

Vue.js

Logistic

- Hours and Scheduling
- Lunch & breaks

Overview

- What is Vue.js
- The Vue instance
- Components
- Directives
- Filters
- Routing
- Data fetching
- Vue-CLI
- Plugins
- Typescript
- Vuex
- Testing
- Server Side rendering





What is Vue.js

Overview

- *What is Vue.js*
- The Vue instance
- Components
- Directives
- Filters
- Routing
- Data fetching
- Vue-CLI
- Plugins
- Typescript
- Vuex
- Testing
- Server Side rendering

Introduction



It has been created in February 2014 by Evan You. It is maintained by himself & his team, not by a GAFAM.

Version 2 has been announced in April 2016 & released in December.
We are currently into the **2.5.21** version since December 2016.

Introduction



Vuejs is 3th most starred project on **Github**. GitHub is a web-based hosting service for version control using git & commonly used to host open-source software projects.

<https://github.com>

Installation

Vuejs can be installed on your computer via **NPM**. NPM is a package manager for Node.js. All packages are present on npmjs.org



... Or you can add the library directly via **CDN** on your project via script tag.



PW1

Inspiration

Vuejs is used to write sophisticated Single-Page Applications. Also it is designed to be incrementally adoptable with other Javascript Libraries.

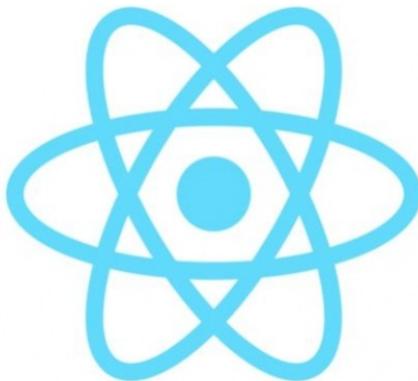
By the way, a lot of concepts of Vuejs are directly inspired by other great libraries & frameworks such as *AngularJS*, *React* or even *jQuery*.

AngularJs shared concepts



- Syntax (v-if vs ng-if)
- Two-way binding
- Directives
- Watchers
- Filters

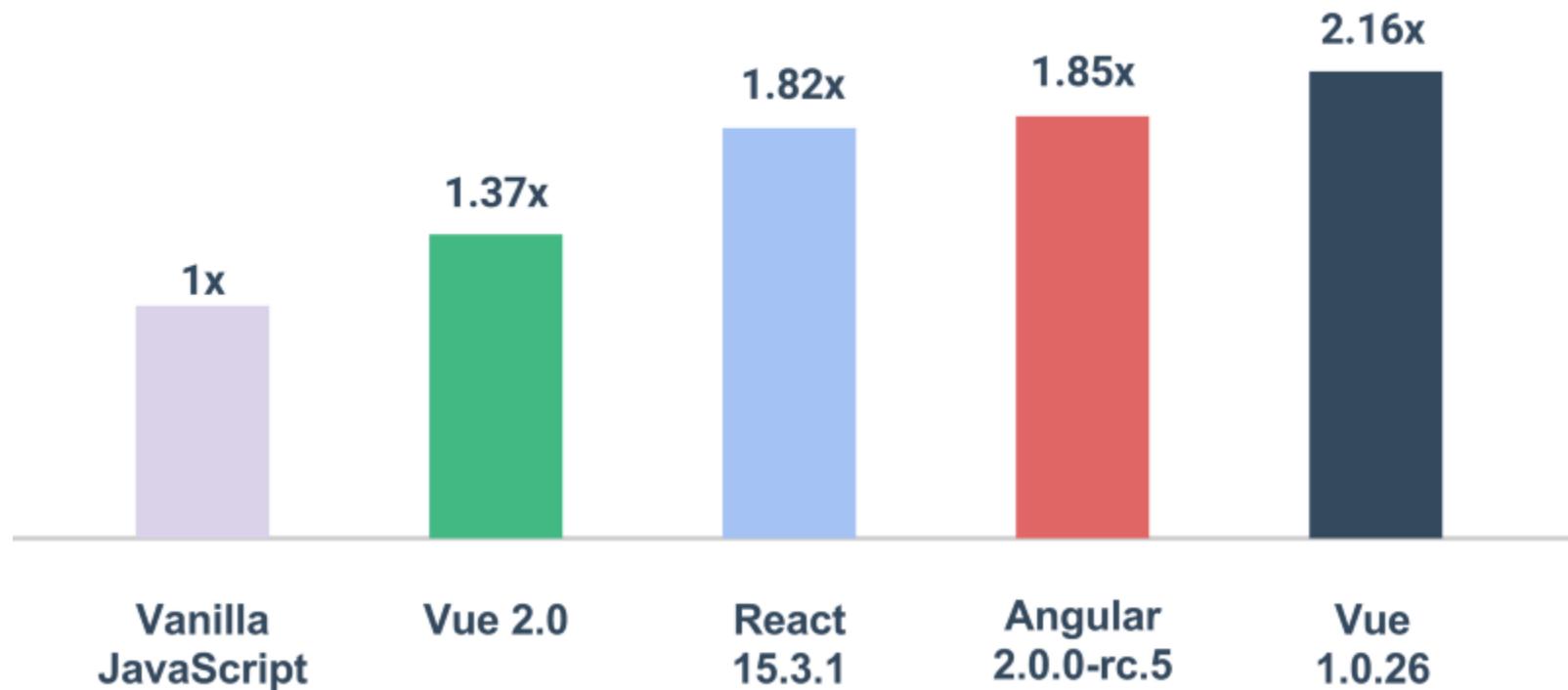
React shared concepts



- Virtual DOM
- Render Functions & JSX Support
- Focus on the View Layer

Benchmark

According to many benchmarks, Vuejs is currently the fastest frameworks between the bigs one.





The Vue instance

Overview

- What is Vue.js
- *The Vue instance*
- Components
- Directives
- Filters
- Routing
- Data fetching
- Vue-CLI
- Plugins
- Typescript
- Vuex
- Testing
- Server Side rendering

Our first instance

A Vue application consists of a root Vue instance created with new Vue, optionally organized into a tree of nested, reusable components. Vue instance is declared with Vue constructor function.

```
let vm = new Vue({  
    // options here  
})
```

- Each instance is declared with an options object which can contain data, methods, element to mount on or lifecycle callbacks.

Mount a DOM element

- We use **el** option to mount an instance to a DOM element.

```
<div id="app">
  <h1>Hello World!</h1>
</div>
```

```
let vm = new Vue({
  el: '#app',
})
```

We can also use the `$mount()` method instead of `el` option on your `Vue` instance.

```
let vm = new Vue({
}).$mount('#app')
```

Data properties

- Each properties found in data object are 'proxified'
- Vue attachs setters & getters to each properties
- All properties are accessible on client side

```
<div id="app">
  <h1>{{message}}</h1>
</div>
```

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

Lifecycle hooks

- Lifecycle hooks help us to manage stages of a Vue instance's lifecycle.
- We will be able to execute code at a precise moment of our instance.

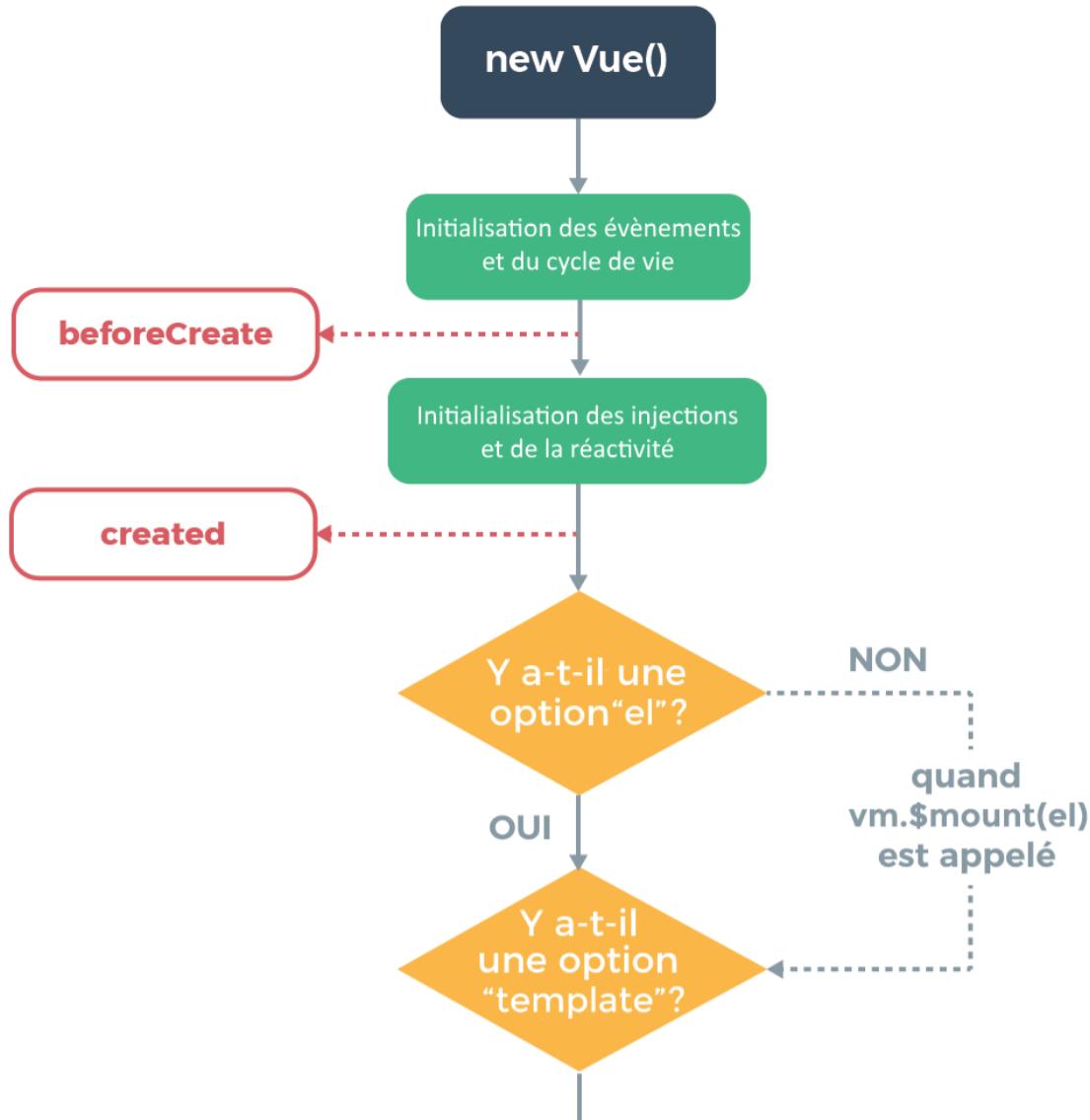
```
let vm = new Vue({  
  el: '#app',  
  data: {  
    message: 'Hello World!'  
  },  
  beforeCreate: function() {  
    console.log("I'm beforeCreate");  
  },  
  created: function() {  
    console.log("I'm created");  
  },  
  beforeMount: function() {  
    console.log("I'm beforeMount");  
  },  
  ...  
})
```

Lifecycle hooks 2

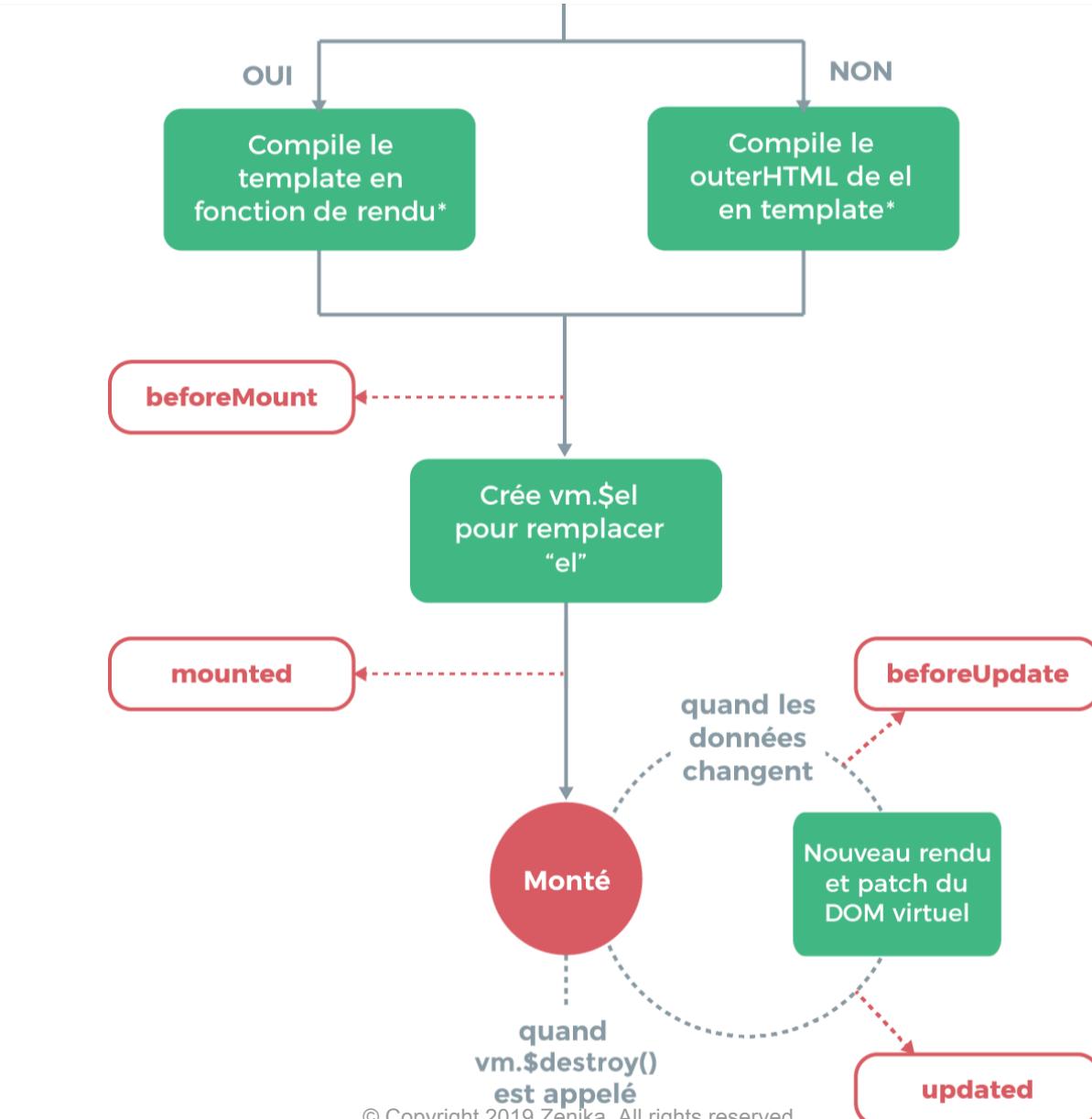
```
...
  mounted: function() {
    console.log("I'm mounted");
  },
  beforeDestroy: function() {
    console.log("I'm beforeDestroy");
  },
  destroyed: function() {
    console.log("I'm destroyed");
  },
});
```



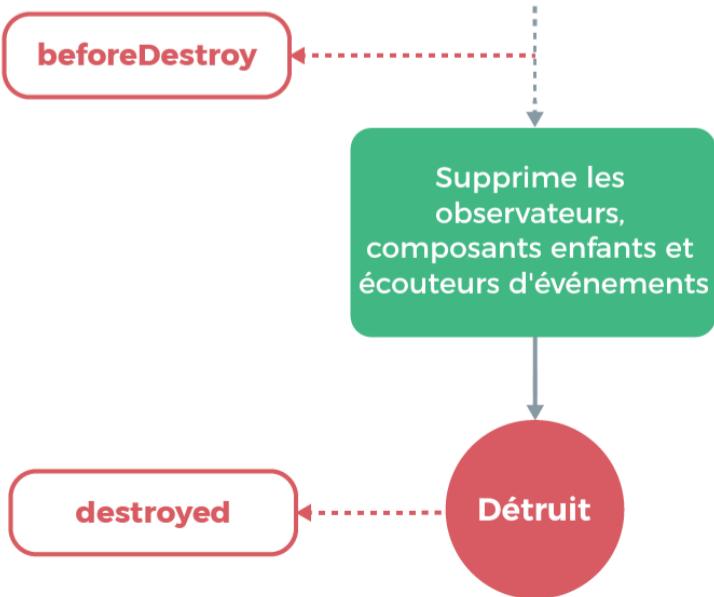
Lifecycle Diagram



Lifecycle Diagram part 2.



Lifecycle Diagram part 3.





PW2



Components

Overview

- What is Vue.js
- The Vue instance
- *Components*
- Directives
- Filters
- Routing
- Data fetching
- Vue-CLI
- Plugins
- Typescript
- Vuex
- Testing
- Server Side rendering

Overview

- Components are custom HTML elements.
- They allow to encapsulate reusable code.
- Each component should have its own behavior.

Our first component

- We use **Vue.component(tagName, options)** to register a component.

```
Vue.component('my-component', {  
  // options  
})  
let vm = new Vue({  
  el: '#app',  
})
```

tagName must be all lowercase & contains a hyphen

- Then we can use it as a custom element in an instance's template

```
<div id="app">  
  <my-component></my-component>  
</div>
```

Template

- To pass HTML element as a template to our component, we use **template** option element.

```
Vue.component('my-component', {  
  template: '<h1>My first component</h1>',  
})  
let vm = new Vue({  
  el: '#app',  
})
```

```
<div id="app">  
  <my-component></my-component>  
</div>
```

- This example above will render :

```
<div id="app">  
  <h1>My first component</h1>  
</div>
```

Data

- Unlike Vue instance, component data return a function. This avoid issues that can appear when many instances are created using the same definition. Every time a new instance is created, we can simply call it to return a fresh copy of the initial data.
- Data can be called in component template.

```
Vue.component('my-component', {  
  template: '<h1>{{message}}</h1>',  
  data() {  
    return {  
      message: 'Hello World!'  
    }  
  },  
})
```

Props

- Data can be passed down to child components using props.
- Props can be used inside templates.
- Props must be explicitly declared with **props** option element.

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

```
<child-component message="foo"></child-component>
```

Methods

- Methods are assigned functions of our component.
- They can only be used in the component's scope.

```
Vue.component('child-component', {  
  template: '<button @click="addLike()">{{likes}}</button>',  
  data() {  
    return {  
      likes: 0,  
    }  
  },  
  methods: {  
    addLike () {  
      this.likes++;  
    },  
  },  
})
```

Computed properties

- Computed properties are used to avoid complex expressions in templates
- We use **computed** option element to use it

```
Vue.component('child-component', {  
  template: '<span>{{fullName}}</span>',  
  data() {  
    return {  
      firstName: 'John',  
      lastName: 'Doe',  
    }  
  },  
  computed: {  
    fullName () {  
      return `${this.firstName} ${this.lastName}`;  
    },  
  },  
})
```

Note that a computed property will only re-evaluate when some of its dependencies have changed.



PW3

Mixins

Mixins allow us to distribute functionalities across many components. When a component uses a mixin, all options in the mixin will be “mixed” into the component’s own options.

Quick example:

```
var myMixin = {
  created: function () {
    this.greeting()
  },
  methods: {
    greeting: function () {
      alert('Hello!')
    }
  },
}

var Component = Vue.extend({
  mixins: [myMixin],
})

var component = new Component() // -> "Hello!"
```

Mixins

Mixin hooks will always be called before the component's own hooks, but component options will take priority when there are conflicting keys in these objects.

```
var myMixin = {
  methods: {
    greeting: function () {
      console.log('Hello!')
    },
    conflicting: function () {
      console.log('Hello from mixin!')
    }
  },
}
var vm = Vue.component({
  mixins: [myMixin],
  methods: {
    conflicting: function () {
      console.log('Hello from component!')
    }
  },
})
vm.greeting() // -> "Hello!"
vm.conflicting() // -> "Hello from component!"
```

Slots

Sometimes, you need to give more than props to a component

```
<!-- HTML of Parent component -->
<my-dialog>
  <div>This is the Dialog Content</div>
</my-dialog>
```

```
<!-- HTML of child component my-dialog-->
<section class="dialog">
  <h1>This is a dialog title</h1>
  <div class="DialogContent">
    <slot></slot>
  </div>
</section>
```

It will produce:

```
<!-- HTML of child component my-dialog-->
<section class="dialog">
  <h1>This is a dialog title</h1>
  <div class="DialogContent">
    <div>This is the Dialog Content</div>
  </div>
</section>
```

Render functions

Most of the time Vue templates can accomplish almost everything we need to build our HTML. But sometimes we will base our component creation on input or slot values. In these cases we need to use **Render functions**.

- Components with render functions do not have a template tag or property.
- These components use a **render** function that receive a createElement argument (commonly aliased as h) and return an element created with that function.

Render functions

Quick example :

```
<rendered-component :tag="span">Hello world!</rendered-component>
```

```
Vue.component('rendered-component', {
  render: function (createElement) {
    return createElement(
      this.tag, // tag attribute, 'span' in our example
      this.$slots.default // Array of children, 'Hello world!' in this case
    )
  },
  props: {
    tag: {
      type: String,
      required: true
    }
  }
})
```

... Will render

```
<span>Hello World!</span>
```

Render functions

createElement() receive as argument (renderElement: String | Object | Function, definition: Object, children: String | Array)

```
// @returns {VNode}
createElement(
  // {String | Object | Function}
  'div',
  // {Object}
  {
    'class': 'example'
  },
  // {String | Array}
  // Children VNodes, built using `createElement()`,
  // or simply using strings to get 'text VNodes'. Optional.
  [
    'Some text comes first.',
    createElement('h1', 'A headline'),
    createElement(MyComponent, {
      props: {
        someProp: 'foobar'
      }
    })
  ]
)
```

Render functions

Note that Render functions do not have access to Vue template features such as 'v-if' or 'v-for'. They must be implemented in plain Javascript.

For the v-for directive, it could be something like this:

```
render (h) {
  return h('ul', this.elements.map(element => h('li')));
}
```

Instead of

```
<template>
  <ul>
    <li v-for="element of elements">
    </li>
  </ul>
</template>
```

JSX

If you're writing a lot of render functions, syntax can be painful, so that's why there is Babel plugin that allows us to use JSX with Vue.

Example :

```
export default {
  render (h) {
    return (
      <p>Hello World!.</p>
    )
  }
}
```



Directives

Overview

- What is Vue.js
- The Vue instance
- Components
- *Directives*
- Filters
- Routing
- Data fetching
- Vue-CLI
- Plugins
- Typescript
- Vuex
- Testing
- Server Side rendering

Basics

v-directive = “js expression”

```
<span v-if="displayMessage"> Now you see me </span>
```

A directive allows Vue to add a side-effect to the DOM

- Starts with a v-
- The value is a javascript expression

Arguments

v-directive: arguments

```

```

Pass arguments to the directive

- Denoted by a : after the directive name

Modifiers

v-directive.modifier

```
<button v-on:click.prevent.stop="clickButtonCallback()">
```

Binds the directive to a certain way

- Denoted by a `.` after the directive name

Conditional Rendering

v-if / v-else / v-else-if / v-show

- Built-in directives used to add or remove DOM element

```
<div v-if="test">Test</div>
```

```
<template v-if="test">
  <h1>Title</h1>
  <h2>Subtitle</h2>
</template>
```

```
<div v-if="done">done</div>
<div v-else-if="loading">loading...</div>
<div v-else>Nothing to display</div>
```

```
<dialog v-show="toast"></dialog>
```

v-if / v-else / v-else-if / v-show

- Data object

```
{  
  display: false,  
  userStatus: 'loggedOut'  
}
```

- Template

```
<div class="alert" v-show="display">Alert displayed</div>  
<template v-if="true">  
  <h1>Title</h1>  
  <h2>Subtitle</h2>  
<button v-if="userStatus === 'loggedIn'">Logout</button>  
<button v-else>Login</button>
```



What will it render ?

v-if / v-else / v-else-if / v-show

- rendered DOM:

```
<div class="alert" style="display: none;">Alert displayed</div>
<h1>Title</h1>
<h2>Subtitle</h2>
<button>Login</button>
```

List Rendering

v-for

- Built in directives used to render a list

```
v-for = "item in array"
```

```
v-for = "(item, index) in items"
```

```
v-for = "(value, key, index) in object"
```

Example:

```
<ul>
  <li v-for="route in routes">{{route.name}}</li>
</ul>
```

- With component you must add a :key attribute

```
<card v-for="element in menu" :key="element.id"></card>
```

v-for

- Data object

```
{  
  users: [  
    { name: 'Evan You', username: '@yyx990803' },  
    { name: 'Edd Yerburgh', username: '@eddyerburgh' }  
,  
    auth: { type: 'jwt', exp: 1501585988266 }  
}
```

- Template

```
<span v-for="n in 5">{{ n }} </span>  
<template v-for="user in users">  
  <span>{{user.name}}</span>  
  <span>{{user.username}}</span>  
</template>  
<div v-for="(key, value, index) in auth">{{index}}# {{key}}: {{value}}</div>
```



What will it render ?

v-for

- rendered DOM:

```
<span>1</span>
<span>2</span>
<span>3</span>
<span>4</span>
<span>5</span>

<span>Evan You</span>
<span>@yyx990803</span>
<span>Edd Yerburgh</span>
<span>@eddyerburgh</span>

<div>0# type: jwt</div>
<div>1# exp: 1501585988266</div>
```

Class and Style bindings

v-bind

- Built-in directive used to dynamically bind an HTML attribute to a JS expression

```
data () {  
  return {  
    dynamicId: 28  
  }  
}
```

```
<div v-bind:id="dynamicId"></div>
```

Render:

```
<div id="28"></div>
```

v-bind:class

Special feature with :class attributes

- Object

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



What will it render ?

Render:

- isActive === true

```
<div class="link active-link"></div>
```

- isActive === false

```
<div class="link"></div>
```

v-bind:class

- Array

```
<div class="alert" v-bind:class="[alertLevel, position]"></div>
```

```
data () {
  return {
    alertLevel: 'info',
    position: 'top'
  }
}
```



What will it render ?

Render:

```
<div class="alert info top"></div>
```

v-bind:style

Special feature with :style attributes

```
<div class="alert" v-bind:style="{ 'background-color': alertLevel }"></div>
```

```
data () {
  return {
    alertLevel: '#00FF00'
  }
}
```



What will it render ?

Render:

```
<div class="alert" style="background-color: #00FF00"></div>
```

Event Handling

v-on

- Built-in directive used to listen to DOM or component event

```
<element v-on:eventName="codeToRunOnEvent"></element>
```

Example:

```
<button v-on:click="alert('login')">Login</button>
```

v-on - Event Modifiers

- `.stop`: `event.stopPropagation()`
- `.prevent`: `event.preventDefault()`
- `.capture`: uses capture mode when adding the event listener
- `.self`: only from the first component (ignore from nested component)
- `.once`: the event is captured only once

```
<a href="" v-on:click.stop.prevent.capture.self.once="alert('link')">Link</a>
```

v-on - Key modifier

- `.enter`: captures enter key
- `.tab`: captures tab key
- `.delete`: captures both “Delete” and “Backspace” keys
- `.esc`: captures escape key
- `.space`: captures space key
- `.up`: captures arrow up key
- `.down`: captures arrow down key
- `.left`: captures arrow left key
- `.right`: captures arrow right key
- `.<keyCode>`: captures the

Shorthand

v-bind

- `v-bind:attr` can be shortened into `:attr`
- The following

```
<a v-bind:id="user.id" v-bind:href="user.loginUrl" v-bind:class="{ 'active-link': isActive}">Link</a>
```

- Is the same as

```
<a :id="user.id" :href="user.loginUrl" :class="{ 'active-link': isActive}">Link</a>
```

v-on

- `v-on:event` can be shortened into `@event`
- The following

```
<input v-on:keyup.13="login()" v-on:click="focus()"></input>
```

- Is the same as

```
<input @keyup.13="login()" @click="focus()"></input>
```

Form Input bindings

Naive implementation

- Using `v-bind` and `v-on`

```
<input type="text" :value="text" @input="text = $event.target.value">
```

👍 Works well with input text or custom component

👎 Limited when using radio, select or multi-checkbox

v-model

- Built-in directive used to provide "two-way" data bindings

Text

```
<input type="text" v-model="text">
```

Checkboxes

```
<input type="checkbox" v-model="selected">
```

Multi-line text

```
<textarea v-model="multiLine"></textarea>
```

v-model

Multi checkbox

```
data () {  
  return {  
    selection: ['red', 'blue']  
  }  
}
```

```
<input type="checkbox" v-model="selection" value="red">  
<input type="checkbox" v-model="selection" value="blue">  
<input type="checkbox" v-model="selection" value="green">
```

Radio

```
<input type="radio" v-model="gender" value="M">  
<input type="radio" v-model="gender" value="F">
```

v-model

Select

```
<select v-model="phoneType">
  <option disabled value="">Pick one</option>
  <option>android</option>
  <option>iOS</option>
  <option>windows phone</option>
</select>
```

v-model - Modifiers

- `.lazy`: update model on `change` event instead of `input`
- `.number`: typecaste into `Number`
- `.trim`: auto trim the value

```
<input type="text" v-model.trim.number.lazy="firstName">
```



PW4

Custom directive

Vue.directive

```
<div v-my-directive></div>
```

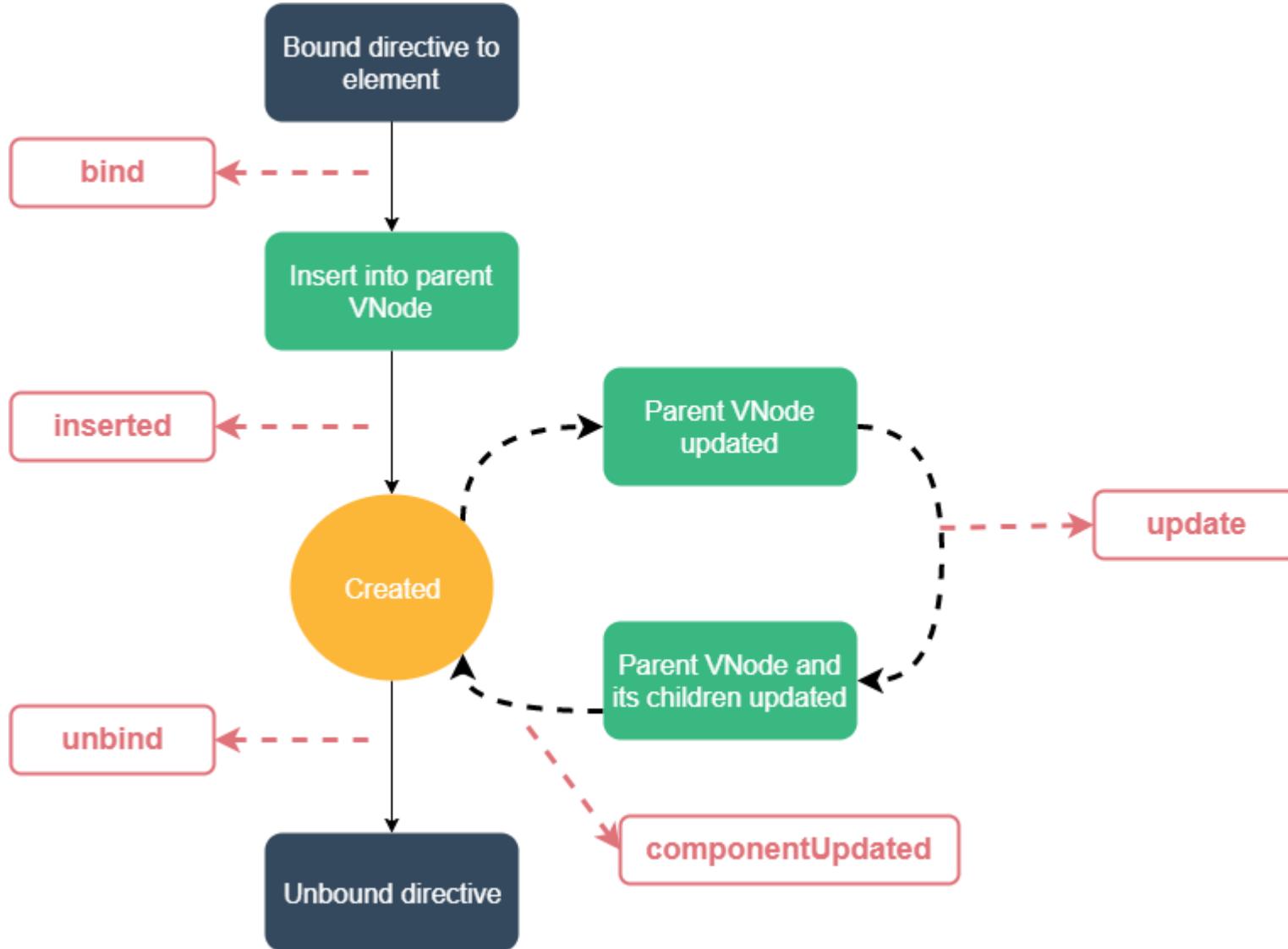
- Register it globally

```
Vue.directive('my-directive', {  
  ...  
})
```

- Register it locally

```
Vue.component('my-component', {  
  directives: {  
    myDirective: {  
      ...  
    }  
  }  
})
```

Directive Hooks



Directive Hooks Arguments

- `el`: The element the directive is bound to. This can be used to directly manipulate the DOM.
- `binding`: An object containing the following properties.
 - `name`: The name of the directive, without the v- prefix.
 - `value`: The value passed to the directive.
 - `oldValue`: The previous value, only available in update and componentUpdated. It is available whether or not the value has changed.
 - `expression`: The expression of the binding as a string.
 - `arg`: The argument passed to the directive, if any.
 - `modifiers`: An object containing modifiers, if any.



Directive Hooks Arguments

- `vnode`: The virtual node produced by Vue's compiler.
- `oldVnode`: The previous virtual node, only available in update and componentUpdated.

Function shorthand

- Many case, same behavior on `bind` and `update` hook
- Vue provides a shorthand notation for this

```
Vue.directive('my-directive', (el, binding) => {  
  ...  
})
```



PW4



Filters

Overview

- What is Vue.js
- The Vue instance
- Components
- Directives
- *Filters*
- Routing
- Data fetching
- Vue-CLI
- Plugins
- Typescript
- Vuex
- Testing
- Server Side rendering

Use a filter

- Function used to apply common text formatting
- Only available in **mustache interpolations** and **v-bind expression**

```
<p>{{ text | myFilter }}</p>
```

```
<div :id="id | myFilter">
```

- Can be chained

```
<p>{{ text | myFilter1 | myFilter2 | myFilter3 }}</p>
```

- Can have arguments

```
<p>{{ text | myFilter(arg1, arg2) }}</p>
```

Write a filter

- Register it globally

```
Vue.filter('myFilter', (value, arg1, arg2) => {  
  // Format the value  
  return valueFormatted  
})
```

- Register it locally

```
Vue.component('my-component', {  
  filters: {  
    myFilter (value, arg1, arg2) {  
      // Format the value  
      return valueFormatted  
    }  
  }  
})
```





PW5

Routing

Overview

- What is Vue.js
- The Vue instance
- Components
- Directives
- Filters
- *Routing*
- Data fetching
- Vue-CLI
- Plugins
- Typescript
- Vuex
- Testing
- Server Side rendering

Routing

Vue has an official router plugin called ***vue-router***. We can add and use it by two ways:

- Import library via CDN
- Installing it via Yarn or npm.

Basic routing

Example :

```
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }
const routes = [
  { path: '/foo', component: Foo },
  { path: '/bar', component: Bar }
]
const router = new VueRouter({
  routes
})
Vue.use(VueRouter)
const app = new Vue({
  router
}).$mount('#app')
```

```
<div id="app">
  <h1>Hello App!</h1>
  <div>
    <router-link to="/foo">Go to Foo</router-link>
    <router-link to="/bar">Go to Bar</router-link>
  </div>
  <router-view></router-view>
</div>
```

Basic routing

In this example we can notice few things :

- We define template for each routes we gonna use (Foo & Bar).
- We associate routes to each components.
- We create the router instance and pass the 'routes' option.
- We inject the router in our root instance.
- We use ***router-link*** component and we provide it a link as a prop.
- Then ***router-view*** render the component that match with our current route.

Dynamic Route Matching

Quickly, we may want to share a component for many users or products. We need to have a core path that can fit with our different IDs that represent our users or products. To do so we can use a dynamic segment in the path.

Example :

```
const User = {  
    template: '<div>User</div>'  
}  
const router = new VueRouter({  
    routes: [  
        {  
            path: '/user/:id',  
            component: User  
        }  
    ]  
})
```

By doing this, we will be able to navigate to '**'user/1337'** or '**'user/42'**' but to use the same User component.

Dynamic Route Matching

The `:id` parameter of our route is exposed in every component in every component. We can access to it easily by using `this.$route.params`.

```
const User = {
  template: `
    <div>
      User {{$route.params.id}}
    </div>
  `
}

const router = new VueRouter({
  routes: [
    {
      path: '/user/:id',
      component: User
    }
  ]
})
```

Dynamic Route Matching

Many parameters can be passed to our routes. They will be all accessible in **\$route.params**, defined by their name.

```
const User = {  
    template: `<div>  
        My name is {{$route.params.firstname}} {{$route.params.lastname}}  
    </div>  
`}  
  
}  
  
const router = new VueRouter({  
    routes: [  
        {  
            path: '/user/:firstname/:lastname',  
            component: User  
        }  
    ]  
})
```

Params Changes

Note that when a routes render the same component, the same component instance will be reused, so lifecycle hooks of the component will not be called again.

For trigger params changes, we can attach a watcher to our route :

```
const User = {
  watch: {
    '$route' (to, from) {
      // Do something
    }
  }
}
```

Or, use the beforeRouteUpdate (but not available before the 2.2)

```
const User = {
  beforeRouteUpdate (to, from, next) {
    // Do something
  }
}
```

Best practices for props

We saw that our props are accessible in `$route.params`. But this syntax can be hard to understand when many props are passed. A better way to do is to use `props` option.

```
const User = {  
  props: ['id'],  
  template: '<div>User {{ id }}</div>',  
}  
const router = new VueRouter({  
  routes: [  
    {  
      path: '/user/:id',  
      component: User,  
      props: true  
    }  
  ]  
})
```

As we can see, `id` is now directly passed as prop to our component.

Named routes

Every routes can have a name in addition of their paths. To do so we just need to precise it in the route declaration.

```
const router = new VueRouter({
  routes: [
    {
      path: '/user',
      name: 'user',
      component: User
    }
  ]
})
```

```
<router-link :to="{ name: 'user' }">
  User
</router-link>
```

Named routes

- Named routes can be very useful when we want to pass props directly from our HTML.

```
const router = new VueRouter({  
  routes: [  
    {  
      path: '/user/:userId',  
      name: 'user',  
      component: User  
    }  
  ]  
})
```

```
<router-link :to="{ name: 'user', params: { userId: 42 }}">  
  User  
</router-link>
```

Named views

Named views can be used when we want to render many views at the same time. For that we just need to pass a parameter to router-view called **name**.

Example :

```
<router-view></router-view>
<router-view name="sidebar"></router-view>
<router-view name="navbar"></router-view>
```

```
const router = new VueRouter({
  routes: [
    {
      path: '/',
      components: {
        default: Main,
        sidebar: Sidebar,
        navbar: Navbar,
      }
    }
  ]
})
```

A router-view without a name will be given default.

Programmatic Navigation

Inside of a Vue instance, you have access to the router instance as \$router.

- We can use \$router to navigate :

```
router.push({ name: 'user', params: { userId: 42 }})
```

same as

```
<router-link :to="{ name: 'user', params: { userId: 42 }}" />
```

- To replace our current history entry.

```
router.replace(...)
```

same as

```
<router-link :to="..." replace />
```

- Or to go forwards or go backwards in the history stack.

```
router.go(-1)
```

Redirect

Vue router allow us to redirect path, for example redirect from '/foo' to '/bar'

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      redirect: '/bar'
    }
  ]
})
```

We can also use a function instead of passing a hard coded path :

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      redirect: to => {
        // Code here
      }
    }
  ]
})
```

Alias

An alias can be used when we want to apply the same logic to some routes.

For example if we want to share the component Main to both paths `/` and `/home` we will use an alias.

Example :

```
const router = new VueRouter({
  routes: [
    {
      path: '/',
      component: Home,
      alias: '/home'
    }
  ]
})
```

Lazy-loading Routes

When application starts to grow, application bundle does too. A good solution to gain performance is to load on demand when the route is accessed.

Here is a basic example of how we can process to do so :

```
import Main from './routes/Main'

const routes = [
  {
    path: '/main',
    component: Main
  },
  {
    path: '/other',
    component: () => import('./routes/LazyLoadedPage')
  }
]
```

Thanks to `() => import('...')` our file LazyLoadedPage will be loaded only when visiting.





PW6

Data Fetching

Overview

- What is Vue.js
- The Vue instance
- Components
- Directives
- Filters
- Routing
- *Data fetching*
- Vue-CLI
- Plugins
- Typescript
- Vuex
- Testing
- Server Side rendering

Data fetching

Most of the time we would want to fetch data from a remote source or consume an API in our applications.

At first, vue-resource was used to do but since the creator of Vuejs (Evan You) has announced that the Vue.js team is retiring vue-resource, Axios is recommended.

Axios

Axios is one of the most popular HTTP client libraries, it uses promises by default and runs on both the client and the server (so we can combine with `async/await`). In addition, it is universal, supports cancellation, and has TypeScript definitions.

Installation :

```
$ yarn add axios
```

```
$ npm install axios --save
```

Fetching Data

Here is a basic example of how we can get some data from an url :

```
new Vue({  
  el: '#app',  
  methods: {  
    getData: function() {  
      axios.get('http://www.example.com/data')  
        .then(function(response) {  
          console.log(response.data);  
        })  
        .catch(function(error) {  
          console.log(error);  
        });  
    }  
  }  
})
```

POST data

Example :

```
new Vue({  
  el: '#app',  
  methods: {  
    getData: function() {  
      axios.post('http://www.example.com/data', {  
        body: 'toto',  
      })  
        .then(function(response) {  
          console.log(response.data);  
        })  
        .catch(function(error) {  
          console.log(error);  
        });  
    }  
  }  
})
```

Create a base instance

Axios allows us to create a base instance to make it easier to configure. We need attributes such as base url or headers.

Example :

```
const HTTP = axios.create({
  baseURL: 'http://www.example.com',
  headers: {
    Authorization: 'authorized token'
  }
})

...
getData: function() {
  HTTP.get('data')
    .then(function(response) {
      console.log(response.data);
    })
    .catch(function(error) {
      console.log(error);
    });
}
...
}
```





PW7

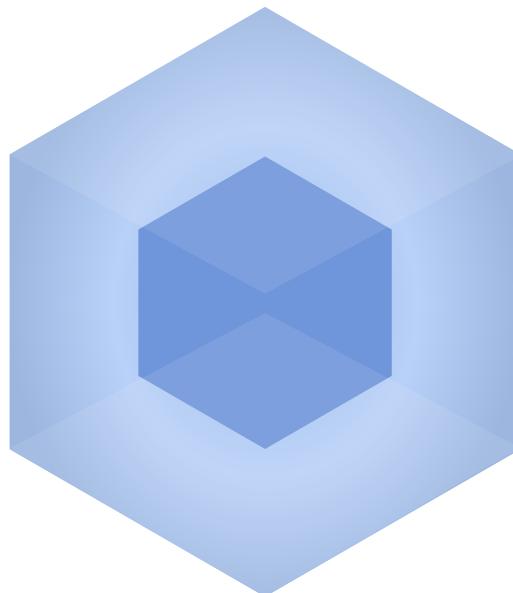
Vue-CLI

Overview

- What is Vue.js
- The Vue instance
- Components
- Directives
- Filters
- Routing
- Data fetching
- *Vue-CLI*
- Plugins
- Typescript
- Vuex
- Testing
- Server Side rendering

Webpack

- Browse all `import` from endpoint.
- Bundle all exports into one file.
- Extensible with plugins / loaders.
- Devtools: HTTP server, live-reload, hot module replacement.
- `vue-loader` to works with vue's `single file components`



Single file components

- File ending with `.vue`
- Allow mixing into one file
 - `template`: `html` or `pug`
 - `script`: `javascript` or `typescript`
 - `style`: `css`, `sass`, `scss`, `stylus`

The style can be scoped to a component:

```
<style scoped>
  ...
</style>
```

Single file components - example

```
▼ SingleFileComponent.vue ✘
1  <template>
2    |   <h1>{{ title }}</h1>
3  </template>
4
5  <script>
6  export default {
7    name: 'app',
8    data () {
9      return {
10        title: 'My app'
11      }
12    }
13  }
14  </script>
15
16  <style>
17  h1 {
18    font-size: 2em;
19    margin-bottom: 20px;
20  }
21  </style>
22  |
```

Single file components - example

```
▼ SingleFileComponent.advanced.vue ×  
1  <template lang="pug">  
2  |   div  
3  |     h1 {{ title }}  
4  |   </template>  
5  <script lang="ts">  
6  import Vue, { ComponentOptions } from 'vue'  
7  export default {  
8    data () {  
9      return {  
10        title: 'My App'  
11      }  
12    }  
13  } as ComponentOptions<Vue>  
14 </script>  
15 <style lang="stylus" scoped>  
16   h1  
17     font-size 2em  
18     margin-bottom 20px  
19 </style>  
20
```

Vue-CLI

Installation

```
npm i -g @vue/cli
```

`vue` binary will allow you to manage vue project via command line or via an ui.

```
vue ui
```

You can also install a plugin and invoke its generator in an already created project

```
vue add [options] <plugin> [pluginOptions]
```

`vue` is able to generate webpack configuration to inspect it and rewrite it inside the **`vue.config.json`**

```
vue inspect [options] [paths...]
```

Scaffold new project

Create new project

```
vue create [options] <app-name>
```

Use default preset or build a custom one by selecting features:

- TypeScript
- PWA Support
- Router
- Vuex
- CSS Pre-processors
- Linter / Formatter
- Unit testing
- E2E Testing



PW8

Extend default configuration

Default configuration can be extended by adding a **vue.config.js** file at the project root.

```
// vue.config.js
module.exports = {
  baseUrl: '/',
  outputDir: 'dist',
  lintOnSave: true,
  compiler: false,
  chainWebpack: (config) => {},
  configureWebpack: (config) => {},
  vueLoader: {},
  productionSourceMap: true,
  css: {},
  parallel: require('os').cpus().length > 1,
  dll: false,
  pwa: {},
  devServer: {},
  pluginOptions: {}
}
```

Extend default webpack configuration

vue-cli 3 hide the configuration of webpack. `vue inspect` command show us configuration of the current project as:

```
module: {
  noParse: /^(vue|vue-router|vuex|vuex-router-sync)$/,
  rules: [
    /* config.module.rule('vue') */
    {
      test: /\.vue$/,
      use: [
        /* config.module.rule('vue').use('cache-loader') */
        {
          loader: 'cache-loader',
          options: { ... }
        }
      ]
    }
  ]
}
```

The comment explains us how overwrite specific configuration.



Plugins vue-cli 3

Useful to keep the same configuration across different projects.

```
.  
├── README.md  
├── generator.js # generator (optional)  
├── prompts.js # prompts file (optional)  
└── index.js # service plugin  
└── package.json
```

- **service plugin**: save a specific webpack configuration
- **generator**: can modify `package.json` and add files
- **prompts file**: make the plugin customizable at runtime

Preset

Vue CLI 3 provide a default preset. You can create and use a company preset to be sure that each new project will use the same libs.

```
{
  "useConfigFiles": true,
  "plugins": {
    "@vue/cli-plugin-babel": {},
    "@vue/cli-plugin-eslint": {
      "config": "airbnb",
      "lintOn": ["save", "commit"]
    },
    "homemade-plugin": {
      "auth": "onstart"
    }
  },
  "configs": {
    "eslintConfig": {
      "env": {
        "jest": true
      }
    }
  }
}
```



Plugins

Overview

- What is Vue.js
- The Vue instance
- Components
- Directives
- Filters
- Routing
- Data fetching
- Vue-CLI
- *Plugins*
- Typescript
- Vuex
- Testing
- Server Side rendering

Why plugins ?

- Add global methods or properties (**vue-router**)
- Register global directives, components, filters... (**vuetify**)
- Enrich components behavior with global mixins (**vuex**)
- Add methods to the Vue instance (**this.\$store**, **this.\$route**)
- Add an external library with custom API (**vue-router**)



Using a plugin

- Just call `Vue.use`

```
import myPlugin from './path/to/myPlugin'  
  
Vue.use(myPlugin, { myPluginOption: true })
```

- `Vue.use` automatically prevent multiple initialization

```
import myPlugin from './path/to/myPlugin'  
Vue.use(myPlugin, { config: 1 })  
Vue.use(myPlugin, { config: 2 })  
Vue.use(myPlugin, { config: 3 })  
  
// myPlugin is initialized with {config: 1}
```

- awesome-vue, the list of Vue plugins



awesome

Writing a plugin

- export an object with `install` function

```
export default {  
  
  install (Vue, options) {  
    // Add anything you want via Vue instance in params  
  }  
}
```

Writing a plugin

- You can add VueJS components, filters, directives

```
export default {

  install (Vue, options) {
    Vue.directive('my-directive', { /* ... */ })
    Vue.component('my-component', { /* ... */ })
    Vue.filter('my-filter', () => { /* ... */ })
  }

}
```

Writing a plugin

- You can add global method or property

```
export default {

  install (Vue, options) {
    Vue.myGlobalMethod = function () {/* ... */}
    Vue.myGlobalObject = {}
  }
}
```

Writing a plugin

- You can add an instance method
- We usually prefix method with a \$

```
export default {

  install (Vue, options) {
    Vue.prototype.$myMethod = function (arg) { return arg * 2; }
  }
}
```

- You can now use \$myMethod in each components

```
export default {
  data() {
    return {
      toto: this.$myMethod(21)
    }
  }
}
```





PW9 - Bonus

TypeScript

Overview

- What is Vue.js
- The Vue instance
- Components
- Directives
- Filters
- Routing
- Data fetching
- Vue-CLI
- Plugins
- *TypeScript*
- Vuex
- Testing
- Server Side rendering

Introduction



- Language created by **Anders Hejlsberg** in 2012
- Open source project maintained by **Microsoft**
- Inspired by **JavaScript**, **Java** et **C#**
- Alternatives : CoffeeScript, Dart, Haxe or Flow

Functionalities

TypeScript is a language for application-scale JavaScript development. It's a typed superset of JavaScript that compiles to plain JavaScript. It allows to write components as classes using properties, decorators and using data types.

- **ES2015+**
- Typing
- Generics
- Classes / Interfaces / Inheritance
- Modular development
- configuration files
- Mixins
- Decorator

Installation

With **vue-cli 3** it's easier to play with TypeScript. During the creation process you can add TypeScript plugin when prompt asks you.

Or use the `invoke` command:

```
vue invoke @vue/typescript
```

This generates base files as `App.vue` with TypeScript configuration and completes webpack configuration to work with TS:

```
config.rule('ts')
config.rule('ts').use('ts-loader')
config.rule('ts').use('babel-loader') (when used alongside @vue/cli-plugin-babel)
config.rule('ts').use('cache-loader')
config.plugin('fork-ts-checker')
```

Declaring component with Typescript

As we can see, we now use the **@Component** decorator from **vue-property-decorator** to declare our component. It transforms the class into a format that Vue can understand during the compilation process.

```
import { Component, Vue } from 'vue-property-decorator';

@Component
export default class HelloWorld extends Vue {
  myData: any
}
```

Also, our data properties are no longer declared in a **data** option but directly in our component, followed by property type.

Passing props

For passing props, we will use again the **vue-property-decorator** by calling **@Prop** decorator.

```
import { Component, Prop, Vue } from 'vue-property-decorator'

@Component
export default class App extends Vue {
  @Prop({default: 'Prop example'}) // We set a default value for our prop
  exampleProperty: string
}
```

Component method

Component methods are no longer declared in **methods** element option but directly in the component as data.

```
import { Component, Vue } from 'vue-property-decorator'

@Component
export default class App extends Vue {
    myMethod () {
        alert('Example method')
    }
}
```

Computed Properties

Computed properties are written as getters and setters in the class.

```
import { Component, Vue } from 'vue-property-decorator'

@Component
export default class App extends Vue {
  get myComputedExample() {
    return console.log('Computed getter example')
  },
  set myComputedExample(value) {
    return console.log('Computed setter example')
  }
}
```



Watchers

Watchers from **vue-property-decorator** have the same behaviour as Vue vanilla version but the syntax is more clear.

```
import { Component, Vue, Watch } from 'vue-property-decorator'

@Component
export default class App extends Vue {
    myData: string

    @Watch('myData') // We trigger here the data we want to watch
    onPropertyChanged(value: string, oldValue: string) {
        console.log(value, oldValue)
    }
}
```

Create custom decorator

vue-class-component allow us to create custom decorator. For that we just need to import **createDecorator()** and declare our new custom decorator.

```
import { createDecorator } from 'vue-class-component'

export const ReverseTab = createDecorator((options, key) => {
  options.computed[key] = options.computed[key].reverse()
})
```

```
import { ReverseTab } from './decorators'

@Component
class MyComp extends Vue {
  myTab = [1, 2, 3]

  @ReverseTab
  get myTab () {
    return this.myTab
  }
}
```



Adding Custom Hooks

Component.registerHooks allows us to register such hooks as :

```
import { Component, Vue } from 'vue-class-component'

Component.registerHooks([
  'beforeRouteEnter',
  'beforeRouteLeave'
])
```

```
import { Component, Vue } from 'vue-class-component'

@Component
class MyComp extends Vue {
  beforeRouteEnter () {
    console.log('beforeRouteEnter')
  }

  beforeRouteLeave () {
    console.log('beforeRouteLeave')
  }
}
```

Note that you have to register the hooks before component definition.





PW10 - Bonus

Vuex

Overview

- What is Vue.js
- The Vue instance
- Components
- Directives
- Filters
- Routing
- Data fetching
- Vue-CLI
- Plugins
- Typescript
- *Vuex*
- Testing
- Server Side rendering

Vuex - Intro

Vuex is a state management pattern + library for Vue.js applications and inspired by **Flux**, **Redux** and **The Elm Architecture**. It serves as a centralized store for all the components in an application.

The basic idea behind that is that passing props for deeply nested components can be painful and hard to maintain, so why not extract shared state out of the components, and manage it in a global singleton?



Installation

Vuex need to be installed in your app with npm or yarn.

```
$ yarn add vuex
```

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

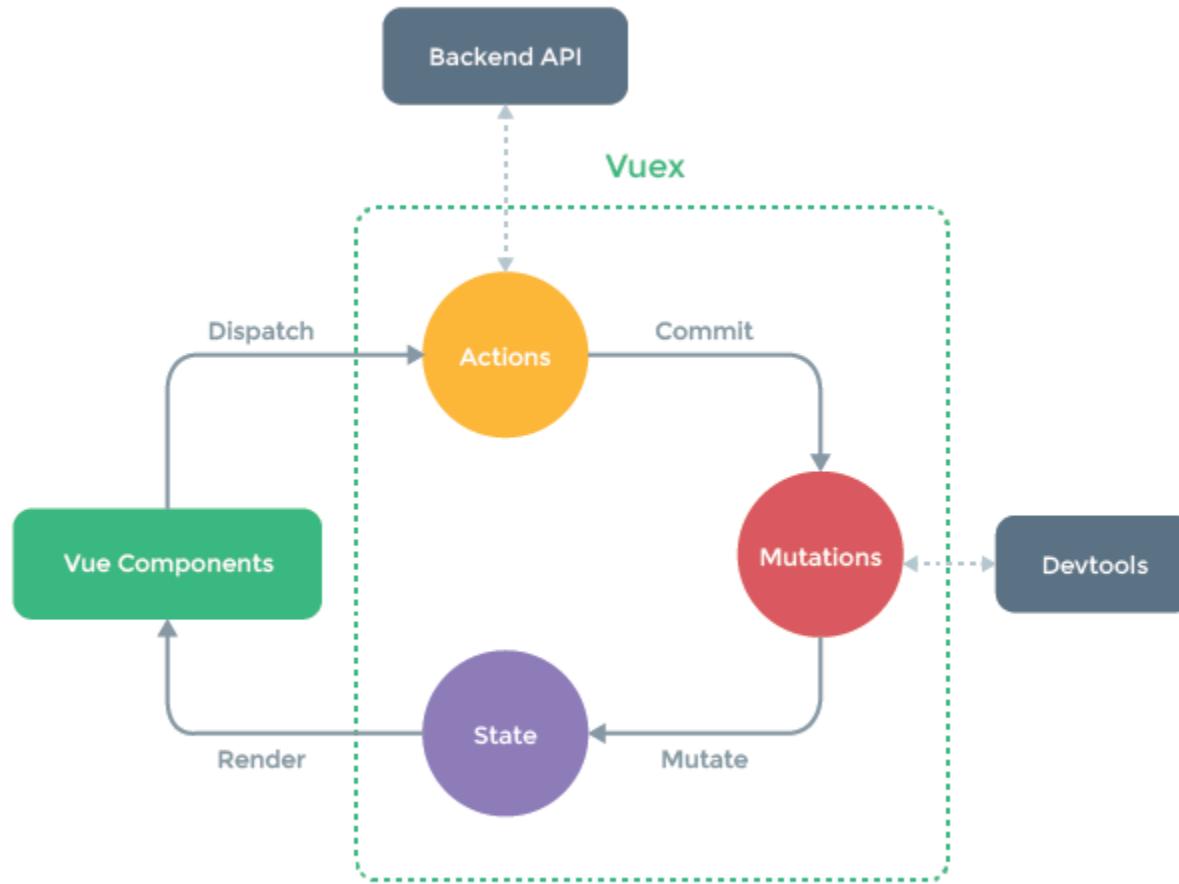
Make sure to enable Vuex in your app by using :

```
import Vue from 'vue';
import Vuex from 'vuex';
import App from 'App.vue';

Vue.use(Vuex);

new Vue({
  el: '#app',
  render: h => h(App)
});
```

State management with Vuex



Creating a store

A store is a container that will hold your application state. It cannot be accessed or modified directly in order to ensure a consistent state.

```
// store.js
const store = new Vuex.Store({
  state: {
    count: 0
  }
})

// app.js
import Vue from 'vue';
import Vuex from 'vuex';
import App from 'App.vue';
import { store } from './store.js'; // importation of our store

Vue.use(Vuex);

new Vue({
  store, // passing our store as option
  el: '#app',
  render: h => h(App)
});
```

Store injection

Injecting our store into the root Vue instance will automatically into every other component in our app.

We will be able to access to our store by simply using ***this.\$store***

Example :

```
console.log(this.$store.state.count) // -> 0
```

Getters

Sometimes we may need to compute derived state based on store state. For that we will use a concept of Vuex.store that is called **Getters**. Getters can be seen as 'computed properties for store'.

In our example we will ask to our store to return a list of todos where done is equal true :

```
const store = new Vuex.Store({
  state: {
    todos: [
      { id: 1, done: true },
      { id: 2, done: false },
    ],
  },
  getters: {
    doneTodos: state => {
      return state.todos.filter(todo => todo.done)
    }
  },
})
store.getters.doneTodos // -> [{id: 1, done: true}]
```

Getters

Getters will also receive other getters as the 2nd argument.

Example :

```
getters: {
  doneTodosCount: (state, getters) => {
    return getters.doneTodos.length
  }
}

store.getters.doneTodosCount // -> 1
```

Also you can pass arguments to getters by returning a function :

```
getters: {
  getTodoById: (state, getters) => (id) => {
    return state.todos.find(todo => todo.id === id)
  }
}

store.getters.getTodoById(2) // -> { id: 2, done: false }
```

Mutations

Direct modification of state in Vuex is done by calling a function called a mutation. Vuex mutations are very similar to events.
A mutation is passed the current state and an optional payload.

Example :

```
const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment (state, n) {
      state.count += n
    }
  }
})
store.commit('increment', 10) // -> count = count + 10
```

Note that mutations must be synchronous and shouldn't return a value. It ensure state is not dependent on the sequence.

Actions

Actions are similar to mutations but instead of mutating the state, actions commit mutations. They can contain arbitrary asynchronous operations.

We need to pass the entire state context as parameter to access getters and commit mutations. Multiple mutations are allowed inside an action.

Example :

```
const store = new Vuex.Store({  
  ...  
  actions: {  
    increment (context) {  
      context.commit('increment')  
    }  
  }  
})  
  
store.dispatch('increment')
```

Actions

Note that if we want to use async / await we can use something like this :

```
// assuming `getData()` and `getOtherData()` return Promises

actions: {
  async actionA ({ commit }) {
    commit('gotData', await getData())
  },
  async actionB ({ dispatch, commit }) {
    await dispatch('actionA') // wait for 'actionA' to finish
    commit('gotOtherData', await getOtherData())
  },
}
```

Modules

Vuex allows us to divide our store into modules. Each module can contain its own state, mutations, actions, getters.

```
const moduleA = {
  state: { ... },
  mutations: { ... },
  actions: { ... },
  getters: { ... }
}

const moduleB = {
  state: { ... },
  mutations: { ... },
  actions: { ... }
}

const store = new Vuex.Store({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a // -> 'moduleA''s state
store.state.b // -> 'moduleB''s state
```

Modules

If you want your modules to be reusable, you can mark it as namespaced. When the module is registered, all of its getters, actions and mutations will be automatically namespaced based on the path the module is registered at.

Example :

```
const store = new Vuex.Store({
  modules: {
    account: {
      namespaced: true,
      state: { ... },
      getters: {
        isAdmin () { ... } // -> getters['account/isAdmin']
      },
      actions: {
        login () { ... } // -> dispatch('account/login')
      },
      mutations: {
        login () { ... } // -> commit('account/login')
      },
    }
  }
})
```





PW11

Testing

Overview

- What is Vue.js
- The Vue instance
- Components
- Directives
- Filters
- Routing
- Data fetching
- Vue-CLI
- Plugins
- Typescript
- Vuex
- *Testing*
- Server Side rendering

Test runner

- You must use...**any runner you like**
- Official recommendation



- Works with webpack and browserify
- Alternative thanks to jsdom



⚠ When you use .vue files, you should use **jest-vue-preprocessor**

Test framework

- Jasmine



- Mocha / Chai



- Jest



Testing Vue

Test the instance

```
import Vue from 'vue'  
import MyComponent from 'path/to/MyComponent.vue'  
  
describe('MyComponent', () => {  
  it('should have a default title', () => {  
    const vm = new Vue(MyComponent).$mount()  
    expect(vm.title).toBe('awesome title')  
  })  
  it('should render the title', () => {  
    const vm = new Vue(MyComponent).$mount()  
    expect(vm.$el.outerHTML).toBe('<h1>awesome title</h1>')  
  })  
})
```



Component testing

Vue.extend to pass custom props

```
import Vue from 'vue'  
import MyComponent from 'path/to/MyComponent.vue'  
  
describe('MyComponent', () => {  
  it('should use the props title', () => {  
    const Ctor = Vue.extend(MyComponent)  
    const vm = new Ctor({  
      propsData: { title: 'my custom title' }  
    }).$mount()  
    expect(vm.title).toBe('my custom title')  
  })  
})
```



Async updates

Vue.nextTick to trigger DOM update

```
import Vue from 'vue'  
import MyComponent from 'path/to/MyComponent.vue'  
  
describe('MyComponent', done => {  
  it('should update the title', () => {  
    const vm = new Vue(MyComponent).$mount()  
    vm.title = 'next title'  
    expect(vm.$el.textContent).toBe('default title')  
    Vue.nextTick(() => {  
      expect(vm.$el.textContent).toBe('next title')  
      done()  
    })  
  })  
})
```



Vue testing tools

vue-test-util

- Testing tools for Vue
- Check the Documentation

```
import { mount } from '@vue/test-utils'
import MyComponent from 'path/to/MyComponent.vue'

describe('MyComponent', () => {
  it('should use the props title', () => {
    const wrapper = mount(MyComponent, {
      propsData: { title: 'my custom title' }
    })
    expect(wrapper.vm.title).toBe('my custom title')
  })
  it('should render the title', () => {
    const wrapper = mount(MyComponent)
    expect(wrapper.html()).toBe('<h1>awesome title</h1>')
    expect(wrapper.first('h1').text()).toBe('awesome title')
  })
})
```



vue-test-util

- Mock instance methods

```
import { mount } from '@vue/test-utils'
import MyComponent from 'path/to/MyComponent.vue'

describe('MyComponent', () => {
  it('should mock router', () => {
    const $route = {path: 'http://www.example-path.com'}
    const wrapper = mount(MyComponent, {
      globals: {
        $route
      }
    })
    expect(wrapper.vm.$route.path).toEqual($route.path);
  })
})
```

vue-test-util

- Shallow render

```
import { shallowMount } from '@vue/test-utils'  
import MyParentComponent from 'path/to/MyParentComponent.vue'  
import MyChildComponent from 'path/to/MyChildComponent.vue'  
  
describe('MyComponent', () => {  
  it('should render only my component', () => {  
    const wrapper = shallowMount(MyParentComponent)  
    expect(wrapper.contains(MyChildComponent)).toBe(true)  
  })  
})
```





PW12

Going deeper

End to end

- Works fully with Vue
- Nightwatch



- Navalia

NAVALIA



Server Side Rendering

Overview

- What is Vue.js
- The Vue instance
- Components
- Directives
- Filters
- Routing
- Data fetching
- Vue-CLI
- Plugins
- Typescript
- Vuex
- Testing
- *Server Side rendering*

Server side rendering vs Client side rendering

Initially, web frameworks had views rendered at the server. Whenever you want to see a new web page, your browser makes a request to the server and it returns the HTML content needed.

But today front end frameworks introduced the fact that content is rendered directly in the browser using JavaScript.

The main benefit is that the browser will not make another request to the server when you want to load more content. It will reload only the part we want to.

Disadvantage of using client-side-rendering is that can be bad for SEO.

Several search engines are unable to crawl well the content.

Another point is that the initial load can be slow. Browser will first ask to the server the HTML content and then our javascript app content that contains all our application.

Building a server side rendering with Express

We will use **vue-server-renderer** and **express** to build a server side rendering application.

```
$ yarn add express vue-server-renderer
```

```
$ npm install express vue-server-renderer --save
```

Rendering a vue instance

```
const Vue = require('vue')
const server = require('express')()
const renderer = require('vue-server-renderer').createRenderer()

server.get('*', (req, res) => {
  const app = new Vue({
    data: {
      url: req.url
    },
    template: '<div>The visited URL is: {{ url }}</div>'
  })
  renderer.renderToString(app, (err, html) => {
    res.end(`
      <html lang="en">
        <head><title>Hello World!</title></head>
        <body>${
          html
        }</body>
      </html>
    `)
    ...
  })
}

server.listen(8080)
```

Using a template page

Let's go ahead by using an HTML template file instead of writing it in our javascript code.

```
index.template.html
```

```
<html lang="en">
  <head><title>Hello world!</title></head>
  <body>
    <!--vue-ssr-outlet-->
  </body>
</html>
```

```
main.js
```

```
const renderer = createRenderer({
  template: require('fs').readFileSync('./index.template.html', 'utf-8')
})
renderer.renderToString(app, (err, html) => {
  console.log(html)
})
```

vue-ssr-outlet is where your app's markup will be injected, it's similar to the **el** option of instance.

Nuxt

Nuxt is another alternative to avoid client side rendering. Nuxt is a framework built on top of the Vue.js. Instead Nuxt offers server side rendering or static file rendering to boost the performance of your web application.

Nuxt - Directory Structure

- `assets`: un-compiled assets such as Less, Sass or JavaScript
- `Components`: Vue.js Components. Nuxt.js doesn't supercharge the data method on these components
- `Layout`: Application Layouts
- `Middleware`: custom functions that can be run before rendering either a page or a group of pages (layouts)
- `Pages`: Application Views and Routes.
- `Plugins`: Javascript plugins that you want to run before instantiating the root Vue.js Application.
- `Static`: static files served on /
- `Store`: static files served on /

Nuxt - Routing

```
pages/  
--| user/  
-----| index.vue  
-----| _id.vue  
--| index.vue
```

will generate

```
router: {  
  routes: [  
    {  
      name: 'index',  
      path: '/',  
      component: 'pages/index.vue'  
    },  
    {  
      name: 'user',  
      path: '/user',  
      component: 'pages/user/index.vue'  
    },  
    {  
      name: 'user',  
      path: '/user/:id',  
      component: 'pages/user/id.vue'  
    }]
```

Nuxt - Template Layout

Nuxt add the following properties and hook to each page components :

- `asyncData`: equivalent to the `data` attribute, it can be asynchronous and receives the context as argument.
- `fetch`: Used to fill the store of vuex before rendering the page, it's like the `data` method except it doesn't set the component data.
- `head`: Set specific Meta Tags for the current page.
- `layout`: Specify a layout defined in the layouts directory.
- `transition`: Set a specific transition for the page.
- `scrollToTop`: Specify if you want the page to scroll to the top before rendering the page, it's used for nested routes. It set to `false` by default
- `validate`: Validator function for dynamic routes.
- `middleware`: Set a middleware for this page, the middleware will be called before rendering the page.

Nuxt - Vuex

Inside `store/index.js` classic store.

Each `store/myModule.js` create a namespaced module.





PW13 - Bonus