Watchers

A watcher is a special Vue.js feature that allows you to spy on one property of the component state, and run a function when that property value changes.

Here’s an example. We have a component that shows a name, and allows you to change it by clicking a button:

```
<template>

 <p>My name is {{name}}</p>

 <button @click="changeName()">Change my name!</button>

</template>
<script>
export default {
 data() {
   return {
     name: 'Flavio'
   }
 },
 methods: {
   changeName: function() {
     this.name = 'Flavius'
   }
 }
}
</script>
```

When the name changes we want to do something, like print a console log.

We can do so by adding to the watch object a property named as the data property we want to watch over:

```
<script>
export default {
 data() {
   return {
     name: 'Flavio'
   }
 },
 watch: {
   name: function() {
     console.log(this.name)
   }
 }
}
</script>
```

The function assigned to watch.name can optionally accept 2 parameters. The first is the new property value. The second is the old property value:

```
<script>

export default {

 /* ... */

 watch: {

   name: function(newValue, oldValue) {

     console.log(newValue, oldValue)

   }

 }

}

</script>
```

Computed Properties

In Vue.js you can output any data value using parentheses:

```
<template>

 <p>{{ count }}</p>

</template>
<script>
export default {
 data() {
   return {
     count: 1
   }
 }
}
</script>
```

This property can host some small computations. For example:

```
<template>

 {{ count * 10 }}

</template>
```

But you’re limited to a single JavaScript **expression**.

In addition to this technical limitation, you also need to consider that templates should only be concerned with displaying data to the user, not perform logic computations.

To do something more than a single expression, and to have more declarative templates, you use a computed property.

Computed properties are defined in the computed property of the Vue component:

```
<script>

export default {

 computed: {
 }
}
</script>
```

Here’s an example that uses a computed property count to calculate the output.

```
<template><p>{{ count }}</p></template>
<script>
export default {
 data() {
   return {
     items: [1, 2, 3]
   }
 },
 computed: {
   count: function() {
     return 'The count is ' + this.items.length * 10
   }
 }
}
</script>
```

I didn’t have to call {{ count() }}. Vue.js automatically invokes the function

Methods vs. Watchers vs. Computed Properties

Now that you know about methods, watchers and computed properties, you might be wondering when should you use one vs the others.

Here's a breakdown of this question.

When to use methods

- To react to some event happening in the DOM
- To call a function when something happens in your component.
  You can call a method from computed properties or watchers.

When to use computed properties

- You need to compose new data from existing data sources
- You have a variable you use in your template that's built from one or more data properties
- You want to reduce a complicated, nested property name to a more readable and easy to use one (but update it when the original property changes)
- You need to reference a value from the template. In this case, creating a computed property is the best thing, because it's cached.
- You need to listen to changes of more than one data property

When to use watchers

- You want to listen when a data property changes, and perform some action
- You want to listen to a prop value change
- You only need to listen to one specific property (you can't watch multiple properties at the same time)
- You want to watch a data property until it reaches some specific value and then do something

Vue - CLI (Setting up Vue - CLI)

What is the Vue CLI? It's a command line utility that helps to speed up development by scaffolding an application skeleton for you, with a sample app in place.

The first is to install the Vue CLI on your computer and run the command:

- **vue create <enter the app name>**

The files structure

Beside package.json, which contains the configuration, these are the files contained in the initial project structure:

- Index.html
- src/App.vue
- src/main.js
- src/assets/logo.png
- src/components/HelloWorld.vue

index.html

The index.html file is the main app file.

In the body it includes just one simple element: <div id="app"></div>. This is the element the Vue application we'll use to attach to the DOM.

src/main.js

This is the main JavaScript file that drives our app.

We first import the Vue library and the App component from App.vue.

Next, we create the Vue instance, by assigning it to the DOM element identified by #app, which we defined in index.html, and we tell it to use the App component.

src/App.vue

App.vue is a Single File Component. It contains three chunks of code: HTML, CSS, and JavaScript.

Vue Components

Components are single, independent units of an interface. They can have their own state, markup, and style.

How to use components:
Vue components can be defined in **four main ways**. Let’s talk in code.

The first is:

```
new Vue({

 /* options */

})
```

The second is:

```
Vue.component('component-name', {

 /* options */

})
```

The third is by using local components. These are components that are only accessible by a specific component, and not available elsewhere (great for encapsulation).

The fourth is in .vue files, also called Single File Components.

Let’s dive into the first 3 ways in detail.

Using new Vue() or Vue.component() is the standard way to use Vue when you're building an application that is not a Single Page Application (SPA). You use this method, rather, when you’re just using Vue.js in some pages, like in a contact form or in the shopping cart. Or maybe Vue is used in all pages, but the server is rendering the layout, and you serve the HTML to the client, which then loads the Vue application you build.

In an SPA, where it’s Vue that builds the HTML, it’s more common to use Single File Components as they are more convenient.

> You instantiate Vue by mounting it on a DOM element. If you have a
>
> <div id="app"></div> tag, you will use:

```
new Vue({ el: '#app' })
```

A component initialized with new Vue has no corresponding tag name, so it's usually the main container component.

Other components used in the application are initialized using Vue.component(). Such a component allows you to define a tag — with which you can embed the component multiple times in the application — and specify the output of the component in the template property:

```
<div id="app">

 <sample-component></sample-component>

</div>
Vue.component('sample-component', {
 template: '<p>This is a Sample Component</p>'
})
new Vue({
 el: '#app'
})
```

In the Vue.component() call we passed sample-component as the first parameter. This gives the component a name. You can write the name in 2 ways here. The first is the one we used, called kebab-case. The second is called PascalCase, which is like camelCase, but with the first letter capitalized:

```
Vue.component('SampleComponent', {

 /* ... */

})
```

Vue automatically creates an alias internally from sample-component to SampleComponent , and vice versa, so you can use whatever you like. It's generally best to use SampleComponent in the JavaScript, and sample-component in the template.

Local components

Any component created using Vue.component() is globally registered. You don't need to assign it to a variable or pass it around to reuse it in your templates.

You can encapsulate components locally by assigning an object that defines the component object to a variable:

```
const Sidebar = {

 template: '<aside>Sidebar</aside>'

}
```

and then make it available inside another component by using the components property:

```
new Vue({

 el: '#app',

 components: {

   Sidebar

 }

})
```

You can write the component in the same file, but a great way to do this is to use JavaScript modules:

```
import Sidebar from './Sidebar'
export default {
 el: '#app',
 components: {
   Sidebar
 }
}
```

The building blocks of a component:

So far we’ve seen how a component can accept the el and template properties.

- el is only used in root components initialized using new Vue({}), and identifies the DOM element the component will mount on.
- template is where we can set up the component template, which will be responsible for defining the output the component generates.

A component accepts other properties:

- props lists all the properties that we can pass down to a child component
- data the component local state
- methods: the component methods
- computed: the computed properties associated with the component
- watch: the component watchers

Single File Components

A Vue component can be declared in a JavaScript file (.js) like this:

```
Vue.component('component-name', {

 /* options */

})
```

or also:

```
new Vue({

 /* options */

})
```

or it can be specified in a .vue file.

The .vue file is pretty cool because it allows you to define:

- JavaScript logic
- HTML code template
- CSS styling

all in just a single file. As such it got the name of Single File Component.

Here’s an example:

```
<template>

 <p>{{ hello }}</p>

</template>
<script>
export default {
 data() {
   return {
     hello: 'Hello World!'
   }
 }
}
</script>
<style scoped>
 p {
   color: blue;
 }
</style>
```

All of this is possible thanks to the use of Webpack. The Vue CLI makes this very easy and supported out of the box. .vue files cannot be used without a Webpack setup. Since Single File Components rely on Webpack, we get for free the ability to use modern Web features.

Reusable Components

A child component can be added multiple times. Each separate instance is independent of the others:

```
<div id="app">

 <sample-component></sample-component>

<sample-component></sample-component>

<sample-component></sample-component>

<sample-component></sample-component>

<sample-component></sample-component>

</div>
Vue.component('sample-component', {
 template: '<p>This is a Sample Component</p>'
})
new Vue({
 el: '#app'
})
```

Props

Props: pass data to child components.
When a component expects one or more prop, it must define them in its props property:

```
Vue.component('user-name', {

 props: ['name'],

 template: '<p>Hi {{ name }}</p>'

})
```

or, in a Vue Single File Component:

```
<template>

 <p>{{ name }}</p>

</template>
<script>
export default {
 props: ['name']
}
</script>
```

Accept multiple props

You can have multiple props by simply appending them to the array:

```
Vue.component('user-name', {
 props: ['firstName', 'lastName'],
 template: '<p>Hi {{ firstName }} {{ lastName }}</p>'
})
```

Set the prop type

You can specify the type of a prop very simply by using an object instead of an array, using the name of the property as the key of each property, and the type as the value:

```
Vue.component('user-name', {
 props: {
   firstName: String,
   lastName: String
 },
 template: '<p>Hi {{ firstName }} {{ lastName }}</p>'
})
```

The valid types you can use are:

- String
- Number
- Boolean
- Array
- Object
- Date
- Function
- Symbol

When a type mismatches, Vue alerts you (in development mode) in the console with a warning.

Prop types can be more articulated.

You can allow multiple different value types:

```
props: {

 firstName: [String, Number]

}
```

Set a prop to be mandatory

You can require a prop to be mandatory:

```
props: {
 firstName: {

   type: String,

   required: true

 }

}
```

Set the default value of a prop

You can specify a default value:

```
props: {

 firstName: {

   type: String,

   default: 'Unknown person'

 }

}
```

For objects:

```
props: {

 name: {

   type: Object,

   default: {

     firstName: 'Unknown',

     lastName: ''

   }

 }

}
```

Passing props to the component

You pass a prop to a component using the syntax

```
<ComponentName color="white" />
```

if what you pass is a static value.

If it’s a data property, you use

```
<template>

 <ComponentName :color=color />

</template>
<script>
...
export default {
 //...
 data: function() {
   return {
     color: 'white'
   }
 },
 //...
}
</script>
```

You can use a ternary operator inside the prop value to check a truthy condition and pass a value that depends on it:

```
<template><ComponentName :colored="color == 'white' ? true : false"
/></template>

<script>
...
export default {
 //...
 data: function() {
   return {
     color: 'white'
   }
 },
 //...
}
</script>
```

Custom Directives

Vue also allows you to register your own custom directives. Note that in Vue 2.0, the primary form of code reuse and abstraction is components - however there may be cases where you need some low-level DOM access on plain elements, and this is where custom directives would still be useful. An example would be focusing on an input element, like this one:

```
// Register a global custom directive called `v-focus`
Vue.directive('focus', {
 // When the bound element is inserted into the DOM...
 inserted: function (el) {
  // Display Alert When Focus the element
  alert("Focused!!")
 }
})
```

If you want to register a directive locally instead, components also accept a directives option:

```
directives: {
  focus: {
    // directive definition
    //inserted - is a Hook Function
    inserted: function (el) {
      alert("Focused!!")
    }
  }
}
```

Then in a template, you can use the new v-focus attribute on any element, like this:

```
<input v-focus>
```

More Information - https://vuejs.org/v2/guide/custom-directive.html

Slots

A component can choose to define its content entirely, like in this case:

```
Vue.component('user-name', {

        props: ['name'],

         template: '<p>Hi {{ name }}</p>'

})
```

Or it can also let the parent component inject any kind of content into it, by using slots.

By putting `<slot></slot>` in a component template:

```
Vue.component('user-information', {

 template: '<div class="user-information"><slot></slot></div>'

})
```

When using this component, any content added between the opening and closing tag will be added inside the slot placeholder:

```
<user-information>

 <h2>Hi!</h2><user-name name="Flavio">

</user-information>
```

If you put any content side the <slot></slot> tags, that serves as the default content in case nothing is passed in.

A complicated component layout might require a better way to organize content.

Enter **Named Slots**.

With a named slot, you can assign parts of a slot to a specific position in your component template layout, and you use a slot attribute to any tag, to assign content to that slot.

Anything outside any template tag is added to the main slot.

For convenience, I use a page single file component in this example:

```
<template>

 <div>

   <main><slot></slot></main>

   <sidebar><slot name="sidebar"></slot></sidebar>

 </div>

</template>
<page>
 <ul slot="sidebar">
   <li>Home</li>
   <li>Contact</li>
 </ul>
 <h2>Page title</h2>
 <p>Page content</p>
</page>
```