FREE eBook

LEARNING Vue.js

Free unaffiliated eBook created from
Stack Overflow contributors.

#vue.js

www.dbooks.org

Table of Contents

www.dbooks.org

www.dbooks.org

www.dbooks.org

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: vue-js

It is an unofficial and free Vue.js ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Vue.js.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

www.dbooks.org

Chapter 1: Getting started with Vue.js

Remarks

`Vue.js` is a rapidly growing front-end framework for `JavaScript`, inspired by `Angular.js`, `Reactive.js`, and `Rivets.js` that offers simplistic user-interface design, manipulation, and deep reactivity.

It is described as a `MVVM` patterned framework, `Model-View View-Model`, which is based on the concept of *two-way binding* data to components and views. It is incredibly fast, exceeding speeds of other top-tier `JS` frameworks, and very user friendly for easy integration and prototyping.

Versions

Version	Release Date
2.4.1	2017-07-13
2.3.4	2017-06-08
2.3.3	2017-05-09
2.2.6	2017-03-26
2.0.0	2016-10-02
1.0.26	2016-06-28
1.0.0	2015-10-26
0.12.0	2015-06-12
0.11.0	2014-11-06

Examples

"Hello, World!" Program

To start using Vue.js, make sure you have the script file included in your HTML. For example, add the following to your HTML.

```
<script src="https://npmcdn.com/vue/dist/vue.js"></script>
```

Simple Example

HTML template

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

JavaScript

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

See a live demo of this example.

You might also want to check out the "Hello World" example made by Vue.js.

Hello World in Vue 2 (The JSX way)

JSX is not meant to be interpreted by the browser. It must be first transpiled into standard Javascript. To use JSX you need to install the plugin for babel `babel-plugin-transform-vue-JSX`

Run the Command below:

```
npm install babel-plugin-syntax-jsx babel-plugin-transform-vue-jsx babel-helper-vue-jsx-merge-
props --save-dev
```

and add it to your `.babelrc` like this:

```
{
  "presets": ["es2015"],
  "plugins": ["transform-vue-jsx"]
}
```

Sample code with VUE JSX:

```
import Vue from 'vue'
import App from './App.vue'

new Vue({
  el: '#app',
  methods: {
    handleClick () {
      alert('Hello!')
    }
  },
  render (h) {
    return (
```


www.dbooks.org

```
      <div>
        <h1 on-click={this.handleClick}>Hello from JSX</h1>
        <p> Hello World </p>
      </div>
    )
  }
})
```

By using JSX you can write concise HTML/XML-like structures in the same file as you write JavaScript code.

Congratulations, You're Done :)

Handling User Input

VueJS can be used to easily handle user input as well, and the two way binding using v-model makes it really easy to change data easily.

HTML :

```
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<div id="app">
    {{message}}
<input v-model="message">
</div>
```

JS :

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

It is very easy to do a two-way binding in VueJS using `v-model` directive.

Check out a live example here.

Read Getting started with Vue.js online: https://riptutorial.com/vue-js/topic/1057/getting-started-with-vue-js

Chapter 2: Components

Remarks

In Component(s):

props is an array of string literals or object references used to pass data from parent component. It can also be in object form when it is desired to have more fine grained control like specifying default values, type of data accepted, whether it is required or optional

data has to be a function which returns an object instead of a plain object. It is so because we require each instance of the component to have its own data for re-usability purpose.

events is an object containing listeners for events to which the component can respond by behavioral change

methods object containing functions defining the behavior associated with the component

computed properties are just like watchers or observables, whenever any dependency changes the properties are recalculated automatically and changes are reflected in DOM instantly if DOM uses any computed properties

ready is a Vue instance's life-cycle hook

Examples

Component scoped (not global)

Demo

HTML

```
<script type="x-template" id="form-template">
    <label>{{inputLabel}} :</label>
  <input type="text" v-model="name" />
</script>

<div id="app">
  <h2>{{appName}}</h2>
  <form-component title="This is a form" v-bind:name="userName"></form-component>
</div>
```

JS

```
// Describe the form component
// Note: props is an array of attribute your component can take in entry.
// Note: With props you can pass static data('title') or dynamic data('userName').
```


www.dbooks.org

```
// Note: When modifying 'name' property, you won't modify the parent variable, it is only
descendent.
// Note: On a component, 'data' has to be a function that returns the data.
var formComponent = {
  template: '#form-template',
  props: ['title', 'name'],
  data: function() {
    return {
      inputLabel: 'Name'
    }
  }
};

// This vue has a private component, it is only available on its scope.
// Note: It would work the same if the vue was a component.
// Note: You can build a tree of components, but you have to start the root with a 'new
Vue()'.
var vue = new Vue({
  el: '#app',
  data: {
    appName: 'Component Demo',
    userName: 'John Doe'
  },
  components: {
    'form-component': formComponent
  }
});
```

What are components and how to define components?

Components in Vue are like widgets. They allow us to write reusable custom elements with desired behavior.

They are nothing but objects which can contain any/all of the options that the root or any Vue instance can contain, including an HTML template to render.

Components consist of:

- HTML markup: the component's template
- CSS styles: how the HTML markup will be displayed
- JavaScript code: the data and behavior

These can each be written in a separate file, or as a single file with the `.vue` extension. Below are examples showing both ways:

.VUE - as a single file for the component

```
<style>
    .hello-world-compoment{
        color:#eeeeee;
        background-color:#555555;
    }
</style>

<template>
    <div class="hello-world-component">
```

```
        <p>{{message}}</p>
        <input @keyup.enter="changeName($event)"/>
    </div>
</template>

<script>
    export default{
        props:[ /* to pass any data from the parent here... */ ],
        events:{ /* event listeners go here */},
        ready(){
            this.name= "John";
        },
        data(){
           return{
              name:''
           }
        },
        computed:{
            message(){
                return "Hello from " + this.name;
            }
        },
        methods:{
            // this could be easily achieved by using v-model on the <input> field, but just
to show a method doing it this way.
            changeName(e){
                this.name = e.target.value;
            }
        }
    }
</script>
```

Separate Files

hello-world.js - the JS file for the component object

```
export default{
    template:require('./hello-world.template.html'),
    props:[ /* to pass any data from the parent here... */ ],
    events:{ /* event listeners go here */ },
    ready(){
        this.name="John";
    },
    data(){
        return{
            name:''
        }
    },
    computed:{
        message(){
            return "Hello World! from " + this.name;
        }
    },
    methods:{
        changeName(e){
            let name = e.target.value;
            this.name = name;
        }
    }
}
```


www.dbooks.org

hello-world.template.html

```
<div class="hello-world-component">
    <p>{{message}}</p>
    <input class="form-control input-sm" @keyup.enter="changeName($event)">
</div>
```

hello-world.css

```
 .hello-world-compoment{
    color:#eeeeee;
    background-color:#555555;
}
```

These examples use es2015 syntax, so Babel will be needed to compile them to es5 for older browsers.
Babel along with **Browserify + vueify** or **Webpack + vue-loader** will be required to compile `hello-world.vue`.

Now that we have the `hello-world` component defined, we should register it with Vue.

This can be done in two ways:

Register as a global component
In the `main.js` file (entry point to the app) we can register any component globally with `Vue.component`:

```
import Vue from 'vue'; // Note that 'vue' in this case is a Node module installed with 'npm
install Vue'
Vue.component('hello-world', require('./hello-world');  // global registeration

new Vue({
    el:'body',

    // Templates can be defined as inline strings, like so:
    template:'<div class="app-container"><hello-world></hello-world></div>'
});
```

Or register it locally within a parent component or root component

```
import Vue from 'vue'; // Note that 'vue' in this case is a Node module installed with 'npm
install Vue'
import HelloWorld from './hello-world.js';

new Vue({
    el:'body',
    template:'<div class="app-container"><hello-world></hello-world></div>",

    components:{HelloWorld}  // local registeration
});
```

Global Components can be used anywhere within the Vue application.

Local Components are only available for use in the parent component with which they are registered.

Fragment component
You may get a console error telling you that you can't do something because yours is a *fragment component*. To solve this sort of issue just wrap your component template inside a single tag, like a `<div>`.

Local registration of components

A component can be registered either globally or locally (bind to another specific component).

```
var Child = Vue.extend({
    // ...
})

var Parent = Vue.extend({
    template: '...',
    components: {
         'my-component': Child
    }
})
```

Thiw new component () will only be available inside the scope (template) of the Parent component.

Inline registration

You can extend and register a component in one step:

```
Vue.component('custom-component', {
    template: '<div>A custom component!</div>'
})
```

Also when the component is registered locally:

```
var Parent = Vue.extend({
    components: {
        'custom-component': {
            template: '<div>A custom component!</div>'
        }
    }
})
```

Data registration in components

Passing an object to the `data` property when registering a component would cause all instances of the component to point to the same data. To solve this, we need to return `data` from a function.

```
var CustomComponent = Vue.extend({
    data: function () {
        return { a: 1 }
```


www.dbooks.org

```
    }
})
```

Events

One of the ways components can communicate with its ancestors/descendants is via custom communication events. All Vue instances are also emitters and implement a custom event interface that facilitates communication within a component tree. We can use the following:

- `$on`: Listen to events emitted by this components ancestors or descendants.
- `$broadcast`: Emits an event that propagates downwards to all descendants.
- `$dispatch`: Emits an event that triggers first on the component itself and than propagates upwards to all ancestors.
- `$emit`: Triggers an event on self.

For example, we want to hide a specific button component inside a form component when the form submits. On the parent element:

```
var FormComponent = Vue.extend({
  // ...
  components: {
    ButtonComponent
  },
  methods: {
    onSubmit () {
        this.$broadcast('submit-form')
    }
  }
})
```

On the child element:

```
var FormComponent = Vue.extend({
  // ...
  events: {
    'submit-form': function () {
        console.log('I should be hiding');
    }
  }
})
```

Some things to keep in mind:

- Whenever an event finds a component that is listening to it and gets triggered, it will stop propagating unless the function callback in this component returns `true`.
- `$dispatch()` always triggers first on the component that has emitted it.
- We can pass any number of arguments to the events handler. Doing `this.$broadcast('submit-form', this.formData, this.formStatus)` allows us to access this arguments like `'submit-form': function (formData, formStatus) {}`

Read Components online: https://riptutorial.com/vue-js/topic/1775/components

Chapter 3: Computed Properties

Remarks

Data vs Computed Properties

The main use-case difference for the `data` and `computed` properties of a `Vue` instance is dependent on the potential state or probability of changing of the data. When deciding what category a certain object should be, these questions might help:

- Is this a constant value? (**data**)
- Does this have the possibility to change? (**computed** or **data**)
- Is the value of it reliant on the value of other data? (**computed**)
- Does it need additional data or calculations to be complete before being used? (**computed**)
- Will the value only change under certain circumstances? (**data**)

Examples

Basic Example

Template

```
<div id="example">
  a={{ a }}, b={{ b }}
</div>
```

JavaScript

```
var vm = new Vue({
  el: '#example',
  data: {
    a: 1
  },
  computed: {
    // a computed getter
    b: function () {
      // `this` points to the vm instance
      return this.a + 1
    }
  }
})
```

Result

```
a=1, b=2
```

Here we have declared a computed property `b`. The function we provided will be used as the getter

www.dbooks.org

function for the property `vm.b`:

```
console.log(vm.b) // -> 2
vm.a = 2
console.log(vm.b) // -> 3
```

The value of `vm.b` is always dependent on the value of `vm.a`.

You can data-bind to computed properties in templates just like a normal property. Vue is aware that `vm.b` depends on `vm.a`, so it will update any bindings that depends on `vm.b` when `vm.a` changes.

Computed properties vs watch

template

```
<div id="demo">{{fullName}}</div>
```

watch example

```
var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
    lastName: 'Bar',
    fullName: 'Foo Bar'
  }
})

vm.$watch('firstName', function (val) {
  this.fullName = val + ' ' + this.lastName
})

vm.$watch('lastName', function (val) {
  this.fullName = this.firstName + ' ' + val
})
```

Computed example

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

Computed Setters

Computed properties will automatically be recomputed whenever any data on which the computation depends changes. However, if you need to manually change a computed property, Vue allows you to create a setter method to do this:

Template (from the basic example above):

```
<div id="example">
  a={{ a }}, b={{ b }}
</div>
```

Javascript:

```
var vm = new Vue({
  el: '#example',
  data: {
    a: 1
  },
  computed: {
    b: {
      // getter
      get: function () {
        return this.a + 1
      },
      // setter
      set: function (newValue) {
        this.a = newValue - 1
      }
    }
  }
```

You can now invoke either the getter or the setter:

```
console.log(vm.b)       // -> 2
vm.b = 4                // (setter)
console.log(vm.b)       // -> 4
console.log(vm.a)       // -> 3
```

`vm.b = 4` will invoke the setter, and set this.a to 3; by extension, vm.b will evaluate to 4.

Using computed setters for v-model

You might need a `v-model` on a computed property. Normally, the v-model won't update the computed property value.

The template:

```
<div id="demo">
  <div class='inline-block card'>
    <div :class='{onlineMarker: true, online: status, offline: !status}'></div>
    <p class='user-state'>User is {{ (status) ? 'online' : 'offline' }}</p>
  </div>

  <div class='margin-5'>
```


www.dbooks.org

```
    <input type='checkbox' v-model='status'>Toggle status (This will show you as offline to
others)
  </div>
</div>
```

Styling:

```
#demo {
  font-family: Helvetica;
  font-size: 12px;
}
.inline-block > * {
  display: inline-block;
}
.card {
  background: #ddd;
  padding:2px 10px;
  border-radius: 3px;
}
.onlineMarker {
  width: 10px;
  height: 10px;
  border-radius: 50%;
  transition: all 0.5s ease-out;
}

.online {
  background-color: #3C3;
}

.offline {
  background-color: #aaa;
}

.user-state {
  text-transform: uppercase;
  letter-spacing: 1px;
}

.margin-5 {
  margin: 5px;
}
```

The component:

```
var demo = new Vue({
    el: '#demo',
    data: {
        statusProxy: null
    },
    computed: {
        status: {
            get () {
                return (this.statusProxy === null) ? true : this.statusProxy
            }
        }
    }
})
```

fiddle Here you would see, clicking the radio button has no use at all, your status is still online.

```
var demo = new Vue({
    el: '#demo',
    data: {
        statusProxy: null
    },
    computed: {
        status: {
            get () {
                return (this.statusProxy === null) ? true : this.statusProxy
            },
            set (val) {
                this.statusProxy = val
            }
        }
    }
})
```

fiddle And now you can see the toggle happens as the checkbox is checked/unchecked.

Read Computed Properties online: https://riptutorial.com/vue-js/topic/2371/computed-properties

www.dbooks.org

Chapter 4: Conditional Rendering

Syntax

- `<element v-if="condition"></element>` //v-if
- `<element v-if="condition"></element><element v-else="condition"></element>` //v-if | v-else
- `<template v-if="condition">...</template>` //templated v-if
- `<element v-show="condition"></element>` //v-show

Remarks

It is very important to remember the difference between `v-if` and `v-show`. While their uses are almost identical, an element bound to `v-if` *will only render into the DOM* when it's condition is `true` for the **first time**. When using the `v-show` directive, *all elements **are** rendered into the DOM* but are hidden using the `display` style if the condition is `false`!

Examples

Overview

In Vue.js, conditional rendering is achieved by using a set of directives on elements in the template.

`v-if`

Element displays normally when condition is `true`. When the condition is `false`, only *partial* compilation occurs and the element isn't rendered into the DOM until the condition becomes `true`.

`v-else`

Does not accept a condition, but rather renders the element if the previous element's `v-if` condition is `false`. Can only be used after an element with the `v-if` directive.

`v-show`

Behaves similarly to `v-if`, however, the element will *always* be rendered into the DOM, even when the condition is `false`. If the condition is `false`, this directive will simply set the element's `display` style to `none`.

v-if / v-else

Assuming we have a `Vue.js` instance defined as:

```
var vm = new Vue({
    el: '#example',
```

```
    data: {
        a: true,
        b: false
    }
});
```

You can conditionally render any html element by including the v-if directive; the element that contains v-if will only render if the condition evaluates to true:

```
<!-- will render 'The condition is true' into the DOM -->
<div id="example">
    <h1 v-if="a">The condition is true</h1>
</div>
```

The `<h1>` element will render in this case, because the variable 'a' is true. v-if can be used with any expression, computed property, or function that evaluates to a boolean:

```
<div v-if="0 === 1">                  false; won't render</div>
<div v-if="typeof(5) === 'number'">   true; will render</div>
```

You can use a `template` element to group multiple elements together for a single condition:

```
<!-- in this case, nothing will be rendered except for the containing 'div' -->
<div id="example">
    <template v-if="b">
        <h1>Heading</h1>
        <p>Paragraph 1</p>
        <p>Paragraph 2</p>
    </template>
</div>
```

When using `v-if`, you also have the option of integrating a counter condition with the `v-else` directive. The content contained inside the element will only be displayed if the condition of the previous v-if was false. Note that this means that an element with v-else must appear immediately after an element with v-if.

```
<!-- will render only 'ELSE' -->
<div id="example">
    <h1 v-if="b">IF</h1>
    <h1 v-else="a">ELSE</h1>
</div>
```

Just as with v-if, with v-else you can group multiple html elements together within a `<template>`:

```
<div v-if="'a' === 'b'"> This will never be rendered. </div>
<template v-else>
    <ul>
      <li> You can also use templates with v-else. </li>
      <li> All of the content within the template </li>
      <li> will be rendered. </li>
    </ul>
</template>
```


www.dbooks.org

v-show

The use of the `v-show` directive is almost identical to that of `v-if`. The only differences are that `v-show` *does not* support the `<template>` syntax, and there is no "alternative" condition.

```
var vm = new Vue({
    el: '#example',
    data: {
        a: true
    }
});
```

The basic use is as follows...

```
<!-- will render 'Condition met' -->
<div id="example">
    <h1 v-show="a">Condition met</h1>
</div>
```

While `v-show` does not support the `v-else` directive to define "alternative" conditions, this can be accomplished by negating the previous one...

```
<!-- will render 'This is shown' -->
<div id="example">
    <h1 v-show="!a">This is hidden</h1>
    <h1 v-show="a">This is shown</h1>
</div>
```

Read Conditional Rendering online: https://riptutorial.com/vue-js/topic/3465/conditional-rendering

Chapter 5: Custom Components with v-model

Introduction

Often times we have to create some components which perform some actions/operations on data and we require that in the parent component. Most of the times `vuex` would be a better solution, but in cases where the child component's behavior has nothing to do with application state, for instance: A range-slider, date/time picker, file reader

Having individual stores for each component each time they get used gets complicated.

Remarks

To have `v-model` on a component you need to fulfil two conditions.

1. It should have a prop named 'value'
2. It should emit an `input` event with the value expected by the parent components.

```
<component v-model='something'></component>
```

is just syntactic sugar for

```
<component
    :value="something"
    @input="something = $event.target.value"
>
</component>
```

Examples

v-model on a counter component

Here `counter` is a child component accessed by `demo` which is a parent component using `v-model`.

```
// child component
Vue.component('counter', {
  template: `<div><button @click='add'>+1</button>
  <button @click='sub'>-1</button>
  <div>this is inside the child component: {{ result }}</div></div>`,
  data () {
    return {
      result: 0
    }
  },
  props: ['value'],
  methods: {
    emitResult () {
```


www.dbooks.org

```
      this.$emit('input', this.result)
    },
    add () {
      this.result += 1
      this.emitResult()
    },
    sub () {
      this.result -= 1
      this.emitResult()
    }
  }
})
```

This child component will be emitting `result` each time `sub()` or `add()` methods are called.

```
// parent component
new Vue({
  el: '#demo',
  data () {
    return {
      resultFromChild: null
    }
  }
})

// parent template
<div id='demo'>
  <counter v-model='resultFromChild'></counter>
  This is in parent component {{ resultFromChild }}
</div>
```

Since `v-model` is present on the child component, a prop with name `value` was sent at the same time, there is an input event on the `counter` which will in turn provide the value from the child component.

Read Custom Components with v-model online: https://riptutorial.com/vue-js/topic/9353/custom-components-with-v-model

Chapter 6: Custom Directives

Syntax

- `Vue.directive(id, definition);`
- `Vue.directive(id, update); //when you need only the update function.`

Parameters

Parameter	Details
`id`	String - The directive id that will be used without the `v-` prefix. (Add the `v-` prefix when using it)
`definition`	Object - A definition object can provide several hook functions (all optional): `bind`, `update`, and `unbind`

Examples

Basics

In addition to the default set of directives shipped in core, Vue.js also allows you to register custom directives. Custom directives provide a mechanism for mapping data changes to arbitrary DOM behavior.

You can register a global custom directive with the `Vue.directive(id, definition)` method, passing in a directive id followed by a definition object. You can also register a local custom directive by including it in a component's `directives` option.

Hook Functions

- **bind**: called only once, when the directive is first bound to the element.
- **update**: called for the first time immediately after `bind` with the initial value, then again whenever the binding value changes. The new value and the previous value are provided as the argument.
- **unbind**: called only once, when the directive is unbound from the element.

```
Vue.directive('my-directive', {
     bind: function () {
       // do preparation work
       // e.g. add event listeners or expensive stuff
       // that needs to be run only once
     },
     update: function (newValue, oldValue) {
       // do something based on the updated value
       // this will also be called for the initial value
     },
```


www.dbooks.org

```
    unbind: function () {
      // do clean up work
      // e.g. remove event listeners added in bind()
    }
})
```

Once registered, you can use it in Vue.js templates like this (remember to add the `v-` prefix):

```
<div v-my-directive="someValue"></div>
```

When you only need the `update` function, you can pass in a single function instead of the definition object:

```
Vue.directive('my-directive', function (value) {
  // this function will be used as update()
})
```

Directive Instance Properties

All the hook functions will be copied into the actual directive object, which you can access inside these functions as their `this` context. The directive object exposes some useful properties:

- **el**: the element the directive is bound to.
- **vm**: the context ViewModel that owns this directive.
- **expression**: the expression of the binding, excluding arguments and filters.
- **arg**: the argument, if present.
- **name**: the name of the directive, without the prefix.
- **modifiers**: an object containing modifiers, if any.
- **descriptor**: an object that contains the parsing result of the entire directive.
- **params**: an object containing param attributes. Explained below.

  You should treat all these properties as read-only and never modify them. You can attach custom properties to the directive object too, but be careful not to accidentally overwrite existing internal ones.

An example of a custom directive using some of these properties:

HTML

```
<div id="demo" v-demo:hello.a.b="msg"></div>
```

JavaScript

```
Vue.directive('demo', {
  bind: function () {
    console.log('demo bound!')
  },
  update: function (value) {
    this.el.innerHTML =
      'name - '       + this.name + '<br>' +
      'expression - ' + this.expression + '<br>' +
```

```
      'argument - '   + this.arg + '<br>' +
      'modifiers - '  + JSON.stringify(this.modifiers) + '<br>' +
      'value - '      + value
  }
})
var demo = new Vue({
  el: '#demo',
  data: {
    msg: 'hello!'
  }
})
```

Result

```
name - demo
expression - msg
argument - hello
modifiers - {"b":true,"a":true}
value - hello!
```

Object Literal

If your directive needs multiple values, you can also pass in a JavaScript object literal. Remember, directives can take any valid JavaScript expression:

HTML

```
<div v-demo="{ color: 'white', text: 'hello!' }"></div>
```

JavaScript

```
Vue.directive('demo', function (value) {
  console.log(value.color) // "white"
  console.log(value.text) // "hello!"
})
```

Literal Modifier

When a directive is used with the literal modifier, its attribute value will be interpreted as a plain string and passed directly into the `update` method. The `update` method will also be called only once, because a plain string cannot be reactive.

HTML

```
<div v-demo.literal="foo bar baz">
```

JavaScript

```
Vue.directive('demo', function (value) {
  console.log(value) // "foo bar baz"
})
```


www.dbooks.org

Read Custom Directives online: https://riptutorial.com/vue-js/topic/2368/custom-directives

Chapter 7: Custom Filters

Syntax

- `Vue.filter(name, function(value){});` //Basic
- `Vue.filter(name, function(value, begin, end){});` //Basic with wrapping values
- `Vue.filter(name, function(value, input){});` //Dynamic
- `Vue.filter(name, { read: function(value){}, write: function(value){} });` //Two-way

Parameters

Parameter	Details
name	String - desired callable name of the filter
value	[Callback] Any - value of the data passing into the filter
begin	[Callback] Any - value to come before the passed data
end	[Callback] Any - value to come after the passed data
input	[Callback] Any - user input bound to Vue instance for dynamic results

Examples

Two-way Filters

With a `two-way filter`, we are able to assign a `read` *and* `write` operation for a single `filter` that changes the value of the *same* data between the `view` and `model`.

```
//JS
Vue.filter('uppercase', {
    //read : model -> view
    read: function(value) {
        return value.toUpperCase();
    },

    //write : view -> model
    write: function(value) {
        return value.toLowerCase();
    }
});

/*
 * Base value of data: 'example string'
 *
 * In the view : 'EXAMPLE STRING'
 * In the model : 'example string'
```


www.dbooks.org

```
 */
```

Basic

Custom filters in `Vue.js` can be created easily in a single function call to `Vue.filter`.

```
//JS
Vue.filter('reverse', function(value) {
    return value.split('').reverse().join('');
});

//HTML
<span>{{ msg | reverse }}</span> //'This is fun!' => '!nuf si sihT'
```

It is good practice to store all custom filters in separate files e.g. under `./filters` as it is then easy to re-use your code in your next application. If you go this way you have to **replace JS part**:

```
//JS
Vue.filter('reverse', require('./filters/reverse'));
```

You can also define your own `begin` and `end` wrappers as well.

```
//JS
Vue.filter('wrap', function(value, begin, end) {
    return begin + value + end;
});

//HTML
<span>{{ msg | wrap 'The' 'fox' }}</span> //'quick brown' => 'The quick brown fox'
```

Read Custom Filters online: https://riptutorial.com/vue-js/topic/1878/custom-filters

Chapter 8: Data Binding

Examples

Text

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

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

The mustache tag will be replaced with the value of the `msg` property on the corresponding data object. It will also be updated whenever the data object’s `msg` property changes.

You can also perform one-time interpolations that do not update on data change:

```
<span>This will never change: {{* msg }}</span>
```

Raw HTML

The double mustaches interprets the data as plain text, not HTML. In order to output real HTML, you will need to use triple mustaches:

```
<div>{{{ raw_html }}}</div>
```

The contents are inserted as plain HTML - data bindings are ignored. If you need to reuse template pieces, you should use partials.

Attributes

Mustaches can also be used inside HTML attributes:

```
<div id="item-{{ id }}"></div>
```

Note that attribute interpolations are disallowed in Vue.js directives and special attributes. Don’t worry, Vue.js will raise warnings for you when mustaches are used in wrong places.

Filters

Vue.js allows you to append optional “filters” to the end of an expression, denoted by the “pipe” symbol:

```
{{ message | capitalize }}
```

Here we are “piping” the value of the `message` expression through the built-in `capitalize` filter, which

www.dbooks.org

is in fact just a JavaScript function that returns the capitalized value. Vue.js provides a number of built-in filters, and we will talk about how to write your own filters later.

Note that the pipe syntax is not part of JavaScript syntax, therefore you cannot mix filters inside expressions; you can only append them at the end of an expression.

Filters can be chained:

```
{{ message | filterA | filterB }}
```

Filters can also take arguments:

```
{{ message | filterA 'arg1' arg2 }}
```

The filter function always receives the expression's value as the first argument. Quoted arguments are interpreted as plain string, while un-quoted ones will be evaluated as expressions. Here, the plain string `'arg1'` will be passed into the filter as the second argument, and the value of expression `arg2` will be evaluated and passed in as the third argument.

Read Data Binding online: https://riptutorial.com/vue-js/topic/1213/data-binding

Chapter 9: Dynamic Components

Remarks

`<component>` is a reserved component element, don't be confused with components instance.

`v-bind` is a directive. Directives are prefixed with *v-* to indicate that they are special attributes provided by Vue.

Examples

Simple Dynamic Components Example

Dynamically switch beetween multiple components using `<component>` element and pass data to *v-bind:is* attribute:

Javascript:

```
new Vue({
  el: '#app',
  data: {
    currentPage: 'home'
  },
  components: {
    home: {
      template: "<p>Home</p>"
    },
    about: {
      template: "<p>About</p>"
    },
    contact: {
      template: "<p>Contact</p>"
    }
  }
})
```

HTML:

```
<div id="app">
   <component v-bind:is="currentPage">
       <!-- component changes when currentPage changes! -->
       <!-- output: Home -->
   </component>
</div>
```

Snippet:

www.dbooks.org

Live Demo

Pages Navigation with keep-alive

Sometimes you want to keep the switched-out components in memory, to make that happen, you should use `<keep-alive>` element:

Javascript:

```
new Vue({
  el: '#app',
  data: {
    currentPage: 'home',
  },
  methods: {
     switchTo: function(page) {
            this.currentPage = page;
     }
  },
  components: {
    home: {
      template: `<div>
                 <h2>Home</h2>
                 <p>{{ homeData }}</p>
                 </div>`,
      data: function() {
         return {
            homeData: 'My about data'
         }
       }
     },
    about: {
      template: `<div>
                 <h2>About</h2>
                 <p>{{ aboutData }}</p>
                 </div>`,
      data: function() {
         return {
            aboutData: 'My about data'
         }
      }
    },
    contact: {
      template: `<div>
             <h2>Contact</h2>
             <form method="POST" @submit.prevent>
             <label>Your Name:</label>
             <input type="text" v-model="contactData.name" >
             <label>You message: </label>
             <textarea v-model="contactData.message"></textarea>
             <button type="submit">Send</button>
             </form>
             </div>`,
       data: function() {
          return {
            contactData: { name:'', message:'' }
         }
      }
```

```
    }
  }
})
```

HTML:

```
<div id="app">
  <div class="navigation">
    <ul>
      <li><a href="#home" @click="switchTo('home')">Home</a></li>
      <li><a href="#about" @click="switchTo('about')">About</a></li>
      <li><a href="#contact" @click="switchTo('contact')">Contact</a></li>
    </ul>
  </div>

  <div class="pages">
    <keep-alive>
      <component :is="currentPage"></component>
    </keep-alive>
  </div>
</div>
```

CSS:

```
.navigation {
  margin: 10px 0;
}

.navigation ul {
  margin: 0;
  padding: 0;
}

.navigation ul li {
  display: inline-block;
  margin-right: 20px;
}

input, label, button {
  display: block
}

input, textarea {
  margin-bottom: 10px;
}
```

Snippet:

Live Demo

Read Dynamic Components online: https://riptutorial.com/vue-js/topic/7702/dynamic-components

www.dbooks.org

Chapter 10: Event Bus

Introduction

Event buses are a useful way of communicating between components which are not directly related, i.e. Have no parent-child relationship.

It is just an empty `vue` instance, which can be used to `$emit` events or listen `$on` the said events.

Syntax

1. export default new Vue()

Remarks

Use vuex if your application has a lot of components requiring the data of each other.

Examples

eventBus

```
// setup an event bus, do it in a separate js file
var bus = new Vue()

// imagine a component where you require to pass on a data property
// or a computed property or a method!

Vue.component('card', {
  template: `<div class='card'>
    Name:
      <div class='margin-5'>
          <input v-model='name'>
    </div>
    <div class='margin-5'>
      <button @click='submit'>Save</button>
    </div>
  </div>`,
  data() {
    return {
      name: null
    }
  },
  methods: {
    submit() {
      bus.$emit('name-set', this.name)
    }
  }
})

// In another component that requires the emitted data.
```

```
var data = {
  message: 'Hello Vue.js!'
}

var demo = new Vue({
  el: '#demo',
  data: data,
  created() {
    console.log(bus)
    bus.$on('name-set', (name) => {
      this.message = name
    })
  }
})
```

Read Event Bus online: https://riptutorial.com/vue-js/topic/9498/event-bus

www.dbooks.org

Chapter 11: Events

Examples

Events syntax

To send an event: `vm.$emit('new-message');`

To catch an event: `vm.$on('new-message');`

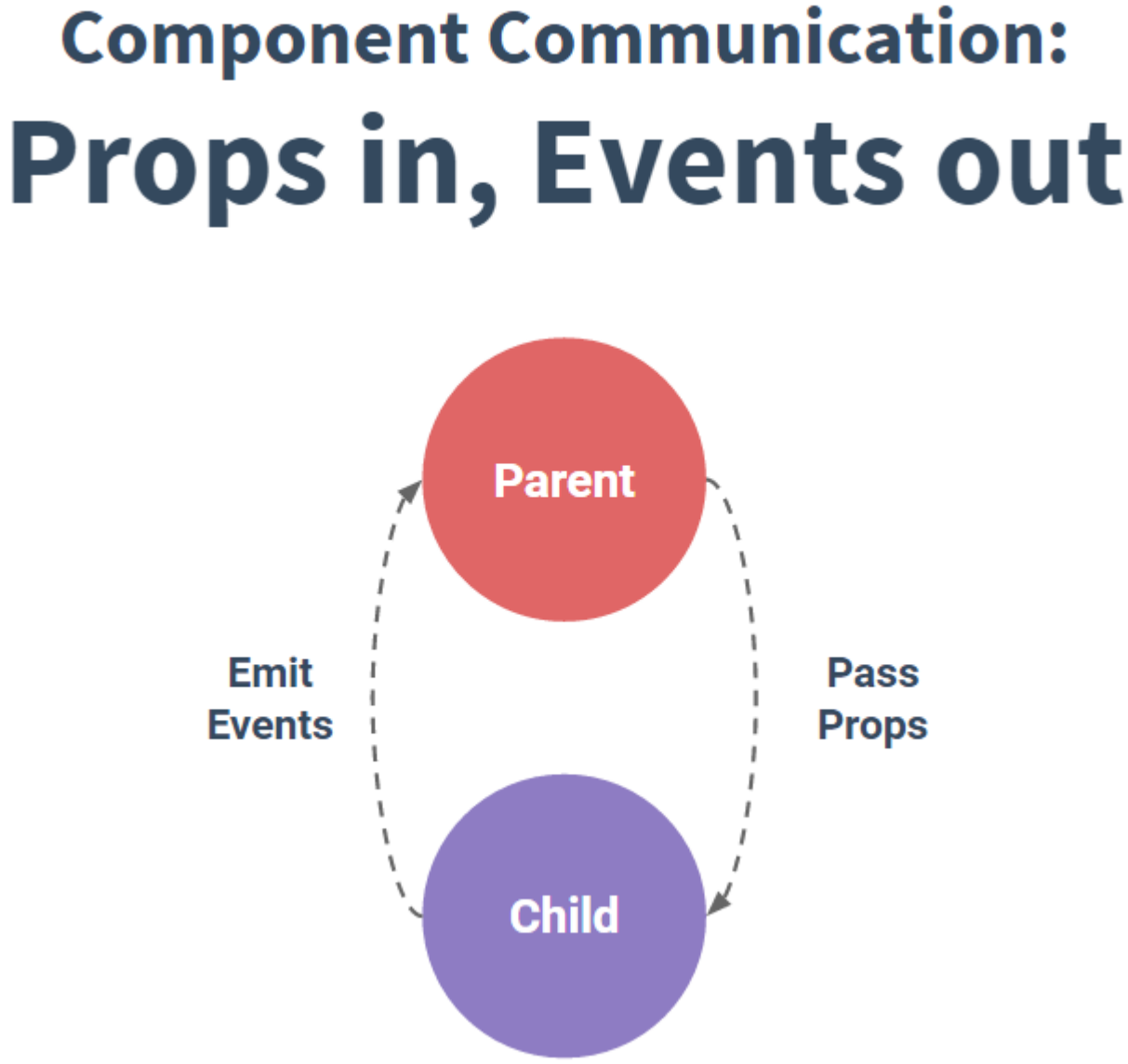
To send an event to all components **down**: `vm.$broadcast('new-message');`

To send an event to all components **up**: `vm.$dispatch('new-message');`

Note: `$broadcast` and `$dispatch` are deprecated in Vue2. (see Vue2 features)

When should I use events ?

The following picture illustrates how component communication should work. The picture comes from The Progressive Framework slides of Evan You (Developer of VueJS).



Here is an example of how it works :

DEMO

HTML

```
<script type="x-template" id="message-box">
    <input type="text" v-model="msg" @keyup="$emit('new-message', msg)" />
</script>

<message-box :msg="message" @new-message="updateMessage"></message-box>
<div>You typed: {{message}}</div>
```

JS

```
var messageBox = {
  template: '#message-box',
  props: ['msg']
};

new Vue({
  el: 'body',
```


www.dbooks.org

```
  data: {
    message: ''
  },
  methods: {
    updateMessage: function(msg) {
      this.message = msg;
    }
  },
  components: {
    'message-box': messageBox
  }
});
```

The example above can be improved !

The example above shows how the component communication works. But in case of a custom input component, to synchronize the parent variable with the value typed, we sould use `v-model`.

DEMO Vue1

DEMO Vue2

In Vue1, you should use `.sync` on the prop sent to the `<message-box>` component. This tells VueJS to synchronize the value in the child component with the parent's.

> **Remember:** Every component instance has its own isolated scope.

HTML Vue1

```
<script type="x-template" id="message-box">
  <input v-model="value" />
</script>

<div id="app">
  <message-box :value.sync="message"></message-box>
  <div>You typed: {{message}}</div>
</div>
```

In Vue2, there is a special `'input'` event you can `$emit`. Using this event allows you to put a `v-model` directly on the `<message-box>` component. The example will look as follow:

HTML Vue2

```
<script type="x-template" id="message-box">
  <input :value="value" @input="$emit('input', $event.target.value)" />
</script>

<div id="app">
  <message-box v-model="message"></message-box>
  <div>You typed: {{message}}</div>
</div>
```

JS Vue 1 & 2

```
var messageBox = {
  template: '#message-box',
  props: ['value']
};

new Vue({
  el: '#app',
  data: {
    message: ''
  },
  components: {
    'message-box': messageBox
  }
});
```

Notice how faster the input is updated.

How to deal with deprecation of $dispatch and $broadcast? (bus event pattern)

You might have realized that `$emit` is scoped to the component that is emitting the event. That's a problem when you want to communicate between components far from one another in the component tree.

Note: In Vue1 you coud use `$dispatch` or `$broadcast`, but not in Vue2. The reason being that it doesn't scale well. There is a popular `bus` pattern to manage this:

DEMO

HTML

```
<script type="x-template" id="sender">
  <button @click="bus.$emit('new-event')">Click me to send an event !</button>
</script>

<script type="x-template" id="receiver">
  <div>I received {{numberOfEvents}} event{{numberOfEvents == 1 ? '' : 's'}}</div>
</script>

<sender></sender>
<receiver></receiver>
```

JS

```
var bus = new Vue();

var senderComponent = {
  template: '#sender',
  data() {
    return {
      bus: bus
    }
  }
};
```


www.dbooks.org

```
var receiverComponent = {
  template: '#receiver',
  data() {
    return {
      numberOfEvents: 0
    }
  },
  ready() {
    var self = this;

    bus.$on('new-event', function() {
      ++self.numberOfEvents;
    });
  }
};

new Vue({
  el: 'body',
  components: {
    'sender': senderComponent,
    'receiver': receiverComponent
  }
});
```

You just need to understand that any `Vue()` instance can `$emit` and catch (`$on`) an event. We just declare a global `Vue` instance call `bus` and then any component with this variable can emit and catch events from it. Just make sure the component has access to the `bus` variable.

Read Events online: https://riptutorial.com/vue-js/topic/5941/events

Chapter 12: Lifecycle Hooks

Examples

Hooks for Vue 1.x

- **`init`**

  Called synchronously after the instance has been initialized and prior to any initial data observation.

- **`created`**

  Called synchronously after the instance is created. This occurs prior to `$el` setup, but after `data observation`, `computed properties`, `watch/event callbacks`, and `methods` have been setup.

- **`beforeCompile`**

  Immediately prior to compilation of the Vue instance.

- **`compiled`**

  Immediately after compilation has completed. All `directives` are linked but still prior to `$el` being available.

- **`ready`**

  Occurs after compilation *and* `$el` are complete and the instance is injected into the DOM for the first time.

- **`attached`**

  Occurs when `$el` is attached to the DOM by a `directive` or instance calls `$appendTo()`.

- **`detached`**

  Called when `$el` is removed/detached from the DOM or instance method.

- **`beforeDestroy`**

  Immediately before the Vue instance is destroyed, but is still fully functional.

- **`destroyed`**

  Called after an instance is destroyed. All `bindings` and `directives` have already been unbound and child instances have also been destroyed.

www.dbooks.org

Using in an Instance

Since *all* lifecycle hooks in `Vue.js` are just `functions`, you can place *any* of them directly in the instance declaraction.

```
//JS
new Vue({

    el: '#example',

    data: {
        ...
    },

    methods: {
        ...
    },

    //LIFECYCLE HOOK HANDLING
    created: function() {
        ...
    },

    ready: function() {
        ...
    }

});
```

Common Pitfalls: Accessing DOM from the `ready()` hook

A common usecase for the `ready()` hook is to access the DOM, e.g. to initiate a Javascript plugin, get the dimensions of an element etc.

The problem

Due to Vue's asynchronous DOM update mechanism, it's not guaranteed that the DOM has been fully updated when the `ready()` hook is called. This usually results in an error because the element is undefined.

The Solution

For this situation, the `$nextTick()` instance method can help. This method defers the execution of the provided callback function until after the next tick, which means that it is fired when all DOM updates are guaranteed to be finished.

Example:

```
module.exports {
  ready: function () {
    $('.cool-input').initiateCoolPlugin() //fails, because element is not in DOM yet.

    this.$nextTick(function() {
      $('.cool-input').initiateCoolPlugin() // this will work because it will be executed
```

```
after the DOM update.
    })
  }
}
```

Read Lifecycle Hooks online: https://riptutorial.com/vue-js/topic/1852/lifecycle-hooks

Chapter 13: List Rendering

Examples

Basic Usage

A list can be rendered using the `v-for` directive. The syntax requires that you specify the source array to iterate on, and an alias that will be used to reference each item in the iteration. In the following example we use `items` as the source array, and `item` as the alias for each item.

HTML

```
<div id="app">
  <h1>My List</h1>
  <table>
    <tr v-for="item in items">
      <td>{{item}}</td>
    </tr>
  </table>
</div>
```

Script

```
new Vue({
  el: '#app',
  data: {
    items: ['item 1', 'item 2', 'item 3']
  }
})
```

You can view a working demo here.

Only render HTML items

In this example will render five `<li>` tags

```
<ul id="render-sample">
  <li v-for="n in 5">
    Hello Loop
  </li>
</ul>
```

Pig countdown list

```
<ul>
  <li v-for="n in 10">{{11 - n}} pigs are tanning at the beach. One got fried, and
</ul>
```

https://jsfiddle.net/gurghet/3jeyka22/

Iteration over an object

`v-for` can be used for iterating over an object keys (and values):

HTML:

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

Script:

```
new Vue({
  el: '#repeat-object',
  data: {
    object: {
      FirstName: 'John',
      LastName: 'Doe',
      Age: 30
    }
  }
})
```

Read List Rendering online: https://riptutorial.com/vue-js/topic/1972/list-rendering

www.dbooks.org

Chapter 14: Mixins

Examples

Global Mixin

You can also apply a mixin globally. Use caution! Once you apply a mixin globally, it will affect **every** Vue instance created afterwards. When used properly, this can be used to inject processing logic for custom options:

```
// inject a handler for `myOption` custom option
Vue.mixin({
  created: function () {
    var myOption = this.$options.myOption
    if (myOption) {
      console.log(myOption)
    }
  }
})

new Vue({
  myOption: 'hello!'
})
// -> "hello!"
```

> Use global mixins sparsely and carefully, because it affects every single Vue instance created, including third party components. In most cases, you should only use it for custom option handling like demonstrated in the example above.

Custom Option Merge Strategies

When custom options are merged, they use the default strategy, which simply overwrites the existing value. If you want a custom option to be merged using custom logic, you need to attach a function to `Vue.config.optionMergeStrategies`:

```
Vue.config.optionMergeStrategies.myOption = function (toVal, fromVal) {
  // return mergedVal
}
```

For most object-based options, you can simply use the same strategy used by `methods`:

```
var strategies = Vue.config.optionMergeStrategies
strategies.myOption = strategies.methods
```

Basics

Mixins are a flexible way to distribute reusable functionalities for Vue components. A mixin object can contain any component options. When a component uses a mixin, all options in the mixin will

be “mixed” into the component’s own options.

```
// define a mixin object
var myMixin = {
  created: function () {
    this.hello()
  },
  methods: {
    hello: function () {
      console.log('hello from mixin!')
    }
  }
}

// define a component that uses this mixin
var Component = Vue.extend({
  mixins: [myMixin]
})

var component = new Component() // -> "hello from mixin!"
```

Option Merging

When a mixin and the component itself contain overlapping options, they will be “merged” using appropriate strategies. For example, hook functions with the same name are merged into an array so that all of them will be called. In addition, mixin hooks will be called **before** the component’s own hooks:

```
var mixin = {
  created: function () {
    console.log('mixin hook called')
  }
}

new Vue({
  mixins: [mixin],
  created: function () {
    console.log('component hook called')
  }
})

// -> "mixin hook called"
// -> "component hook called"
```

Options that expect object values, for example `methods`, `components` and `directives`, will be merged into the same object. The component’s options will take priority when there are conflicting keys in these objects:

```
var mixin = {
  methods: {
    foo: function () {
      console.log('foo')
    },
    conflicting: function () {
      console.log('from mixin')
```


www.dbooks.org

```
    }
  }
}

var vm = new Vue({
  mixins: [mixin],
  methods: {
    bar: function () {
      console.log('bar')
    },
    conflicting: function () {
      console.log('from self')
    }
  }
})

vm.foo() // -> "foo"
vm.bar() // -> "bar"
vm.conflicting() // -> "from self"
```

Note that the same merge strategies are used in `Vue.extend()`.

Read Mixins online: https://riptutorial.com/vue-js/topic/2562/mixins

Chapter 15: Modifiers

Introduction

There are some frequently used operations like `event.preventDefault()` or `event.stopPropagation()` inside event handlers. Although we can do this easily inside methods, it would be better if the methods can be purely about data logic rather than having to deal with DOM event details.

Examples

Event Modifiers

Vue provides event modifiers for `v-on` by calling directive postfixes denoted by a dot.

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

For examples:

```
<!-- 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>

<!-- use capture mode when adding the event listener -->
<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

When listening for keyboard events, we often need to check for common key codes. Remembering all the keyCodes is a hassle, so Vue provides aliases for the most commonly used keys:

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

www.dbooks.org

For examples:

```
<input v-on:keyup.enter="submit">
```

Input Modifiers

- `.trim`

If you want user input to be trimmed automatically, you can add the `trim` modifier to your `v-model` managed inputs:

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

- `.number`

If you want user input to be automatically typecast as a number, you can do as follow:

```
<input v-model.number="age" type="number">
```

- `.lazy`

Generally, `v-model` syncs the input with the data after each input event, but you can add the `lazy` modifier to instead sync after change events:

```
<input v-model.lazy="msg" >
```

Read Modifiers online: https://riptutorial.com/vue-js/topic/8612/modifiers

Chapter 16: Plugins

Introduction

Vue plugins adds global functionality as, global methods, directives, transitions, filters, instance methods, objects and inject some component options using mixins

Syntax

- MyPlugin.install = function (Vue, options) {}

Parameters

Name	Description
Vue	Vue constructor, injected by Vue
options	Additional options if needed

Remarks

In most cases you will need to explicitly tell Vue to use a plugin

```
// calls `MyPlugin.install(Vue)`
Vue.use(MyPlugin)
```

To pass options

```
Vue.use(MyPlugin, { someOption: true })
```

Examples

Simple logger

```
//myLogger.js
export default {

  install(Vue, options) {
     function log(type, title, text) {
       console.log(`[${type}] ${title} - ${text}`);
     }

     Vue.prototype.$log = {
       error(title, text) { log('danger', title, text) },
       success(title, text) { log('success', title, text) },
```


www.dbooks.org

```
      log
    }
  }
}
```

Before your main Vue instance tell to register your plugin

```
//main.js
import Logger from './path/to/myLogger';

Vue.use(Logger);

var vm = new Vue({
  el: '#app',
  template: '<App/>',
  components: { App }
})
```

Now you can call `this.$log` on any child component

```
//myComponent.vue
export default {
  data() {
    return {};
  },
  methods: {
    Save() {
      this.$log.success('Transaction saved!');
    }
  }
}
```

Read Plugins online: https://riptutorial.com/vue-js/topic/8726/plugins

Chapter 17: Polyfill "webpack" template

Parameters

Files or packages	Command or configuration to modify
babel-polyfill	`npm i -save babel-polyfill`
karma.conf.js	`files: ['../../node_modules/babel-polyfill/dist/polyfill.js','./index.js'],`
webpack.base.conf.js	`app: ['babel-polyfill', './src/main.js']`

Remarks

The configurations described above, the example using a non-sstandardised function will work on "internet explorer" and `npm test` will pass.

Examples

Usage of functions to polyfill (ex: find)

```
<template>
  <div class="hello">
    <p>{{ filtered() }}</p>
  </div>
</template>

<script>
export default {
  name: 'hello',
  data () {
    return {
      list: ['toto', 'titi', 'tata', 'tete']
    }
  },
  methods: {
    filtered () {
      return this.list.find((el) => el === 'tata')
    }
  }
}
</script>
```

Read Polyfill "webpack" template online: https://riptutorial.com/vue-js/topic/9174/polyfill--webpack--template

www.dbooks.org

Chapter 18: Props

Remarks

camelCase <=> kebab-case

When defining the names of your `props`, always remember that HTML attribute names are case-insensitive. That means if you define a `prop` in camel case in your component definition...

```
Vue.component('child', {
    props: ['myProp'],
    ...
});
```

...you must call it in your HTML component as my-prop.

Examples

Passing Data from parent to child with props

In Vue.js, every component instance has ***its own isolated scope***, which means that if a parent component has a child component - the child component has its own isolated scope and the parent component has its own isolated scope.

For any medium to large size app, following best practices conventions prevents lots of headaches during the development phase and then after while maintenance. One of such things to follow is that ***avoid referencing/mutating parent data directly from the child component***. So then how do we reference the parent data from within a child component?

Whatever parent data is required in a child component should be passed to the child as `props` from the parent.

Use Case: Suppose we have a User database with two tables `users` and `addresses` with the following fields:
`users` Table

name	phone	email
John Mclane	(1) 234 5678 9012	john@dirhard.com
James Bond	(44) 777 0007 0077	bond@mi6.com

`addresses` Table

block	street	city
Nakatomi Towers	Broadway	New York
Mi6 House	Buckingham Road	London

and we want to have three components to display corresponding user information anywhere in our app

user-component.js

```
export default{
    template:`<div class="user-component">
                <label for="name" class="form-control">Name: </label>
                <input class="form-control input-sm" name="name" v-model="name">
                <contact-details :phone="phone" :email="email"></contact-details>
              </div>`,
    data(){
        return{
            name:'',
            phone:'',
            email:''
        }
    },
}
```

contact-details.js

```
import Address from './address';
export default{
    template:`<div class="contact-details-component>
                <h4>Contact Details:</h4>
                <label for="phone" class="form-control">Phone: </label>
                <input class="form-control input-sm" name="phone" v-model="phone">
                <label for="email" class="form-control">Email: </label>
                <input class="form-control input-sm" name="email" v-model="email">

                <h4>Address:</h4>
                <address :address-type="addressType"></address>
                //see camelCase vs kebab-case explanation below
            </div>`,
    props:['phone', 'email'],
    data:(){
        return:{
            addressType:'Office'
        }
    },
    components:{Address}
}
```

address.js

```
export default{
    template:`<div class="address-component">
                <h6>{{addressType}}</h6>
                <label for="block" class="form-control">Block: </label>
```


www.dbooks.org

```
                <input class="form-control input-sm" name="block" v-model="block">
                <label for="street" class="form-control">Street: </label>
                <input class="form-control input-sm" name="street" v-model="street">
                <label for="city" class="form-control">City: </label>
                <input class="form-control input-sm" name="city" v-model="city">
            </div>`,
    props:{
        addressType:{
            required:true,
            type:String,
            default:'Office'
        },
    data(){
        return{
            block:'',
            street:'',
            city:''
        }
    }
}
```

main.js

```
import Vue from 'vue';

Vue.component('user-component', require'./user-component');
Vue.component('contact-details', require'./contact-details');

new Vue({
    el:'body'
});
```

index.html

```
...
<body>
    <user-component></user-component>
        ...
</body>
```

We are displaying the `phone` and `email` data, which are properties of `user-component` in `contact-details` which doesn't have phone or email data.

Passing data as props

So within the `user-component.js` in the ***template*** property, where we include the `<contact-details>` component, we are passing the ***phone*** and the ***email*** data from `<user-component>`(parent component) to `<contact-details>`(child component) by dynamically binding it to the **props** - `:phone="phone"` and `:email="email` which is same as `v-bind:phone="phone"` and `v-bind:email="email"`

Props - Dynamic Binding

Since we are dynamically binding the props any change in ***phone*** or ***email*** within the parent component i.e. `<user-component>` will immediately be reflected in the child component i.e. `<contact-details>`.

Props - as Literals

However, if we would have passed the values of ***phone*** and ***email*** as string literal values like `phone="(44) 777 0007 0077" email="bond@mi6.com"` then it would not reflect any data changes which happen in the parent component.

One-Way binding

By default the direction of changes is top to bottom i.e. any change to dynamically bound props in the parent component will propagate to the child component but any change to the prop values in a child component will not propagate to the parent.

For eg: if from within the `<contact-details>` we change the email from `bond@mi6.com` to `jamesbond@mi6.com`, the parent data i.e. phone data property in `<user-component>` will still contain a value of `bond@mi6.com`.

However, if we change the value of email from `bond@mi6.com` to `jamesbond@mi6.co` in the parent component (`<user-component>` in our use case) then the value of email in the child component (`<contact-details>` in our use case) will change to `jamesbond@mi6.com` automatically - change in parent is instantly propagated to the child.

Two-Way Binding

If we want two-way binding then we have to explicitly specify two-way binding as `:email.sync="email"` instead of `:email="email"`. Now if we change the value of prop in the child component the change will be reflected in the parent component as well.

In a medium to large app changing parent state from the child state will be very hard to detect and keep track of especially while debugging - **Be cautious** .

There won't be any .sync option available in Vue.js 2.0. **The two-way binding for props is being deprecated in Vue.js 2.0**.

One-time Binding

It is also possible to define explicit **one-time** binding as `:email.once="email`, it is more or less similar to passing a literal, because any subsequent changes in the parent property value will not propagate to the child.

CAVEAT

When **Object or Array** is passed as prop, they are **ALWAYS PASSED BY REFERENCE**, which means irrespective of the binding type explicitly defined `:email.sync="email"` or `:email="email"` or `:email.once="email"`, if email is an Object or an Array in the parent then regardless of the binding type, any change in the prop value within the child component will affect the value in the parent as well.

Props as Array

In the `contact-details.js` file we have defined `props:['phone', 'email']` as an array, which is fine if we do not want fine grained control with props.

www.dbooks.org

Props as Object

If we want more fine grained control over props, like

- if we want to define what type of values are acceptable as the prop
- what should be a default value for the prop
- whether a value is MUST (required) to be passed for the prop or is it optional

then we need to use object notation for defining the props, as we have done in `address.js`.

If we are authoring reusable components which may be used by other developers on the team as well, then it is a good practice to define props as objects so that anyone using the component has a clear idea of what should be the type of data and whether it is compulsory or optional.

It is also referred to as ***props validation***. The **type** can be any one of the following native constructors:

- String
- Number
- Boolean
- Array
- Object
- Function
- or a Custom Constructor

Some examples of prop validation as taken from http://vuejs.org/guide/components.html#Props

```
Vue.component('example', {
   props: {
       // basic type check (`null` means accept any type)
       propA: Number,
       // multiple possible types (1.0.21+)
       propM: [String, Number],
       // a required string
       propB: {
         type: String,
         required: true
       },
       // a number with default value
       propC: {
          type: Number,
          default: 100
       },
       // object/array defaults should be returned from a
       // factory function
       propD: {
          type: Object,
          default: function () {
             return { msg: 'hello' }
         }
       },
       // indicate this prop expects a two-way binding. will
       // raise a warning if binding type does not match.
       propE: {
          twoWay: true
```

```
        },
        // custom validator function
        propF: {
           validator: function (value) {
              return value > 10
           }
        },
        // coerce function (new in 1.0.12)
        // cast the value before setting it on the component
        propG: {
           coerce: function (val) {
             return val + '' // cast the value to string
           }
        },
        propH: {
           coerce: function (val) {
             return JSON.parse(val) // cast the value to Object
           }
        }
    }
});
```

camelCase vs kebab-case

HTML attributes are case-insensitive, which means it cannot differentiate between `addresstype` and `addressType`, so when using camelCase prop names as attributes we need to use their kebab-case(hyphen-delimited) equivalents:
`addressType` should be written as `address-type` in HTML attribute.

Dynamic Props

Just as you're able to bind data from a view to the model, you can also bind props using the same v-bind directive for passing information from parent to child components.

JS

```
new Vue({
    el: '#example',
    data: {
        msg: 'hello world'
    }
});

Vue.component('child', {
    props: ['myMessage'],
    template: '<span>{{ myMessage }}</span>
});
```

HTML

```
<div id="example">
    <input v-model="msg" />
    <child v-bind:my-message="msg"></child>
```


www.dbooks.org

```
    <!-- Shorthand ... <child :my-message="msg"></child> -->
</div>
```

Result

```
hello world
```

Passing Props While Using Vue JSX

We have a parent component: Importing a child component in it we'll pass props via an attribute. Here the attribute is 'src' and we're passing the 'src' too.

ParentComponent.js

```
import ChildComponent from './ChildComponent';
export default {
    render(h, {props}) {
        const src = 'https://cdn-images-1.medium.com/max/800/1*AxRXW2j8qmGJixIYg7n6uw.jpeg';
        return (
           <ChildComponent src={src} />
        );
    }
};
```

And a child component, where we need to pass props. We need to specify which props we are passing.

ChildComponent.js:

```
export default {
    props: ['src'],
    render(h, {props}) {
        return (
            <a href = {props.src} download = "myimage" >
                Click this link
            </a>
        );
    }
};
```

Read Props online: https://riptutorial.com/vue-js/topic/3080/props

Chapter 19: Slots

Remarks

Important! Slots after render don't guarantee order for positions for slots. Slot, which was the first, may have a different position after render.

Examples

Using Single Slots

Single slots are used when a child component only defines one `slot` within its template. The `page` component above uses a single slot to distribute content.

An example of the `page` component's template using a single slot is below:

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <slot>
            This will only be displayed if there is no content
            to be distributed.
        </slot>
    </body>
</html>
```

To illustrate how the slot works we can set up a page as follows.

```
<page>
    <p>This content will be displayed within the page component</p>
</page>
```

The end result will be:

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <p>This content will be displayed within the page component</p>
    </body>
</html>
```

If we didn't put anything between the `page` tags an instead had `<page></page>` we would instead yield the following result since there is default content between the `slot` tags in the `page` component template.

www.dbooks.org

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        This will only be displayed if there is no content
        to be distributed.
    </body>
</html>
```

What are slots?

Slots offer a convenient way of distributing content from a parent component to a child component. This content can be anything from text, HTML or even other components.

It can be helpful sometimes to think of slots as a means of injecting content directly into a child component's template.

Slots are especially useful when the component composition underneath the parent component isn't always the same.

Take the following example where we have a `page` component. The content of the page could change based on whether that page displays e.g. an article, blog post or form.

Article

```
<page>
    <article></article>
    <comments></comments>
</page>
```

Blog Post

```
<page>
    <blog-post></blog-post>
    <comments></comments>
</page>
```

Form

```
<page>
    <form></form>
</page>
```

Notice how the content of the `page` component can change. If we didn't use slots this would be more difficult as the inner part of the template would be fixed.

Remember: *"Everything in the parent template is compiled in parent scope; everything in the child template is compiled in child scope."*

Using Named Slots

Named slots work similarly to single slots but instead allow you to distribute content to different regions within your child component template.

Take the `page` component from the previous example but modify it's template so it is as follows:

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <aside>
            <slot name="sidebar"></slot>
        </aside>
        <main>
            <slot name="content"></slot>
        </main>
    </body>
</html>
```

When using the `page` component we can now determine where content is placed via the `slot` attribute:

```
<page>
    <p slot="sidebar">This is sidebar content.</p>
    <article slot="content"></article>
</page>
```

The resulting page will be:

```
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
        <aside>
            <p>This is sidebar content.</p>
        </aside>
        <main>
            <article></article>
        </main>
    </body>
</html>
```

If a `slot` is defined without a `name` attribute then any content which is placed within component tags not specifying a `slot` attribute will be placed into that slot.

See the multi insertion example on the Vue.js official docs.

Using Slots in Vue JSX with 'babel-plugin-transform-vue-jsx'

If you're Using VueJS2 and like to use JSX along with it. In this case,to use the slot, the solution with example is below.We have to use `this.$slots.default` It's almost like `this.props.children` in React JS.

www.dbooks.org

Component.js :

```
export default {
    render(h) {  //eslint-disable-line
        return (
            <li>
                { this.$slots.default }
            </li>
        );
    }
};
```

ParentComponent.js

```
import Component from './Component';

export default {
    render(h) {  //eslint-disable-line
        return (
            <ul>
                <Component>
                    Hello World
                </Component>
            </ul>
        );
    }
};
```

Read Slots online: https://riptutorial.com/vue-js/topic/4484/slots

Chapter 20: The array change detection caveats

Introduction

When you try to set a value of an item at a particular index of an array initialized in the data option, vue can't detect the change and does not trigger an update to the state. In order to overcome this caveat you should either use vue's Vue.$set or use Array.prototype.splice method

Examples

Using Vue.$set

In your method or any lifecycle hook that changes the array item at particuar index

```
new Vue({
    el: '#app',
    data:{
        myArr : ['apple', 'orange', 'banana', 'grapes']
    },
    methods:{
        changeArrayItem: function(){
            //this will not work
            //myArr[2] = 'strawberry';

            //Vue.$set(array, index, newValue)
            this.$set(this.myArr, 2, 'strawberry');
        }
    }
})
```

Here is the link to the fiddle

Using Array.prototype.splice

You can perform the same change instead of using `Vue.$set` by using the Array prototype's `splice()`

```
new Vue({
    el: '#app',
    data:{
        myArr : ['apple', 'orange', 'banana', 'grapes']
    },
    methods:{
        changeArrayItem: function(){
            //this will not work
            //myArr[2] = 'strawberry';

            //Array.splice(index, 1, newValue)
            this.myArr.splice(2, 1, 'strawberry');
```


www.dbooks.org

```
        }
    }
})
```

For nested array

If yoi have nested array, the following can be done

```
new Vue({
    el: '#app',
    data:{
        myArr : [
            ['apple', 'banana'],
            ['grapes', 'orange']
        ]
    },
    methods:{
        changeArrayItem: function(){
            this.$set(this.myArr[1], 1, 'strawberry');
        }
    }
})
```

Here is the link to the jsfiddle

Array of objects containing arrays

```
new Vue({
    el: '#app',
    data:{
        myArr : [
            {
                name: 'object-1',
                nestedArr: ['apple', 'banana']
            },
            {
                name: 'object-2',
                nestedArr: ['grapes', 'orange']
            }
        ]
    },
    methods:{
        changeArrayItem: function(){
            this.$set(this.myArr[1].nestedArr, 1, 'strawberry');
        }
    }
})
```

Here is the link to the fiddle

Read The array change detection caveats online: https://riptutorial.com/vue-js/topic/10679/the-array-change-detection-caveats

Chapter 21: Using "this" in Vue

Introduction

One of the most common errors we find in Vue code on StackOverflow is the misuse of `this`. The most common mistakes fall generally in two areas, using `this` in callbacks for promises or other asynchronous functions and using arrow functions to define methods, computed properties, etc.

Examples

WRONG! Using "this" in a callback inside a Vue method.

```
new Vue({
  el:"#app",
  data:{
    foo: "bar"
  },
  methods:{
    doSomethingAsynchronous(){
      setTimeout(function(){
        // This is wrong! Inside this function,
        // "this" refers to the window object.
        this.foo = "baz";
      }, 1000);
    }
  }
})
```

WRONG! Using "this" inside a promise.

```
new Vue({
  el:"#star-wars-people",
  data:{
    people: null
  },
  mounted: function(){
    $.getJSON("http://swapi.co/api/people/", function(data){
      // Again, this is wrong! "this", here, refers to the window.
      this.people = data.results;
    })
  }
})
```

RIGHT! Use a closure to capture "this"

You can capture the correct `this` using a closure.

```
new Vue({
  el:"#star-wars-people",
  data:{
```


www.dbooks.org

```
    people: null
  },
  mounted: function(){
    // Before executing the web service call, save this to a local variable
    var self = this;
    $.getJSON("http://swapi.co/api/people/", function(data){
      // Inside this call back, because of the closure, self will
      // be accessible and refers to the Vue object.
      self.people = data.results;
    })
  }
})
```

RIGHT! Use bind.

You can bind the callback function.

```
new Vue({
  el:"#star-wars-people",
  data:{
    people: null
  },
  mounted:function(){
    $.getJSON("http://swapi.co/api/people/", function(data){
      this.people = data.results;
    }.bind(this));
  }
})
```

RIGHT! Use an arrow function.

```
new Vue({
  el:"#star-wars-people",
  data:{
    people: null
  },
  mounted: function(){
    $.getJSON("http://swapi.co/api/people/", data => this.people = data.results);
  }
})
```

Caution! Arrow functions are a syntax introduced in Ecmascript 2015. It is not yet supported but *all* modern browsers, so only use it if you are targetting a browser you *know* supports it, or if you are compiling your javascript down to ES5 syntax using something like babel.

WRONG! Using an arrow function to define a method that refers to "this"

```
new Vue({
  el:"#app",
  data:{
    foo: "bar"
  },
  methods:{
    // This is wrong! Arrow functions capture "this" lexically
```

```
    // and "this" will refer to the window.
    doSomething: () => this.foo = "baz"
  }
})
```

RIGHT! Define methods with the typical function syntax

```
new Vue({
  el:"#app",
  data:{
    foo: "bar"
  },
  methods:{
    doSomething: function(){
      this.foo = "baz"
    }
  }
})
```

Alternatively, if you are using a javascript compiler or a browser that supports Ecmascript 2015

```
new Vue({
  el:"#app",
  data:{
    foo: "bar"
  },
  methods:{
    doSomething(){
      this.foo = "baz"
    }
  }
})
```

Read Using "this" in Vue online: https://riptutorial.com/vue-js/topic/9350/using--this--in-vue

www.dbooks.org

Chapter 22: Vue single file components

Introduction

Describe how to create single file components in a .vue file.

Specially the design decisions that can be made.

Examples

Sample .vue component file

```
<template>
  <div class="nice">Component {{title}}</div>
</template>

<script>
export default {
    data() {
        return {
            title: "awesome!"
        };
    }
}
</script>

<style>
.nice {
    background-color: red;
    font-size: 48px;
}
</style>
```

Read Vue single file components online: https://riptutorial.com/vue-js/topic/10118/vue-single-file-components

Chapter 23: VueJS + Redux with Vua-Redux (Best Solution)

Examples

How to use Vua-Redux

Installing Vua Redux from NPM:

Install through:

```
npm i vua-redux --save
```

Initialize:

==========

// main.js

```
import Vue from 'vue';
import { reduxStorePlugin } from 'vua-redux';
import AppStore from './AppStore';
import App from './Component/App';

// install vua-redux
Vue.use(reduxStorePlugin);

new Vue({
    store: AppStore,
    render(h) {
        return <App />
    }
});
```

// AppStore.js

```
import { createStore } from 'redux';

const initialState = {
  todos: []
};

const reducer = (state = initialState, action) => {
  switch(action.type){
    case 'ADD_TODO':
      return {
        ...state,
        todos: [...state.todos, action.data.todo]
      }
```


www.dbooks.org

```
    default:
      return state;
    }
}

const AppStore = createStore(reducer);

export default AppStore;
```

Use in your component :

// components/App.js

```
import { connect } from 'vua-redux';

const App = {
    props: ['some-prop', 'another-prop'],

    /**
     * everything you do with vue component props
     * you can do inside collect key
     */
    collect: {
        todos: {
            type: Array,
        },
        addTodo: {
            type: Function,
        },
    },

    methods: {
        handleAddTodo() {
            const todo = this.$refs.input.value;
            this.addTodo(todo);
        }
    },

    render(h) {
        return <div>
            <ul>
                {this.todos.map(todo => <li>{todo}</li>)}
            </ul>

            <div>
                <input type="text" ref="input" />
                <button on-click={this.handleAddTodo}>add todo</button>
            </div>
        </div>
    }
};

function mapStateAsProps(state) {
    return {
        todos: state.todos
    };
}

function mapActionsAsProps(dispatch) {
    return {
```

```
        addTodo(todo) {
            dispatch({
                type: 'ADD_TODO',
                data: { todo }
            })
        }
    }
}

export default connect(mapStateAsProps, mapActionsAsProps)(App);
```

Read VueJS + Redux with Vua-Redux (Best Solution) online: https://riptutorial.com/vue-js/topic/7396/vuejs-plus-redux-with-vua-redux--best-solution-

www.dbooks.org

Chapter 24: vue-router

Introduction

vue-router is the officially supported routing library for vue.js.

Syntax

- `<router-link to="/path">Link Text</router-link> <!-- Creates a link to the route that matches the path -->`
- `<router-view></router-view> <!-- Outlet for the currently matched route. It's component will be rendered here. -->`

Examples

Basic Routing

The easiest way to get up and running with vue-router is to use the version provided via CDN.

HTML:

```
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

<div id="router-example">
    <router-link to="/foo">Link to Foo route</router-link>
    <router-view></router-view>
</div>
```

JavaScript (ES2015):

```
const Foo = { template: <div>This is the component for the Foo route</div> }

const router = new VueRouter({
    routes: [
       { path: '/foo', component: Foo}
    ]
})

const routerExample = new Vue({
    router
}).$mount('#router-example')
```

Read vue-router online: https://riptutorial.com/vue-js/topic/9654/vue-router

Chapter 25: Vuex

Introduction

Vuex is a state management pattern + library for Vue.js applications. It serves as a centralised store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion. It also integrates with Vue's official dev tools extension to provide advanced features such as zero-config time-travel debugging and state snapshot export/import.

Examples

What is Vuex?

Vuex is an official plugin for Vue.js which offers a centralised datastore for use within your application. It is heavily influenced by the Flux application architecture which features a unidirectional data flow leading to simpler application design and reasoning.

Within a Vuex application the datastore holds all **shared application state**. This state is altered by **mutations** which are performed in response to an **action** invoking a mutation event via the **dispatcher**.

An example of the data flow in a Vuex application is outlined in the diagram below.

www.dbooks.org

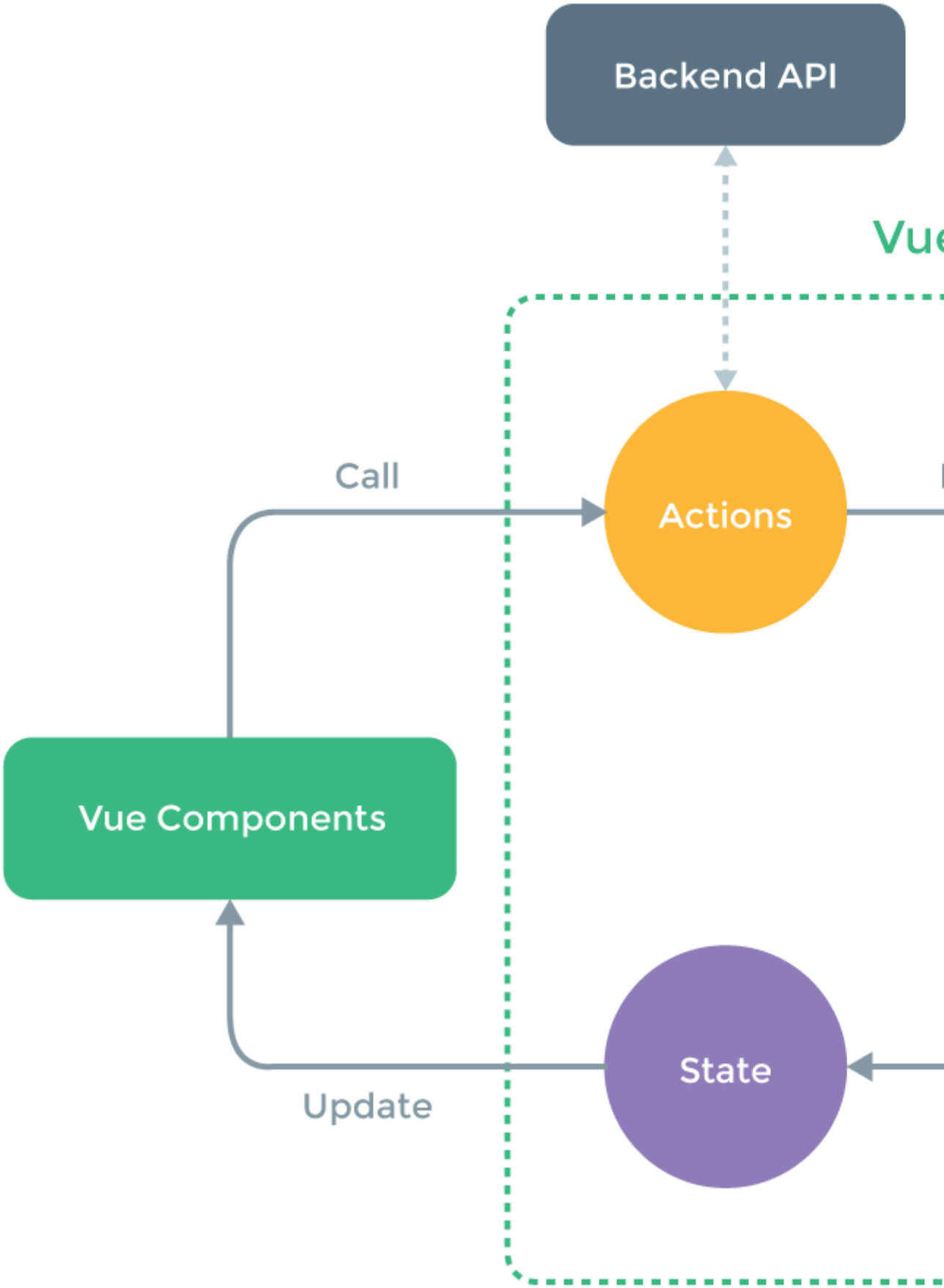

Backend API
Vue
Call
Actions
Vue Components
State
Update

Diagram used under the MIT licence, originally from the Official Vuex GitHub repo.

Individual Vue.js application components can access the store object to retrieve data via **getters**, which are pure functions returning a read-only copy of the desired data.

Components can have **actions** which are functions that perform changes to the component's own copy of the data, then use the **dispatcher** to dispatch a mutation event. This event is then handled by the datastore which updates the state as necessary.

Changes are then automatically reflected throughout the application since all components are reactively bound to the store via their getters.

An example illustrating the use of vuex in a vue project.

```
const state = {
    lastClickTime: null
}

const mutations = {
    updateLastClickTime: (state, payload) => {
      state.lastClickTime = payload
  }
}

const getters = {
  getLastClickTime: state => {
    return new Date(state.lastClickTime)
  }
}

const actions = {
    syncUpdateTime: ({ commit }, payload) => {
    commit("updateLastClickTime", payload)
  },
  asyncUpdateTime: ({ commit }, payload) => {
    setTimeout(() => {
      commit("updateLastClickTime", payload)
    }, Math.random() * 5000)
  }
}

const store = new Vuex.Store({
  state,
  getters,
  mutations,
  actions
})

const { mapActions, mapGetters } = Vuex;

// Vue
const vm = new Vue({
    el: '#container',
  store,
  computed: {
      ...mapGetters([
        'getLastClickTime'
```


www.dbooks.org

```
    ])
  },
  methods: {
      ...mapActions([
        'syncUpdateTime',
      'asyncUpdateTime'
    ]),
    updateTimeSyncTest () {
        this.syncUpdateTime(Date.now())
    },
    updateTimeAsyncTest () {
        this.asyncUpdateTime(Date.now())
    }
  }
})
```

And the HTML template for the same:

```
<div id="container">
  <p>{{ getLastClickTime || "No time selected yet" }}</p>
  <button @click="updateTimeSyncTest">Sync Action test</button>
  <button @click="updateTimeAsyncTest">Async Action test</button>
</div>
```

1. Here the **state** contains **lastClickTime** property initialized as null. This setting up of default values is important to keep the properties **reactive**. **Properties not mentioned in the state** will be available but the changes made thereafter **would not be accessible** by using getters.

2. The getter used, provides a computed property which will be updated each time a mutation updates the value of the state property.

3. **Only mutations** are allowed to change the state and its properties, that said, it does so **synchronously only**.

4. An Action can be used in case of asynchronous updates, where the API call (here mocked by the randomly timed setTimeout) can be made in the action, and after getting the response a mutation can be committed to, to make the change to the state.

Why use Vuex?

When building large applications such as Single Page Apps (SPA's), which typically consist of many reusable components they can quickly become difficult to build and maintain. The sharing of data and state between these components can also quickly break down and become difficult to debug and maintain.

By using a centralised application data store the entire application state can be represented in one place making the application more organised. Through the use of a unidirectional data flow, mutations and by scoping component data access to only the data required it becomes much simpler to reason about the component role and how it should affect the application state.

VueJS components are separate entities and they cannot share data between each other easily.

To share data without vuex we need to `emit` event with data and then listen and catch that event with `on`.

component 1

```
this.$emit('eventWithDataObject', dataObject)
```

component 2

```
this.$on('eventWithDataObject', function (dataObject) {
    console.log(dataObject)
})
```

With vuex installed we can simply access its data from any component without a need of listening to events.

```
this.$store.state.myData
```

We can also change data synchronously with *mutators*, use asynchronous *actions* and get data with *getter* functions.

Getter functions might work as global computed functions. We can access them from components:

```
this.$store.getters.myGetter
```

Actions are global methods. We can dispatch them from components:

```
this.$store.dispatch('myAction', myDataObject)
```

And mutations are the only way to change data in vuex.We can commit changes:

```
this.$store.commit('myMutation', myDataObject)
```

Vuex code would look like this

```
state: {
  myData: {
    key: 'val'
  }
},
getters: {
  myGetter: state => {
     return state.myData.key.length
  }
},
actions: {
  myAction ({ commit }, myDataObject) {
    setTimeout(() => {
      commit('myMutation', myDataObject)
    }, 2000)
  }
```


www.dbooks.org

```
},
mutations: {
  myMutation (state, myDataObject) {
    state.myData = myDataObject
  }
}
```

How to install Vuex?

Most of the time that you'll be using Vuex will be in larger component based applications where you likely be using a module bundler such as Webpack or Browserify in conjunction with Vueify if you're using single files.

In this case the easiest way to get Vuex is from NPM. Run the command below to install Vuex and save it to your application dependencies.

```
 npm install --save vuex
```

Ensure that you load link Vuex with your Vue setup by placing the following line after your `require('vue')` statement.

```
Vue.use(require('vuex'))
```

Vuex is also available on CDN; you can grab the latest version from cdnjs here.

Auto dismissible notifications

This example will register an vuex module dynamically for storing custom notifications that can automatically dismissed

notifications.js

resolve vuex store and define some constants

```
//Vuex store previously configured on other side
import _store from 'path/to/store';

//Notification default duration in milliseconds
const defaultDuration = 8000;

//Valid mutation names
const NOTIFICATION_ADDED = 'NOTIFICATION_ADDED';
const NOTIFICATION_DISMISSED = 'NOTIFICATION_DISMISSED';
```

set our module initial state

```
const state = {
  Notifications: []
}
```

set our module getters

```
const getters = {
  //All notifications, we are returning only the raw notification objects
  Notifications: state => state.Notifications.map(n => n.Raw)
}
```

set our module Actions

```
const actions = {
  //On actions we receive a context object which exposes the
  //same set of methods/properties on the store instance
  //{commit} is a shorthand for context.commit, this is an
  //ES2015 feature called argument destructuring
  Add({ commit }, notification) {
    //Get notification duration or use default duration
    let duration = notification.duration || defaultDuration

    //Create a timeout to dismiss notification
    var timeOut = setTimeout(function () {
      //On timeout mutate state to dismiss notification
      commit(NOTIFICATION_DISMISSED, notification);
    }, duration);

    //Mutate state to add new notification, we create a new object
    //for save original raw notification object and timeout reference
    commit(NOTIFICATION_ADDED, {
      Raw: notification,
      TimeOut: timeOut
    })
  },
  //Here we are using context object directly
  Dismiss(context, notification) {
    //Just pass payload
    context.commit(NOTIFICATION_DISMISSED, notification);
  }
}
```

set our module mutations

```
const mutations = {
  //On mutations we receive current state and a payload
  [NOTIFICATION_ADDED](state, notification) {
    state.Notifications.push(notification);
  },
  //remember, current state and payload
  [NOTIFICATION_DISMISSED](state, rawNotification) {
    var i = state.Notifications.map(n => n.Raw).indexOf(rawNotification);
    if (i == -1) {
      return;
    }

    clearTimeout(state.Notifications[i].TimeOut);
    state.Notifications.splice(i, 1);
  }
}
```

Register our module with defined state, getters, actions and mutation

www.dbooks.org

```
_store.registerModule('notifications', {
  state,
  getters,
  actions,
  mutations
});
```

Usage

componentA.vue

This components displays all notifications as bootstrap's alerts on top right corner of screen, also allows to manually dismiss each notification.

```
<template>
<transition-group name="notification-list" tag="div" class="top-right">
  <div v-for="alert in alerts" v-bind:key="alert" class="notification alert alert-dismissible"
v-bind:class="'alert-'+alert.type">
    <button v-on:click="dismiss(alert)" type="button" class="close" aria-label="Close"><span
aria-hidden="true">&times;</span></button>
    <div>
      <div>
        <strong>{{alert.title}}</strong>
      </div>
      <div>
        {{alert.text}}
      </div>
    </div>
  </div>
</transition-group>
</template>

<script>
export default {
  name: 'arc-notifications',
  computed: {
    alerts() {
      //Get all notifications from store
      return this.$store.getters.Notifications;
    }
  },
  methods: {
    //Manually dismiss a notification
    dismiss(alert) {
      this.$store.dispatch('Dismiss', alert);
    }
  }
}
</script>
<style lang="scss" scoped>
$margin: 15px;

.top-right {
    top: $margin;
    right: $margin;
    left: auto;
    width: 300px;
    //height: 600px;
    position: absolute;
```

```
    opacity: 0.95;
    z-index: 100;
    display: flex;
    flex-wrap: wrap;
    //background-color: red;
}
.notification {
    transition: all 0.8s;
    display: flex;
    width: 100%;
    position: relative;
    margin-bottom: 10px;
    .close {
        position: absolute;
        right: 10px;
        top: 5px;
    }

    > div {
        position: relative;
        display: inline;
    }
}
.notification:last-child {
    margin-bottom: 0;
}
.notification-list-enter,
.notification-list-leave-active {
    opacity: 0;
    transform: translateX(-90px);
}
.notification-list-leave-active {
    position: absolute;
}
</style>
```

Snippet for add notification in any other component

```
//payload could be anything, this example content matches with componentA.vue
this.$store.dispatch('Add', {
  title = 'Hello',
  text = 'World',
  type = 'info',
  duration = 15000
});
```

Read Vuex online: https://riptutorial.com/vue-js/topic/3430/vuex

www.dbooks.org

Chapter 26: Watchers

Examples

How it works

You can watch data property of any Vue instance. When watching a property, you trigger a method on change:

```
export default {
    data () {
        return {
            watched: 'Hello World'
        }
    },
    watch: {
        'watched' () {
            console.log('The watched property has changed')
        }
    }
}
```

You can retrieve the old value and the new one:

```
export default {
    data () {
        return {
            watched: 'Hello World'
        }
    },
    watch: {
        'watched' (value, oldValue) {
            console.log(oldValue) // Hello World
            console.log(value) // ByeBye World
        }
    },
    mounted () {
        this.watched = 'ByeBye World'
    }
}
```

If you need to watch nested properties on an object, you will need to use the `deep` property:

```
export default {
    data () {
        return {
            someObject: {
                message: 'Hello World'
            }
        }
    },
    watch: {
        'someObject': {
```

```
            deep: true,
            handler (value, oldValue) {
                console.log('Something changed in someObject')
            }
        }
    }
}
```

When is the data updated?

If you need to trigger the watcher before making some new changes to an object, you need to use the `nextTick()` method:

```
export default {
    data() {
        return {
            foo: 'bar',
            message: 'from data'
        }
    },
    methods: {
        action () {
            this.foo = 'changed'
            // If you juste this.message = 'from method' here, the watcher is executed after.
            this.$nextTick(() => {
                this.message = 'from method'
            })
        }
    },
    watch: {
        foo () {
            this.message = 'from watcher'
        }
    }
}
```

Read Watchers online: https://riptutorial.com/vue-js/topic/7988/watchers

www.dbooks.org

Credits

S. No	Chapters	Contributors
1	Getting started with Vue.js	Community, Erick Petrucelli, ironcladgeek, J. Bruni, James, Lambda Ninja, m_callens, MotKohn, rap-2-h, Ru Chern Chong, Sankalp Singha, Shog9, Shuvo Habib, user1012181, user6939352, Yerko Palma
2	Components	Donkarnash, Elfayer, Hector Lorenzo, Jeff, m_callens, phaberest, RedRiderX, user6939352
3	Computed Properties	Amresh Venugopal, cl3m, jaredsk, m_callens, Theo, Yerko Palma
4	Conditional Rendering	jaredsk, m_callens, Nirazul, user6939352
5	Custom Components with v-model	Amresh Venugopal
6	Custom Directives	Mat J, Ogie Sado
7	Custom Filters	Finrod, M U, m_callens
8	Data Binding	gurghet, Jilson Thomas
9	Dynamic Components	Med, Ru Chern Chong
10	Event Bus	Amresh Venugopal
11	Events	Elfayer
12	Lifecycle Hooks	Linus Borg, m_callens, PatrickSteele, xtreak
13	List Rendering	chuanxd, gurghet, Mahmoud, Theo
14	Mixins	Ogie Sado
15	Modifiers	sept08
16	Plugins	AldoRomo88
17	Polyfill "webpack" template	Stefano Nepa

18	Props	asemahle, Donkarnash, FlatLander, m_callens, rap-2-h, Shuvo Habib
19	Slots	Daniel Waghorn, Elfayer, Shuvo Habib, Slava
20	The array change detection caveats	Vamsi Krishna
21	Using "this" in Vue	Bert
22	Vue single file components	jordiburgos, Ru Chern Chong
23	VueJS + Redux with Vua-Redux (Best Solution)	Aniko Litvanyi, FlatLander, Shuvo Habib, Stefano Nepa
24	vue-router	AJ Gregory
25	Vuex	AldoRomo88, Amresh Venugopal, Daniel Waghorn, Matej Vrzala M4, Ru Chern Chong
26	Watchers	El_Matella

www.dbooks.org