WEB PROGRAMMING 06016322

BY DR BUNDIT THANASOPON

INTRO TO VUE.JS

SRC: HTTPS://VUEJS.ORG/V2/GUIDE/INDEX.HTML#WHAT-IS-VUE-JS

WHAT IS VUE.JS?

- Vue (pronounced /vjuː/, like **view**) is a **progressive framework** for building user interfaces.
- The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects.
- **Free videos:** https://www.vuemastery.com/courses/intro-to-vue-js/vue-instance/

LET'S GET STARTED

- Include this in your index.html
  - `<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>`
- Vue Devtools
  - https://github.com/vuejs/vue-devtools#vue-devtools

It is that EASY!!!

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:

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

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

The data and the DOM are now linked, and everything is now **reactive**.

Open your browser’s JavaScript console (right now, on this page) and set app.message to a different value. You should see the rendered example above update accordingly.

V-BIND DIRECTIVE

- We can also bind element attributes like this:

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

- Directives are prefixed with v- to indicate that they are special attributes provided by Vue.
- They apply special reactive behavior to the rendered DOM.

CONDITIONALS AND LOOPS

```
<span v-if="seen">Now you see me</span>
<span v-else>Now you do not see me</span>
```

- Whether the <span> tag are displayed or not is based on the value of “seen” variable.
- This example demonstrates that we can bind data to not only text and attributes, but also the **structure** of the DOM.

CONDITIONALS AND LOOPS

- The v-for directive can be used for displaying a list of items using the data from an Array:

```
<div id="app-4">
  <ol>
    <li v-for="todo in todos">{{ todo.text }}</li>
  </ol>
</div>
```

```
var app4 = new Vue({
        el: '#app-4’,
        data: {
                todos: [
                        { text: 'Learn JavaScript' },
                        { text: 'Learn Vue' },
                        { text: 'Build something awesome’ }
                        ]
                }
        })
```

HANDLING USER 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 id="app-5">
  <p>{{ message }}</p>
  <button v-on:click="reverseMessage">Reverse Message</button>
</div>
```

```
var app5 = new Vue({
        el: '#app-5',
        data: { message: 'Hello Vue.js!' },
        methods: {
                        reverseMessage: function () {
                        this.message =
this.message.split('').reverse().join('')
                }
        }
})
```

HANDLING USER INPUT

- Vue also provides the v-model directive that makes two-way binding between form input and app state a breeze:

```
<div id="app-6">
  <p>{{ message }}</p>
  <input v-model="message">
</div>
```

```
var app6 = new Vue({
        el: '#app-6',
        data: {
            message: 'Hello Vue!'
        }
    })
```

THE VUE INSTANCE

CREATING A VUE INSTANCE

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

```
var vm = new Vue({
        // options
})
```

- When you create a Vue instance, you pass in an ”**options object**”.

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.
- 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.
- If you know you'll need a property later, but it starts out empty or non-existent, you'll need to set some initial value. For example:

```
data: {
  newTodoText: '',
  visitCount: 0,
  hideCompletedTodos:
false,
  todos: [],
  error: null
}
```

TEMPLATE SYNTAX

INTERPOLATIONS

- Text
  - The most basic form of data binding is text interpolation using the “Mustache” syntax (double curly braces):

```
<span>Message: {{ msg }}</span>
```

  - You can also perform one-time interpolations that do not update on data change by using the **v-once directive**

```
<span v-once>This will never change: {{ msg }}</span>
```

- Raw HTML

```
<span v-html="rawHtml"></span>
rawHtml = “<span style=“color: red;”>Hello</span>”
```

INTERPOLATIONS

- Attributes
  - Mustaches cannot be used inside HTML attributes. Instead, use a **v-bind directive**:

```
<div v-bind:id="dynamicId"></div>
<button v-bind:disabled="isButtonDisabled">Button</button>
```

- Using JavaScript Expressions
  - Vue.js actually supports the full power of JavaScript expressions inside all data bindings:

```
{{ number + 1 }}
{{ message.split('').reverse().join('') }}
<div v-bind:id="'list-' + id"></div>
```

SHORTHANDS

- **v-bind Shorthand**

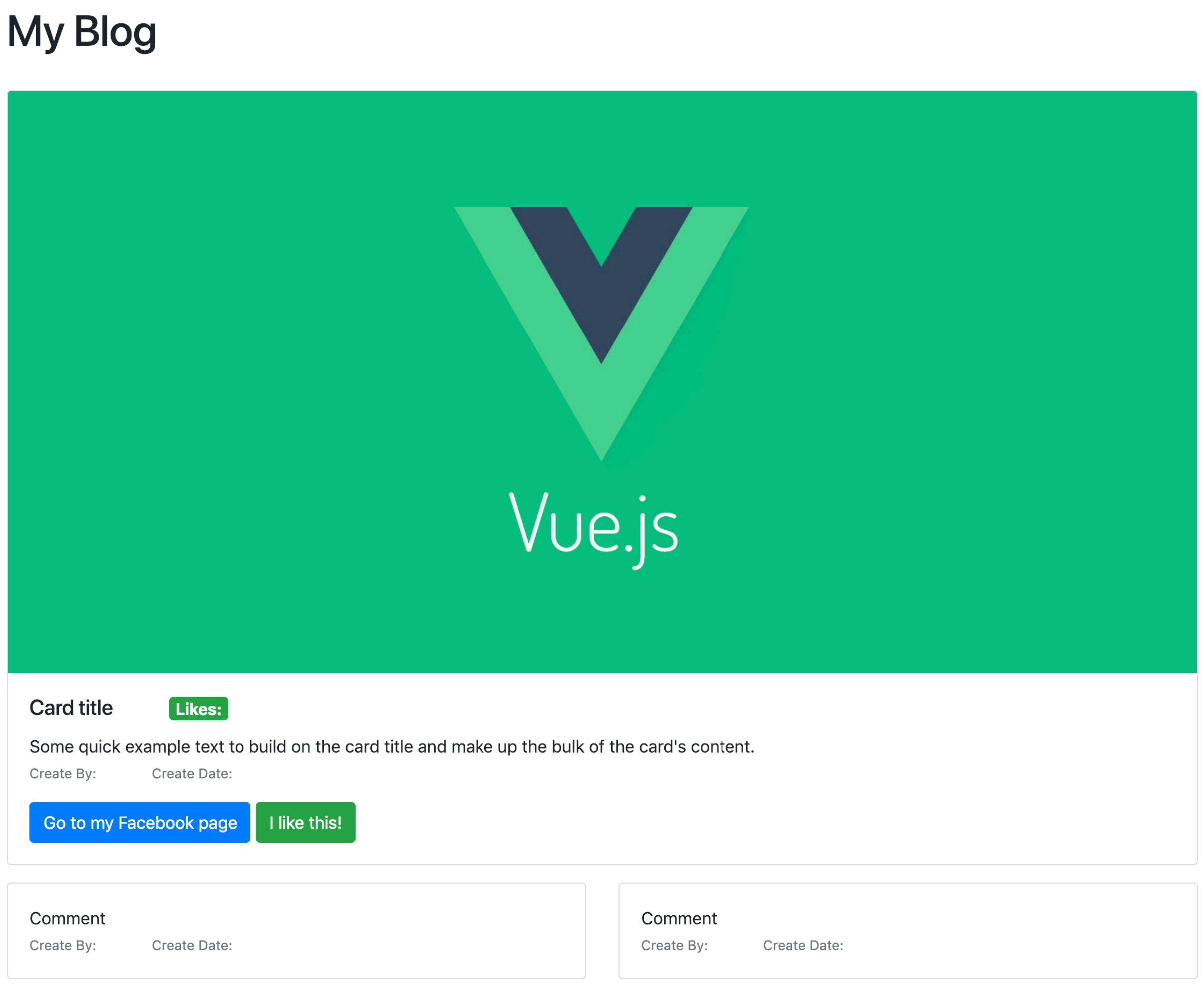
```
<!-- full syntax -->
<a v-bind:href="url"> ... </a>

<!-- shorthand -->
<a :href="url"> ... </a>
```

- **v-on Shorthand**

```
<!-- full syntax -->
<a v-on:click="doSomething"> ... </a>

<!-- shorthand -->
<a @click="doSomething"> ... </a>
```


My Blog

Card title Likes:

Some quick example text to build on the card title and make up the bulk of the card's content.

Create By: Create Date:

Comment
Create By: Create Date:

Comment
Create By: Create Date:

LET'S CREATE A BLOG PAGE

Create a blog page with Vue

Open file blog.html and modify the code

CONDITIONAL RENDERING

V-IF

```
<h1 v-if="ok">Yes</h1>
```

```
<h1 v-if="ok">Yes</h1>
<h1 v-else>No</h1>
```

A v-else element must immediately follow a v-if or a v-else-if element - otherwise it will not be recognized.

- Conditional Groups with v-if on <template>

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

V-SHOW

- Another option for conditionally displaying an element is the v-show directive. The usage is largely the same:

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

- The difference is that an element with v-show will always be rendered and remain in the DOM; v-show only toggles the display CSS property of the element.

LIST RENDERING

MAPPING AN ARRAY TO ELEMENTS WITH V-FOR

- We can use the v-for directive to render a list of items based on an array.
- The v-for directive requires a special syntax in the form of item in items, where items is the source data array and item is an **alias** for the array element being iterated on:

```
<ul id="example-1">
        <li v-for=”(item, index) in items">
        {{ item.message }}
        </li>
</ul>
```

```
var example1 = new Vue({
    el: '#example-1’,
    data: {
        items: [
                { message: 'Foo' },
                { message: 'Bar’ }
            ]
        }
})
```

V-FOR WITH AN OBJECT

- You can also use v-for to iterate through the properties of an object.

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

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

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

```
new Vue({
        el: '#v-for-object’,
        data: {
                object: {
                        firstName: 'John’,
                        lastName: 'Doe’,
                        age: 30
                }
        }
    })
```

ARRAY CHANGE DETECTION

- Mutation Methods - Vue wraps an observed array’s mutation methods so they will also trigger view updates. The wrapped methods are:
  - push()
  - pop()
  - shift()
  - unshift()
  - splice()
  - sort()
  - reverse()

LIMITATIONS - ARRAY

- Due to limitations in JavaScript, Vue **cannot** detect the following changes to an array:
  - – When you directly set an item with the index,
    - e.g. vm.items[indexOfItem] = newValue
  - – When you modify the length of the array,
    - e.g. vm.items.length = newLength
- To deal with limitation 1:
  - – vm.items.splice(indexOfItem, 1, newValue)
- To deal with limitation 2:
  - – vm.items.splice(newLength)

LIMITATIONS - OBJECT

- Due to limitations of modern JavaScript, **Vue cannot detect property addition or deletion**. Vue does not allow dynamically adding new root-level reactive properties to an already created instance. For example:

```
var vm = new Vue({
                  data: {
                           a: 1
                  }
         })
// `vm.a` is now reactive

vm.b = 2
// `vm.b` is NOT reactive
```

DISPLAYING FILTERED/SORTED RESULTS

- Sometimes we want to display a filtered or sorted version of an array **without actually mutating or resetting the original data**. In this case, you can create a computed property that returns the filtered or sorted array.

```
<li v-for="n in evenNumbers">{{ n }}</li>
```

```
data: {
        numbers: [ 1, 2, 3, 4, 5 ]
},
computed: {
        evenNumbers: function () {
                return this.numbers.filter(function
(number) {
                        return number % 2 === 0
                })
        }
}
```

V-FOR WITH A RANGE

- v-for can also take an integer. In this case it will repeat the template that many times.

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

Results: 1 2 3 4 5 6 7 8 9 10

V-FOR WITH V-IF

- When they exist on the same node, v-for has a higher priority than v-if. That means the v-if will be run on each iteration of the loop separately.

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

- The above only renders the todos that are not complete.

EVENT HANDLING

JAVASCRIPT EVENTS

- **HTML events** are **"things"** that happen to HTML elements.
- When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

- An HTML event can be something the browser does, or something a user does.
- Here are some examples of HTML events:
  - An HTML web page has finished loading
  - An HTML input field was changed
  - An HTML button was clicked
- Often, when events happen, you may want to do something. JavaScript lets you execute code when events are detected.

JAVASCRIPT EVENTS

```
<button onclick="document.getElementById('demo').innerHTML =
Date()">The time is?</button>

<button onclick="this.innerHTML = Date()">The time is?</button>

<button onclick="displayDate()">The time is?</button>
```

COMMON HTML EVENTS

Event	**Description**
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

EXAMPLES

```
<button>Click me</button>
<p>No handler here.</p>
<script>
   let button = document.querySelector("button");
   button.addEventListener("click", () =>
{ console.log("Button clicked."); });
</script>
```

```
<button>Act-once button</button>
<script>
      let button =
      document.querySelector("button");
      function once() { console.log("Done.");
        button.removeEventListener("click", once); }
      button.addEventListener("click", once);
</script>
```

PROPAGATION

- For most event types, handlers registered on nodes with children will also receive events that happen in the children.
- If a button inside a paragraph is clicked, event handlers on the paragraph will also see the click event.

- At any point, an event handler can call the **stopPropagation** method on the event object to prevent handlers further up from receiving the event

```
<p>A paragraph with a <button>button</button>.</
p>
<script>
  let para = document.querySelector("p");
  let button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Handler for paragraph.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Handler for button.");
    if (event.button == 2) event.stopPropagation();
  });
</script>
```

DEFAULT ACTIONS

- Many events have a default action associated with them.
  - If you click a link, you will be taken to the link’s target.
  - If you press the down arrow, the browser will scroll the page down.
  - If you right-click, you’ll get a context menu.
- You can use **preventDefault** to prevent the default action from happening.

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("Nope."); event.preventDefault();
  });
</script>
```

LISTENING TO EVENTS IN VUE

- We can use the **v-on** directive to listen to DOM events and run some JavaScript when they're triggered.

```
<div id="example-1">
  <button v-on:click="counter += 1">Add
1</button>
  <p>The button above has been clicked
{{ counter }} times.</p>
</div>
```

```
var example1 = new Vue({
                    el: '#example-1',
                    data: { counter: 0 }
        })
```

METHOD EVENT HANDLERS

```html
<div id="example-2">
<!-- `greet` is the name of a method defined below -->
    <button v-on:click="greet">Greet</button>
</div>
```

```javascript
var example2 = new Vue({
  el: '#example-2’,
  data: { name: 'Vue.js' },
  methods: {
    greet: function (event) {
      alert('Hello ' + this.name + '!’)
      if (event) { alert(event.target.tagName) }
    }
  }
})
```

EVENT MODIFIERS

- It is a very common need to call **event.preventDefault()** or **event.stopPropagation()** inside event handlers.
- Vue provides **event modifiers** for v-on. Recall that modifiers are directive postfixes denoted by a dot.

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

COMPUTED PROPERTIES AND WATCHERS

COMPUTED PROPERTIES

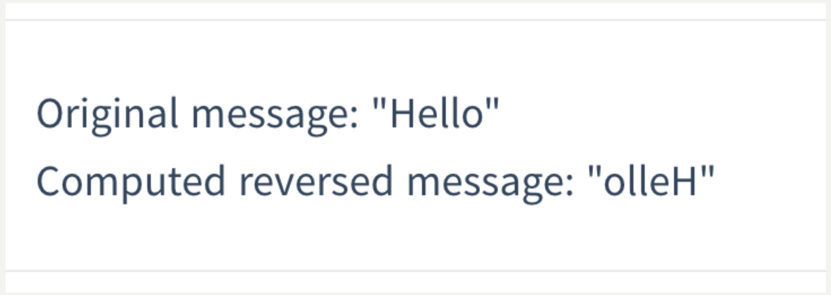
- In-template expressions are very convenient, but they are meant for simple operations.
- Putting too much logic in your templates can make them bloated and hard to maintain.

```
<div id="example">
  {{ message.split('').reverse().join('') }}
</div>
```

- The problem is made worse when you want to include the reversed message in your template more than once.
- That’s why for any complex logic, you should use a **computed property**.

BASIC EXAMPLES

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



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

```
var vm = new Vue({
              el: '#example’,
              data: {
                message: 'Hello’
              },
              computed: {
                // a computed getter
                reversedMessage: function () {
                  // `this` points to the vm instance
                  return this.message.split('').reverse().join(”)
                }
              }
})
```

COMPUTED CACHING VS METHODS

- You may have noticed we can achieve the same result by invoking a method in the expression.
- For the end result, the two approaches are indeed exactly the same. However, the difference is that **computed properties are cached based on their dependencies.**
- A computed property will only re-evaluate when some of its dependencies have changed.

```
methods: {
    reverseMessage: function () {
        return this.message.split('').reverse().join('')
    }
}
```

COMPUTED SETTER

- Computed properties are by default getter-only, but you can also provide a setter when you need it:

```
computed: {
        fullName: {
                // getter
                get: function () {
                  return this.firstName + ' ' + this.lastName
               },
                // setter
                set: function (newValue) {
                  var names = newValue.split(' ‘)
                  this.firstName = names[0]
                  this.lastName = names[names.length - 1]
               }
        }
}
```

Now when you run vm.fullName = 'John Doe', the setter will be invoked

and vm.firstName and vm.lastName will be updated accordingly.

WATCHERS

- Watchers are most useful when you want to perform asynchronous or expensive operations in response to changing data.

```
watch: {
		// whenever question changes, this function will run
		question: function (newQuestion, oldQuestion) {
			this.answer = 'Waiting for you to stop typing...’
			this.debouncedGetAnswer()
		}
},
```

COMPUTED VS WATCHED PROPERTY

- When you have some data that needs to change based on some other data, it is tempting to overuse watch.
- However, it is often a better idea to use a computed property rather than an imperative watch callback.
- Try to do with computed!

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

CLASS AND STYLE BINDINGS

BINDING HTML CLASSES - OBJECT SYNTAX

- We can pass an object to v-bind:class to dynamically toggle classes:

```
<div v-bind:class="{ active: isActive }"></div>
```

- You can have multiple classes toggled by having more fields in the object.
- In addition, the v-bind:class directive can also co-exist with the plain class attribute.

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

- The bound object doesn’t have to be inline:
- <div v-bind:class="classObject"></div>

```
data: {
	classObject: {
		active: true,
		'text-danger': false
	}
}
```

ARRAY SYNTAX

- We can pass an array to v-bind:class to apply a list of classes:

```
<div v-bind:class="[activeClass, errorClass]"></div>
```

```
data: {
        activeClass: 'active’,
        errorClass: 'text-danger'
}
```

- Which will render:

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

- It’s also possible to use the object syntax inside array syntax:

```
<div v-bind:class="[{ active: isActive }, errorClass]"></div>
```

BINDING INLINE STYLES - OBJECT SYNTAX & ARRAY SYNTAX

- The object syntax for v-bind:style is pretty straightforward - it looks almost like CSS, except it’s a JavaScript object.
- You can use either **camelCase** or **kebab-case** (use quotes with kebab-case) for the CSS property names:

```
<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
data: {
        activeColor: 'red’,
        fontSize: 30
}
```

- The array syntax for v-bind:style allows you to apply multiple style objects to the same element:

```
<div v-bind:style="[baseStyles, overridingStyles]"></div>
```