WEB PROGRAMMING 0601632 2

BY DR BUNDIT THANASOPON

FORM INPUT BINDINGS

V-MODEL MAGIC

You can use the v-model directive to create **two-way data bindings** on form input, textarea, and select elements. It automatically picks the correct way to update the element based on the input type.

```
<input v-model="message" placeholder="edit me">
<p>Message is: {{ message }}</p>
```

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

v-model will ignore the initial **value**, **checked**, or **selected** attributes found on any form elements.

V-MODEL EXAMPLE

- <input type=”text” />
- <textarea>
- <select>
- <checkbox>
- <radio>
- <input type=“date” />
- <input type=“time” />
- <input type=“color” />

VUE 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>
```

HANDLE EVENT IN VUEJS

```
// inline
<button onclick="this.innerHTML = Date()">The time is?</button>
<button v-on:click=”now = Date()">The time is {{ now }}</button>
<button @click=”now = Date()">The time is {{ now }}</button>

// method
<button onclick="displayDate()">The time is?</button>
<button v-on:click="displayDate()">The time is?</button>
<button @click="displayDate()">The time is?</button>
```

Handling event in Vue is very similar to vanilla JavaScript, just change `onclick` to `@click`

HANDLE EVENT WITH METHODS

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

Normally handling event is more complex than one-line JavaScript, for that we can use "methods"

```
var example2 = new Vue({
    el: '#app',
    data: {
        now: ''
    },
    methods: {
        displayDate() {
            // this is a function
            // write any js you want!
            this.now = Date()
        }
    }
})
```

METHODS ARGUMENTS

```
<button @click=”say(‘hi’)">Hi</button>
<button @click=”say(‘hello’)">Hello</button>
```

You can also call methods with arguments

```
var example2 = new Vue({
    el: ‘#app’,
    methods: {
        say(msg) {
            alert(msg)
        }
    }
})
```

THE $EVENT OBJECT

```
<button @click=”say($event)">Hi</button>
<button @click=”say($event)">Hello</button>
```

You can also access DOM Event by using the $event special variable

```
var app = new Vue({
    el: ‘#app’,
    methods: {
        say(e) {
            alert(e.target.innerHTML)
        }
    }
})
```

EVENT 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>
```

EVENT MODIFIER

To make handling event easier. Vuejs provided Event Modifier to do common task like stopPropagation() or preventDefault()

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

```
<!-- 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>
```

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: "{{ msg }}"</p>
  <p>Computed reversed message: "{{ reversedMessage }}"</p>
</div>
```

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

Original message: "Hello"

Computed reversed message: "olleH"

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('')
        }
}
```

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
            }
        }
})
```

VUE INSTANCE LIFE CYCLE

LIFE CYCLE HOOK

- Each Vue instance goes through a series of initialization steps when it's created
- it needs to set up data observation
- compile the template
- mount the instance to the DOM
- 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.

```
var vm = new Vue({
    el: '#example',
    data: {
        msg: 'Hello'
    },
    created () {
        // code to run when vue is created
    },
    mounted () {
        // code to run when vue is mounted
    }
})
```

LIFE CYCLE DIAGRAM

- https://vuejs.org/v2/guide/instance.html#Lifecycle-Diagram