**Computed properties and watchers**

Computed property is a property which integrates complex logic that is cached based on its dependencies such that they can modify, manipulate and display their dependencies. Whenever these reactive dependencies or reactive values change, this property will be recomputed. Otherwise, the cached value will be returned.

The purpose of having computed property is to eradicate the issues that arise from bloated code that are difficult to read and to be maintained. Consider the following example.

<div>{{ firstName + ‘ ‘ + lastName + ‘, Age: ‘ + age }}</div>

Even with a short example here, it can be seen how we have inserted too many logical operations into the moustache brackets. Although, in this case, the code is still fairly readable, this will be hard to maintain as the code grows longer, resulting in multiple lines of code that are enclosed within the brackets. We can, instead, move these lines of code outside of the brackets into the computed properties.

Computed: {

printData: function() {

return this.firstName + ‘ ‘ + lastName + ‘, Age: ‘ + age;

}

}

Such that we can simply enclose this property inside the moustache brackets.

<div>{{ printData }}</div>

Watchers on the other hand, is a property which allows asynchronous and expensive operations to take place when there is a change in data. Unlike computed properties, watchers only call functions and is capable of only watching one props at a time.

In the following example, every time there is a change to count data, for example, when a data binding took place that change 0 to other values, the watch property which watches this data will be notified, and will execute the function.

data() {

return {

count: 0;

}

},

watch: {

count() {

console.log(‘count has been changed’)

}

}

**Class and Style Bindings**

A concrete implementation of data binding by manipulating the class list of an element and its inline styles.

Class binding is a concept of using data binding to set a class property for an element. For instance, in the following example, with add a new class ‘responsive’ if the screen size is said to be narrower than the threshold (Boolean value: true).

<div v-bind:class = ‘{ responsive: isNarrow }’></div>

It is also possible to bind an array of classes with v-bind.

<div v-bind:class = “[firstClass, secondClass]”></div>

data: {

firstClass = “responsive”,

secondClass = “interactive”

}

When this code is rendered, it will produce a div with two classes; responsive and interactive.

Style binding is a concept of using data binding to add or remove one or several styles to a DOM element. In the following example, we bind the value of chosenColor onto the color property in CSS. This is powerful, as, chosenColor is in the data property, thus is dynamic and can be modified from the user’s interface with the help of v-model.

<div v-bind:style=”{color: chosenColor}”></div>

data: {

chosenColor: ‘white’

}

**Conditional Rendering**

Conditional rendering is a directive which toggles the presence of an element when the directive’s expression returns a true value. In other words, is has the ability to control and decide, whether or not a template of code should be rendered. Keywords used in conditional rendering are v-if, v-else-if, v-else (similar to if else statements )and v-show.

Using the example from section 3.2, different paragraphs would be rendered depending on the condition that it fulfils

<p v-if = “guess < number>Number is bigger</p>

<p v-else-if = “guess > number>Number is smaller</p>

<p v-else>You got that right!</p>

v-show on the other hand, will always be rendered and stayed in the DOM. It is much simpler as it will only display the element if the condition is true, otherwise, it will not do anything.

<p v-show=”guess==number”>You got that right!</p>

**List Rendering**

List rendering is a directive which renders items in an array. The key modifier used is v-for.

<p v-for=”data in datas”>

{{data}}

</p>

In this example, datas is an array, and we use the alias ‘data’ to represent all the items in datas. All individual object inside this array will be rendered as data is passed into the moustache brackets.

**Event handling**

Event handling is a directive which captures DOM events such as user input, user clicks, etc. and runs JavaScript when the events that they are listening to are triggered. The key modifier used is v-on.

<button v-on:click=”count+=1”>Click Me</button>

data: {

count: 0

}

In this example, the v-on directive listens to the button DOM and will increment count variable when the button is clicked (event detected).

**Form input bindings**

This is a directive which is capable of monitoring user input and updates the data once a change is detected. The key modifier used is v-model.

This directive applies different properties and perform different events for different types of input. On input event takes place when the element gets user is changed from the user’s interface, while on change occurs when the state of an element changes (when it loses focus).

|  |  |  |
| --- | --- | --- |
| Input Type | Property | Event |
| text | value | input |
| textarea | value | input |
| checkbox | checked | change |
| radiobutton | checked | change |
| select | value | change |

**Component Basics**

Components are Vue instances that are reusable, they accept same properties as Vue instances such as data, methods, computed, watch.

In the following example, the data property contains the data specific to the component, we can then use a template to render this data whenever this component is used.

//creates new component named reminder

Vue.component(‘reminder, {

data: function() {

return: {

event: ‘Coding Bootcamp’,

date: 13-08-2021

}

},

template: <div><h1>{{event}}</h1><p>{{date}}</p></div>

})

//uses the component in HTML file

<reminder></reminder>

**Component Registration**

The first argument is registering a component is Vue.component.

Vue.component(‘component-name, {..})

Two options in defining a component names; kebab-case and PascalCase. The above code is an example of registering a Vue component globally, whereas it can also be registered locally.

var ComponentName = {..}

We can also register a parent-child component communication by using a module system.

import ComponentName from ‘./ComponentName’

export default {

components: { ComponentName },

**Props**

Props is a special keyword in Vue used to pass data between different components, especially when a parent component needs to pass data to its children components; when a value is passed onto prop, it is a property of that component. Props can be seen as passing variables into function as arguments in JavaScript.

In the following example, whenever FirstComponent is used, we need to pass a SomeWords as a props, before it can be rendered properly.

Vue.component(‘FirstComponent, {

Props: [‘SomeWords’],

Template: ‘<p>{{SomeWords}}</p>’ })

**Custom Events**

Custom events is an ability in Vue which listens to specified event that take place in elements, and can fire off events when this event occur. It has the opposite functionality of props, which allows a communication from child component to perform events which parent components listen for. The key modifier used is $emit.

Unlike components and props, event names is automatically transformed to lowercase, thus it is recommended to use kebab case for event names.

**Slots**

Slots are booked space which is used to display contents that were passed from one component to another. Unlike props which are vulnerable to scripting attack, slots allow a more reliable and secure method for passing html elements.

To use a slot, inside the template of child component, we insert slot html tag <slot></slot>. When parent component render this component, any content that is provided between the opening and closing tag of this component will replace the slot tag.

**Dynamic and Async Components**

Dynamic components allow us to dynamically switch between components. It has, but not bounded to, the following use cases.

Navigating tabbed interface without routing to a new page

Managing multiple types of popups in a single component element

Displaying multiple content depending on whether a user is logged in or not.

Two things that need to be included to use dynamic components:

* Component element
* V-bind:is attribute

Async components divide an application to smaller bite pieces and only load them when needed, also known as lazy loading. The component is defined as factory function component and will be triggered when the component needs to be rendered, the results will also be cached for future re-renders.

**Handling Edge Cases**

Handling edge cases allow us to bend Vue’s rules in unusual situations. For example, forcing an update even though no reactive data has changed. Several features of handling edge cases include:

* Accessing root instance
* Accessing parent component instance
* Accessing child component instances
* Dependency injection
* Recursive components
* Circular references between components