted
beforeUpdate
when data changes
Mounted
Virtual DOM re-render and patch
updated
when vm.$destroy() is called
beforeDestroy
Teardown watchers, child components and event listeners
destroyed
Destroyed
* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

Data binding using v-bind directive

In HTML:

```
<div id="app">
 <span v-bind:title="message">
   Hover your mouse over me for a few seconds
   to see my dynamically bound title!
 </span>
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
   message: 'You loaded this page on ' + new
Date().toLocaleString()

 }
})
```

"v-bind" keyword can be removed so this is the same code:

```
<div id="app">
 <span :title="message">
   Hover your mouse over me for a few seconds to see my dynamically bound title!
 </span>
</div>
```

Conditions with v-if

In HTML:

```
<div id="app">
 <span v-if="show">Now you see me</span>
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
   show: true
 }
})
```

Open Console and type:

```
app.show = false
```

And look at the elements inspector

Conditional Groups with v-if on <template>

In HTML:

```html
<div id="app">
  <template v-if="ok">
    <h1>Title</h1>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
  </template>
</div>
```

In JS:

```js
var app = new Vue({
 el: '#app',
 data: {
   ok: true
 }
})
```

`<template> serves as an invisible wrapper.`

`The final rendered result will not include the <template> element.`

v-if, v-else-if, v-else

In HTML:

```
<div v-if="type === 'A'">
 A
</div>
<div v-else-if="type === 'B'">
 B
</div>
<div v-else-if="type === 'C'">
 C
</div>
<div v-else>
 Not A/B/C
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
   type: 'A'
 }
})
```

v-show

In HTML:

```
<div id="app">
  <h1 v-show="ok">Hello!</h1>
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
   ok: true
 }
})
```

- v-if injects/removes the element from the DOM.
- v-show only toggles the display CSS property of the element.
- Prefer v-show if you need to toggle something very often.
- Prefer v-if if the condition is unlikely to change at runtime.

Class binding

In HTML:

```
<div id="app">
  <div :class="{ active: isActive }"></div>
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
   isActive: true
 }
})
```

`isActive can be an expression that evaluates boolean value (true/false)`

Class binding with regular class attribute

In HTML:

```
<div id="app">
   <div class="static"
  :class="{ active: isActive,
'text-danger': hasError }">
  </div>
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
    isActive: true,
    hasError: false
 }
})
```

Will render:

```
<div class="static active"></div>
```

Class binding as object

In HTML:

```
<div id="app">
  <div class="static"
:class="classObject"></div>
</div>
```

Will also render:

```
<div class="static active"></div>
```

In JS:

```
var app = new Vue({
 el: '#app',
data: {
    classObject: {
      active: true,
      'text-danger': false
    }
 }
})
```

Class binding as array of objects

In HTML:

```
<div id="app">
      <div :class="[{ active: isActive },
      errorClass]"></div>
</div>
```

In JS:

```
var app = new Vue({
  el: '#app',
  data: {
    isActive: true,
    activeClass: 'active',
    errorClass: 'text-danger'
  }
})
```

Will render:

```
<div class="active text-danger"></div>
```

Inline style binding

In HTML:

```
<div id="app">
  <div :style="{ color: activeColor,
fontSize: fontSize + 'px' }"></div>
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
     activeColor: 'red',
     fontSize: 30
 }
})
```

Will render:

```
<div style="color: 'red'; font-size: 30px"></div>
```

Inline style binding using object

In HTML:

```
<div id="app">
   <div :style="styleObject"></div>
</div>
```

Will also render:

```
<div style="color: 'red'; font-size: 30px"></div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
     styleObject: {
       color: 'red',
       fontSize: '13px'
     }
 }
})
```

Loops with v-for

In HTML:

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

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
   todos: [
     { text: 'Learn JavaScript' },
     { text: 'Learn Vue' },
     { text: 'Build something awesome' }
   ]
 }
})
```

Open Console and type:

```
app.todos.push({ text: 'New item' })
```

With index:

```
<li v-for="(todo, index) in todos">
  index: {{ index }} {{ todo.text }}
</li>
```

v-for with an Object

In HTML:

```
<div id="app">
  <ul id="v-for-object" class="demo">
    <li v-for="value in object">
      {{ value }}
    </li>
  </ul>
</div>
```

Output:

```
• John
• Doe
• 30
```

Or with the prop key:

```
<div v-for="(value, key) in object">
  {{ key }}: {{ value }}
</div>
```

Output:

```
firstName: John
lastName: Doe
age: 30
```

In JS:

```
var app = new Vue({
    el: '#app',
    data: {
      object: {
        firstName: 'John',
        lastName: 'Doe',
        age: 30
      }
    }
    })
```

v-for key

In HTML:

```
<div id="app">
 <ol>
   <li v-for="todo in todos"  :key="todo.id">
     {{ todo.text }}
   </li>
 </ol>
</div>
```

You need to provide a unique **key** attribute for each item to give Vue a hint to track each node's identity, and thus reuse and reorder existing elements.

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
   todos: [
     { id: 1, text: 'Learn JavaScript' },
     { id: 2, text: 'Learn Vue' },
     { id: 3, text: 'Build something awesome' }
   ]
 }
})
```

v-for with a range

In HTML:

```
<div id="app">
  <span v-for="n in 10">{{ n }}</span>
</div>
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

Note: "n" can be replaced with any variable

v-for with v-if and on <template>

In HTML: (Only renders the todos that are not complete.)

```
<div id="app">
      <li v-for="todo in todos" v-if="!todo.isComplete">
       {{ todo }}
      </li>
</div>
```

Loop items without adding a wrapper element:

```
<div id="app">
       <ul>
        <template v-for="todo in todos">
          <li>{{ todo.text }}</li>
          <li class="divider"></li>
        </template>
       </ul>
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
   todos: [
     { id: 1, text: 'Learn JavaScript', isCompleted: true},
     { id: 2, text: 'Learn Vue', isCompleted: false },
     { id: 3, text: 'Build something', isCompleted: true }
   ]
 }
})
```

Events with v-on that calls methods

In HTML:

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

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
   message: 'Hello Vue.js!'
 },
 methods: {
   reverseMessage() {
     this.message = this.message.split('').reverse().join('')
   }
 }
})
```

"v-on:" keyword can be replaced with "@"

so this is the same code:

```
<div id="app">
 <p>{{ message }}</p>
 <button @click="reverseMessage">Reverse Message</button>
</div>
```

Note: instead of writing **reverseMessage: function() { }**

We can use ES6 Function Syntax: **reverseMessage(){ }**

Methods with parameter

In HTML:

```
<div id="app">
  <div>
   <button @click="say('hi')">Say hi</button>
   <button @click="say('what')">Say what</button>
  </div>
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 methods: {
   say(message) {
     alert(message)
   }
 }
})
```

Events Modifiers

In HTML:

```html
<!-- the click event's propagation will be stopped
-->
<a v-on:click.stop="doThis"></a>

<!-- the submit event will no longer reload the
page -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- modifiers can be chained -->
<a v-on:click.stop.prevent="doThat"></a>


<!-- just the modifier -->
<form v-on:submit.prevent></form>
```

- `.stop`
- `.prevent`
- `.capture`
- `.self`
- `.once`
- `.passive`

```html
<!-- use capture mode when adding the event listener -->
<!-- i.e. an event targeting an inner element is handled
here before being handled by that element -->
<div v-on:click.capture="doThis">...</div>

<!-- only trigger handler if event.target is the element
itself -->
<!-- i.e. not from a child element -->
<div v-on:click.self="doThat">...</div>
```

Key Modifiers

In HTML:

```html
<!-- only call `vm.submit()` when the `keyCode` is 13 -->
<input v-on:keyup.13="submit">

<!-- same as above -->
<input v-on:keyup.enter="submit">

<!-- also works for shorthand -->
<input @keyup.enter="submit">
```

- `.enter`
- `.tab`
- `.delete` (captures both “Delete” and “Backspace” keys)
- `.esc`
- `.space`
- `.up`
- `.down`
- `.left`
- `.right`

System Key Modifiers

In HTML:

```
<!-- Alt + C -->
<input @keyup.alt.67="clear">

<!-- Ctrl + Click -->
<div @click.ctrl="doSomething">Do something</div>
```

- `.ctrl`
- `.alt`
- `.shift`
- `.meta`

JSFIDDLE - Chained Key Modifiers

Two way data-binding with v-model

In HTML:

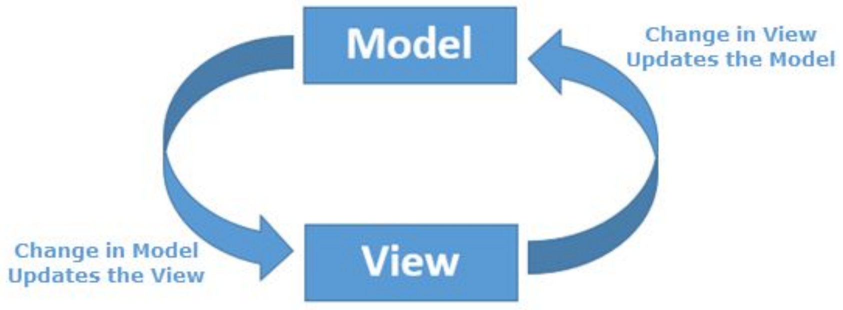
```
<div id="app">
 <p>{{ message }}</p>
 <input v-model="message">
</div>
```

In JS:

```
var app = new Vue({
 el: '#app',
 data: {
   message: 'Hello Vue!'
 }
})
```

Every change in the message property on model will change the view.

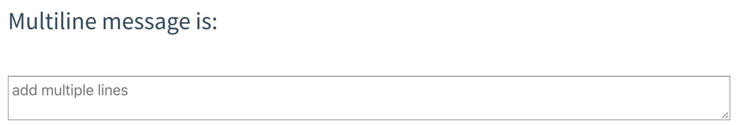
And Every change in the input on view will change the model.


Model
Change in View Updates the Model
Change in Model Updates the View
View

Other form elements with v-model

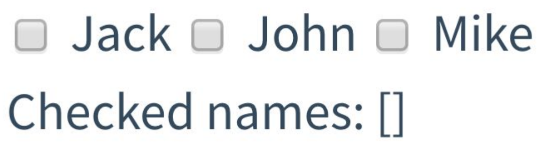
In HTML:

```
<span>Multiline message is:</span>
<p style="white-space: pre-line;">{{ message }}</p><br/>
<textarea v-model="message" placeholder="add multiple lines"></textarea>
```


Multiline message is:

add multiple lines

```
<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
<label for="jack">Jack</label>
<input type="checkbox" id="john" value="John" v-model="checkedNames">
<label for="john">John</label>
<input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
<label for="mike">Mike</label><br/>
<span>Checked names: {{ checkedNames }}</span>
</div>
```


☐ Jack ☐ John ☐ Mike

Checked names: []

JSFIDDLE

Dynamic options with v-model

In HTML:

```
<select v-model="selected">
 <option v-for="option in options" v-bind:value="option.value">
   {{ option.text }}
 </option>
</select>
<span>Selected: {{ selected }}</span>
```

In JS:

```
var app = new Vue({
     el: '...',
     data: {
       selected: 'A',
       options: [
         { text: 'One', value: 'A' },
         { text: 'Two', value: 'B' },
         { text: 'Three', value: 'C' }
       ]
     }
    })
```

JSFIDDLE

v-model Modifiers

In HTML:

```html
<!-- synced after "change" instead of "input" -->
<input v-model.lazy="msg">

<!-- auto typecast user input as a number -->
<input v-model.number="age" type="number">

<!-- auto trim user input -->
<input v-model.trim="msg">
```

- `.lazy`
- `.number`
- `.trim`

Computed Properties

In HTML:

```
<div id="app">
  <p>Original message: "{{ message }}"</p>
  <p>Computed reversed message: "{{ reversedMessage }}"</p>
</div>
```

Output:

```
Original message: "Hello"
Computed reversed message: "olleH"
```

In JS:

```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello'
  },
  computed: {
    // a computed getter
    reversedMessage() {
      return this.message.split('').reverse().join('')
    }
  }
})
```

Computed Properties Vs. Methods

By Computed:

```
var app = new Vue({
 el: '#app,
 data: {
   message: 'Hello'
 },
 computed: {
   reversedMessage() {
     return this.message.split('').reverse().join('')
   }
 }
})
```

By Methods:

```
var app = new Vue({
 el: '#app,
 data: {
   message: 'Hello'
 },
 methods: {
     reverseMessage() {
       return this.message.split('').reverse().join('')
     }
    }
})
```

<u>The Difference</u>: Computed properties are cached based on their dependencies.

A computed property will only re-evaluate when some of its dependencies have changed.

JSFIDDLE

Class binding as object with computed

In HTML:

```
<div id="app">
  <div class="static"
:class="classObject"></div>
</div>
```

Will also render:

```
<div class="static active"></div>
```

In JS:

```
var app = new Vue({
  el: '#app',
  data: {
   isActive: true,
   error: null
  },
  computed: {
   classObject() {
     return {
       active: this.isActive && !this.error,
       'text-danger': this.error && this.error.type === 'fatal'
     }
   }
  }
})
```

Watchers Properties

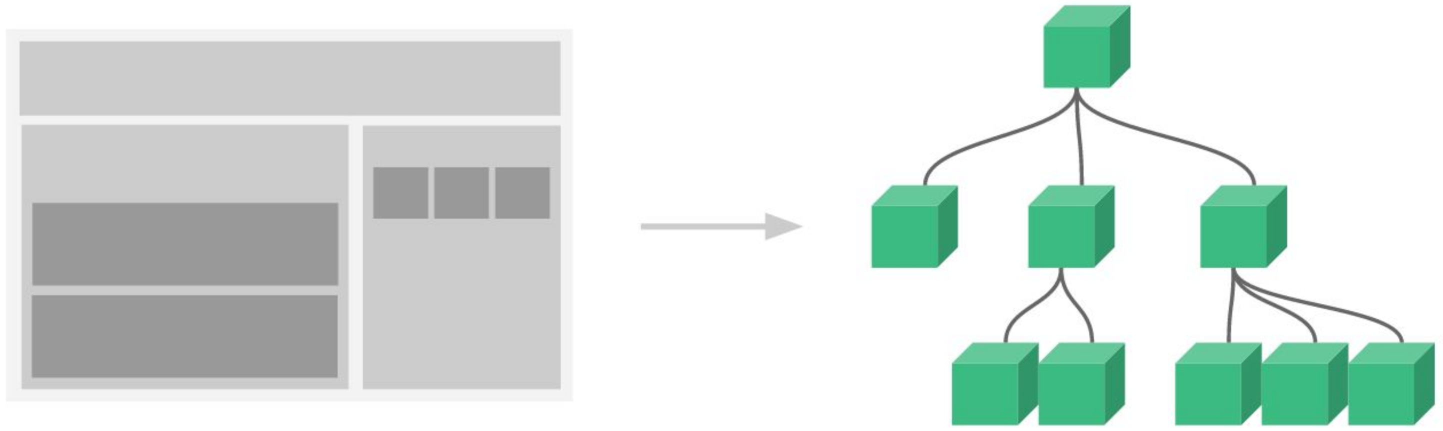
In HTML:

```
<div id="app">
    <div><input type="text" v-model="firstName"></div>
    <div><input type="text" v-model="lastName"></div>
   <div>{{ fullName }}</div>
</div>
```

In JS:

```
var app = new Vue({
 el: '#app,
 data: {
   firstName: 'Foo',
   lastName: 'Bar',
   fullName: 'Foo Bar'
 },
 watch: {
   firstName(val, oldVal) {
     this.fullName = val + ' ' + this.lastName
   },
   lastName(val, oldVal) {
     this.fullName = this.firstName + ' ' + val
   }
 }
})
```

Components



Components - Example

In HTML:

```
<div id="components-demo">
  <button-counter></button-counter>
</div>
```

In JS:

```
Vue.component('button-counter', {
  data: function () {
    return {
      count: 0
    }
  },
  template: '<button @click="count++">You clicked me {{ count }} times.</button>'
})

var app = new Vue({
 el: '#components-demo,
})
```

Called **Global Component Registration.**
Must be used before Vue instance, This even applies to all subcomponents, meaning all three of these components will also be available inside each other.

Reusing Components

In HTML:

```
<div id="components-demo">
  <button-counter></button-counter>
  <button-counter></button-counter>
  <button-counter></button-counter>
</div>
```

In JS:

```
Vue.component('button-counter', {
  data: function () {
    return {
      count: 0
    }
  },
  template: '<button @click="count++">You clicked me {{
count }} times.</button>'
})

var app = new Vue({
 el: '#components-demo,
})
```

Called **Global Component Registration.**
Must be used before Vue instance, This even applies to all subcomponents, meaning all three of these components will also be available inside each other.

Data for component must be function

When we define a component, don't directly pass `data` object.
Instead of writing this:

```
data: {
  count: 0
}
```

Because component’s `data` option must be a function, each instance can maintain an independent copy of the returned data object:

```
data: function () {
  return {
    count: 0
  }
}
```

If Vue didn’t have this rule, clicking on one button would affect the data of *all other instances*

Global data example:

```
<div id="app"></div>
<script>
  const data = {
    count: 0
  }
  Vue.component('counter',{
    data() {return data},
    methods: {
      increment() {
        this.count++;
      }
    },
    template: `<div>Count: {{count}}
<button @click="increment">Increment</button></div>`
  });
  var app = new Vue({
    el: '#app',
    template:
`<div><counter></counter><counter></counter></div>`
  });
</script>
```

Every component must have only one root element

If you try this in your **template**, Vue will show an error, explaining that every component must have a single root element.
You can fix this error by wrapping the template in a parent element.

WRONG!

```
<h3>{{ title }}</h3>
<div v-html="content"></div>
```

RIGHT!

```
<div class="blog-post">
  <h3>{{ title }}</h3>
  <div v-html="content"></div>
</div>
```

Tip: Check vue-fragments plugin!

Passing Data to Child Components with Props

In HTML:

```html
<div id="app">
  <blog-post title="My journey with Vue"></blog-post>
  <blog-post title="Blogging with Vue"></blog-post>
  <blog-post title="Why Vue is so fun"></blog-post>
</div>
```

In JS:

```javascript
Vue.component('blog-post', {
  props: ['title'],
  template: '<h3>{{ title }}</h3>'
})

var app = new Vue({
 el: '#app',
 data: {
   message: 'Hello Vue!'
 }
})
```

Passing Data to Child Components with Props in loop

In HTML:

```
<div id="app">
      <blog-post
        v-for="post in posts"
        v-bind:key="post.id"
        v-bind:title="post.title"></blog-post>
</div>
```

In JS:

```
Vue.component('blog-post', {
  props: ['title'],
  template: '<h3>{{ title }}</h3>'
})

var app = new Vue({
  el: '#blog-post-demo',
  data: {
    posts: [
      { id: 1, title: 'My journey with Vue' },
      { id: 2, title: 'Blogging with Vue' },
      { id: 3, title: 'Why Vue is so fun' }
    ]
  }
})
```

Props - One way data flow - From parent to child only!

1. The prop is used to pass in an **initial value**; the child component wants to **use it as a local data property afterwards.**
   In this case, it’s best to define a local data property that uses the prop as its initial value:

```
props: ['initialCounter'],
data: function () {
  return {
    counter: this.initialCounter
  }
}
```

2. The prop is passed in as a raw value **that needs to be transformed.**
   In this case, it’s best to define a computed property using the prop’s value.

```
props: ['size'],
computed: {
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```

Props Validations

```
Vue.component('my-component', {
  props: {
    // Basic type check (`null` and `undefined` values will pass any type validation)
    propA: Number,
    // Multiple possible types
    propB: [String, Number],
    // Required string
    propC: {
      type: String,
      required: true
    },
    // Number with a default value
    propD: {
      type: Number,
      default: 100
    },
    // Object with a default value
    propE: {
      type: Object,
      // Object or array defaults must be returned from
      // a factory function
      default: function () {
        return { message: 'hello' }
      }
    },
    // Custom validator function
    propF: {
      validator: function (value) {
        // The value must match one of these strings
        return ['success', 'warning', 'danger'].indexOf(value) !== -1
      }
    }
  }
})
```

Passing props to childs and use them - Demo

https://codepen.io/aspittel/pen/oVMBmO
By Ali Spittel from Vue Vixens

Tip: try to use computed instead of a method!

Listening to child components events

As we develop our component, some features may require communicating **back up to the parent**.

1. Use built-in $emit function on the component, like this:

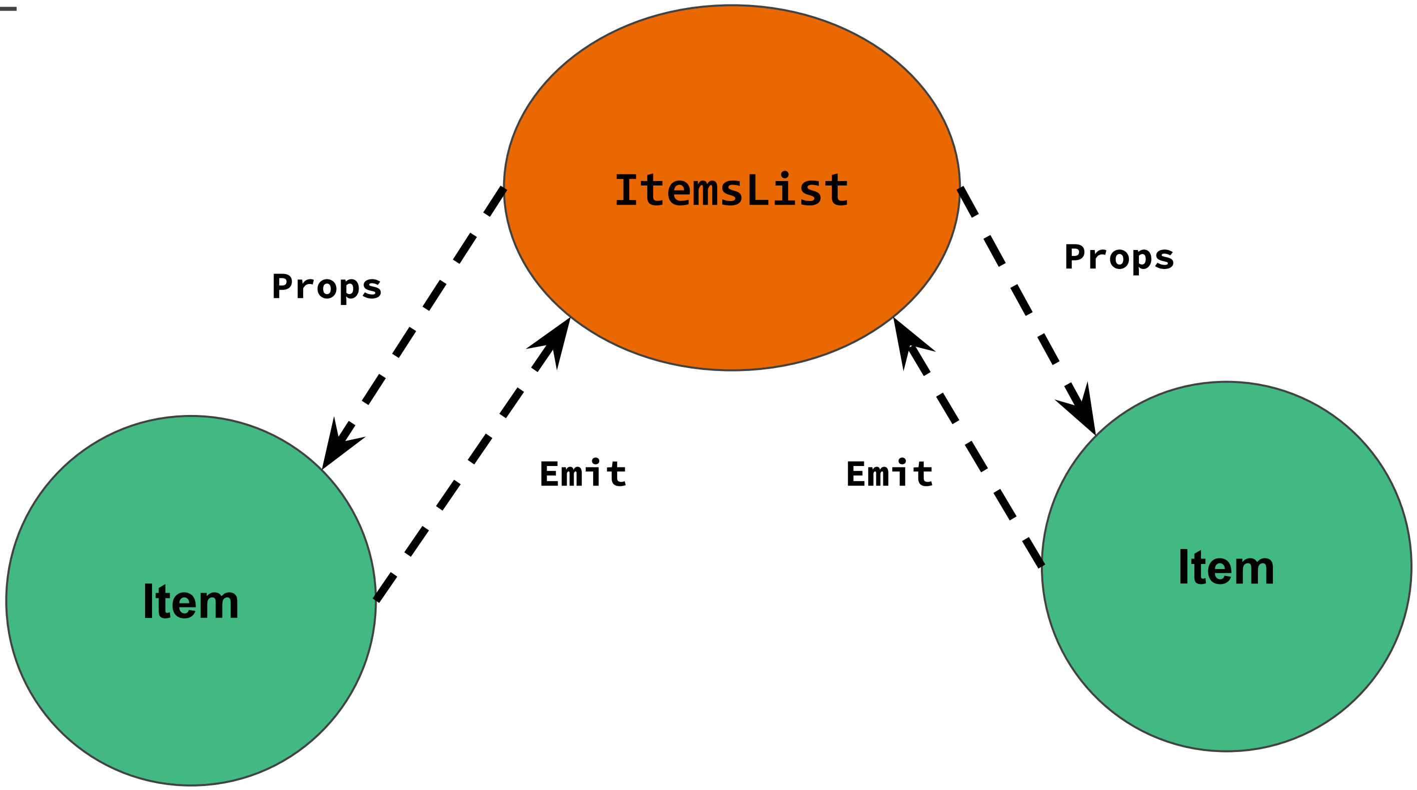
```
<button @click="$emit('item-clicked')">Push me!</button>
```

2. Listen and catch the event on the parent, with event listener:

```
<item-comp @item-clicked="doSomething()" />
```

Tip: you can also pass data using `$emit('item-clicked', data)`

Parent to child - Props
Child to parent - Emit events



ItemsList & Item Example

Cart with badge:

https://gist.github.com/eladcandroid/d5af5f781c46a6fdcffc249dd955277f

Cart with badge and cart items (Emit with value):

https://gist.github.com/eladcandroid/b2a7d919ac7cb24fa077780c8306acc0

Using v-model on components

```
<custom-input v-model="searchText"></custom-input>
```

```
<input v-model="searchText">
```

is the same as

```
<input
  v-bind:value="searchText"
  v-on:input="searchText = $event.target.value">
```

That's why

```
<custom-input v-model="searchText">
```

is the same as

```
<custom-input
  v-bind:value="searchText"
  v-on:input="searchText = $event"
></custom-input>
```

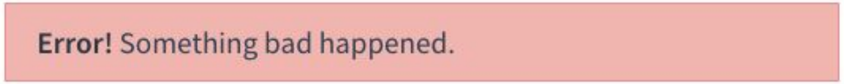
For this to actually work though, the <input> inside the component must:

- **Bind** the value attribute to a **value prop**
- On input, **emit** its own **custom input** event with the new value

```
Vue.component('custom-input', {
  props: ['value'],
  template: `
    <input
      v-bind:value="value"
      v-on:input="$emit('input', $event.target.value)"
    >
  `
})
```

Content Distribution with **Slots**

```
<alert-box>
  Something bad happened.
</alert-box>
```


Error! Something bad happened.

```
Vue.component('alert-box', {
  template: `
    <div class="demo-alert-box">
      <strong>Error!</strong>
      <slot></slot>
    </div>
  `
})
```

Dynamic components

```
<!-- Component changes when currentTabComponent changes -->
<component v-bind:is="currentTabComponent"></component>
```

Note that Component and "is" attribute are special keywords

In the example above, currentTabComponent can contain either:

the **name** of a **registered component**, or a **component's options object**

Tabs example:

https://jsfiddle.net/chrisvfritz/o3nycadu/

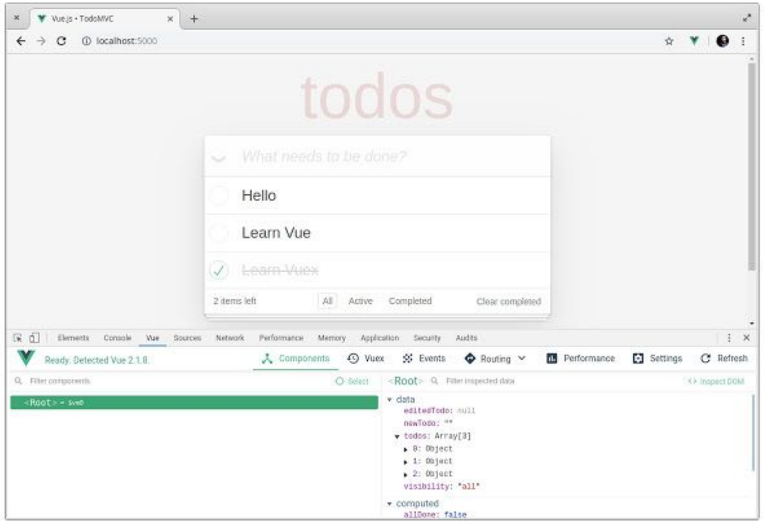
Dynamic components views array:

https://jsfiddle.net/eladcandroid/2on3gu51/

Vue DevTools

We can debug our app with Vue Dev Tools chrome extension which is a great friend to see our data, computed (etc..) in action and change our values on the fly.
Including a support for vue-router.

https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd?hl=en



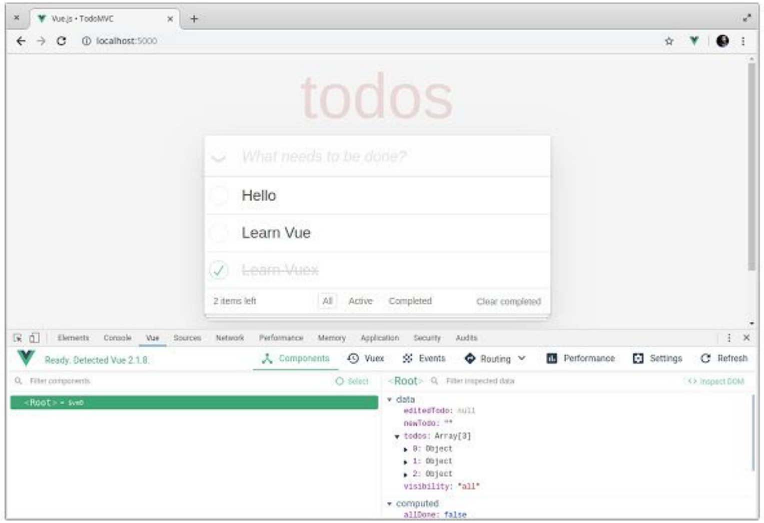
```
▾ data
    editedTodo: null
    newTodo: ""
  ▾ todos: Array[3]
    ▸ 0: Object
    ▸ 1: Object
    ▸ 2: Object
    visibility: "all"
▾ computed
    allDone: false
```

Vue.JS CLI

To work with a development server for arranging our code, importing libraries, haing linting support and getting ready for a production – we use Vue CLI system.

Install from here:

```
npm install -g @vue/cli
```



```
▾ data
    editedTodo: null
    newTodo: ""
  ▾ todos: Array[3]
    ▸ 0: Object
    ▸ 1: Object
    ▸ 2: Object
    visibility: "all"
▾ computed
    allDone: false
```

Vue.JS CLI Getting Started

Create a new project hello-world using CLI:

```
vue create hello-world
```

Create a new project hello-world using GUI:

```
vue ui
```