
XebiCon'17
Entrevue avec Vue.js
par Ludovic Ladeu et Thomas Champion
1

Nous

Ludovic Ladeu

Développeur Fullstack

#Back #Web

#Cloud

Thomas Champion

Développeur Fullstack

#Web #CSS

#JS

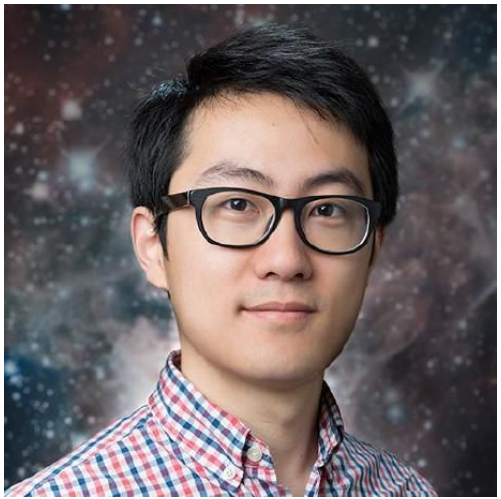
Sommaire

1. Pourquoi Vue ?
2. Architecture d'une application Vue
    a. Les composants
    b. La syntaxe de templating
3. Extensions de Vue
    a. Les plugins
    b. Vue router
4. Hand's On

Histoire

En février 2014, Evan You publie la première version de Vue.js

"I figured, what if I could just extract the part that I really liked about **Angular** and build something really **lightweight** without all the **extra concepts involved**?"



En trois mots

« Approchable »

En trois mots

« Versatile »

En trois mots

« Performant »

Une application en 30 secondes

Architecture d'une application Vue

Disclaimer

Les exemples de code sont en **EcmaScript 6 (ES2015)**

Déclaration de variable

```
var myVar = 1;
```

```
const myVar = 2;
let myVar = 2;
```

Shorthand method

```
var object = {
   hello: function() {

   }
};
```

```
const object = {
   hello() {

   }
};
```

Shorthand property

```
var name = "bob";

var object = {
   name : name
};
```

```
const name = "bob";

const object = {
   name
};
```

Introduction

Pour bootstrapper une application Vue.js :

- Un élément html
- Une instance de Vue lié à cet élément

index.html

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

main.js

```
new Vue({
   el: '#app',
   data: {
       title: 'My application'
   }
})
```

Et c'est parti ?

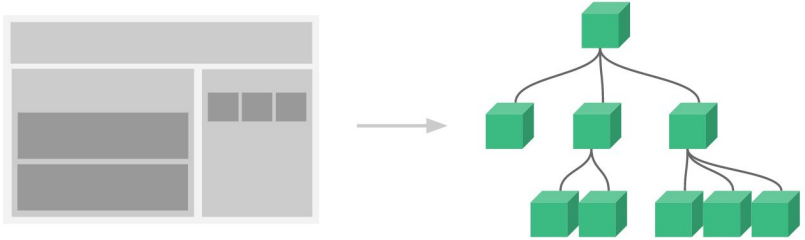
Introduction

Pour bootstrapper une application Vue.js :

- Un élément html
- Une instance de Vue lié à cet élément

Une bonne pratique :

- Mettre en place un composant racine



index.html

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

main.js

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

Les composants

Un composant Vue est défini par **un objet JS** qui se décompose selon la structure suivante :

- **template** : template HTML du composant

```
const MyHeader = {
    template: `<h1>My application</h1>`
};
```

Les composants

Un composant Vue est défini par **un objet JS** qui se décompose selon la structure suivante :

- **template** : template HTML du composant
- **data** : attributs internes au composant

```
const MyHeader = {
    template: `<h1>
      My application at {{ date }}
      </h1>`,
    data() {
        return {
            date: new Date().toString()
        };
    }
};
```

Les composants

Un composant Vue est défini par **un objet JS** qui se décompose selon la structure suivante :

- **template** : template HTML du composant
- **data** : attributs internes au composant
- **methods** : fonctions définies dans le composant

```
const MyHeader = {
    template: `<h1>
      My application at {{ date }} -
      {{ hello() }}</h1>`,
    data() {
        return {
            date: new Date().toString()
        };
    },
    methods : {
        hello() {
            return 'hello world';
        }
    }
};
```

Les composants

Un composant Vue est défini par **un objet JS** qui se décompose selon la structure suivante :

- **template** : template HTML du composant
- **data** : attributs internes au composant
- **methods** : fonctions définies dans le composant
- **props** : attributs passés par le composant appelant

Valoriser une propriété d'un composant :

- Passer un objet

```
<my-header :name="username"></my-header>
```

- Passer une chaîne

```
<my-header name="Bob"></my-header>
```

équivalent à

```
<my-header :name="'Bob'"></my-header>
```

```
const MyHeader = {
    props: ['name'],
    template: `<h1>
      My application at {{ date }} -
      {{ hello() }}</h1>`,

    data() {
        return {
            date: new Date().toString()
        };
    },
    methods : {
        hello() {
            return `hello ${this.name}`;
        }
    }
};
```

Les composants

Un composant Vue est défini par **un objet JS** qui se décompose selon la structure suivante :

- **template** : template HTML du composant (pas pour les fichiers Vue)
- **data** : attributs internes au composant
- **methods** : fonctions définies dans le composant
- **props** : attributs passés par le composant appelant
- **components** : enregistrer localement des composants

```
const MyTime = {
    template: `<span>{{ date }}</span>`,
    data() {
        return {
            date: new Date().toString()
        };
    }
};

const MyHeader = {
    template: `<h1>
      My application at <my-time /> -
      {{ hello() }}</h1>`,
    props: ['name'],
    methods : {
        hello() {
            return `hello ${this.name}`;
        }
    },
    components : { MyTime }
};
```

Les propriétés avancées

- **watch** : méthodes qui permettent d'être notifié lors de la modification d'une propriété

```
const App = {
    template: `
    <div>
        Your name : <input v-model="firstname" />
    </div>`,
    data() {
        return {
            firstname: 'bob'
        }
    },
    watch: {
        firstname(value, oldValue) {
            console.log('watch: firstname changed', value, oldValue);
        }
    }
};
```

Les propriétés avancées

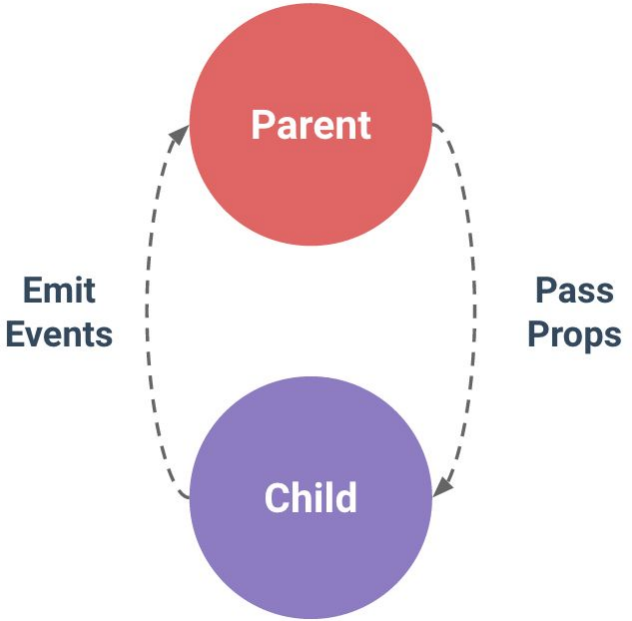
- **computed** : propriété qui est recalculé à chaque fois qu'une propriété dont elle dépend est modifiée. La valeur de la propriété est alors mise en cache

```
const App = {
    template: `<div>{{ computedFullName }} - {{ computedFullName }}</div>`,
    data() {
        return {
            firstname: 'bob',
            lastname: 'dupont'
        }
    },
    computed: {
        computedFullName() {
            return this.firstname + ' ' + this.lastname;
        }
    },
};
```

Communication composants

Un composant peut posséder des entrées-sorties.

- Les entrées sont les propriétés (props)
- Les sorties correspondent à des événements émisent par le composant enfant



Communication composants

Composant parent

```
const MyNotebook = {
    template: `
    <div>
        <p>My name : <input v-model="name" /></p>
        <my-text :text="name" @textChanged="nameChanged()" />
    </div>`,
    data() {
        return {
            name: 'foobar',
        }
    },
    methods: {
        nameChanged(content) {
          this.name = content;
      }
    },
    components: {
        MyText
    }
};
```

Composant enfant

```
const MyText = {
    template: `
    <div>
       <input v-model="content" />
    </div>`,
    props: ['text'],
    data() {
        return {
            content: this.text
        }
    },
    watch: {
        content() {
            this.$emit('textChanged', this.content);
        }
    }
};
```

Le cycle de vie

Un composant peut aussi posséder des "hooks" pour être notifié de son cycle de vie :

Création du composant
- beforeCreated
- created

Ajout au DOM
- beforeMount
- mounted

Mise à jour du DOM
- beforeUpdate
- updated

Destruction du composant
- beforeDestroy
- destroyed

```
const MyContent = {
    template: `<main>{{ content }}</main>`,
    data() {
        return {
            content: ''
        }
    },
    created() {
        restService.getContent()
            .then(content => {
                this.content = content;
            })
    }
};
```

Les fichiers Vue

Un fichier vue est composé comme suit :

- template
- script
- style (**scoped** ou non)

Pris en charge par vue-cli

```
<template>
   <h1>{{ title }} - {{ date }}</h1>
</template>

<script>
   export default {
       props: ['title'],
       data() {
           return {
               date: new Date().toString()
           };
       },
   };
</script>

<style scoped>
   h1 {
       color: blue;
   }
</style>
```

Les composants globaux

- Par défaut les composants utilisés doivent être importés (via la propriété **components**)
- Mais il est possible de créer des composants globaux :

```
Vue.component('box', {
 template: `<div class="box">{{ content }}</div>`,
 props: ['content']
});
```

En déclarant le composant dans un fichier Vue :

```
import Box from './components/Box.vue'
Vue.component('box', Box);
```

La syntaxe de templating

Les bindings

Comment afficher une propriété dans mon template ?

```
<h1>{{ title }}</h1>
```

Comment lier une propriété de mon composant à un attribut html ?

```
<a href="https://vuejs.org/">link</a>

<a v-bind:href="myLink">link</a>

<a :href="myLink">link</a>
```

Les conditions

```
<div v-if="type === 'A'">
      A
</div>

<div v-else-if="type === 'B'">
      B
</div>

<div v-else>
      Not A/B
</div>
```

Les boucles

```
<ul>
      <li v-for="todo in todos">
      {{ todo.text }}
      </li>
</ul>
```

Boucler sur une liste de composants

Depuis Vue 2.2.0 + la propriété **key** est **obligatoire** quand on utilise v-for avec les **composants**.

```
<my-component v-for="item in items" :key="item.id"></my-component>
```

Les événements

```
<button v-on:click="method()"></button>

<button @click="method($event)"></button>
```

Modèle binding

```
<script>
    export default {
        data() {
            return {
                name: 'Bob'
            }
        }
    }
</script>
```

```
<template>
    <div>
        <p>Your name : <input v-model="name" /></p>
        <p>Hello {{ name }}</p>
    </div>
</template>
```

Modèle binding : les modifiers

Les modifiers servent à modifier les valeurs associés à un modèle avant de mettre à jour le modèle

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

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

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

Il est aussi possible de combiner les modifiers :

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

Les Extensions de Vue

Les plugins

- Les plugins ajoutent des fonctionnalités globales à Vue
- Il n'y a pas de portée strictement définie pour un plugin
- il existe plusieurs types de plugins:
  - Ajout de méthodes ou propriétés globales ex. vue-element
  - Ajout d'un ou plusieurs assets: directives/filters/transitions etc. ex. vue-touch
  - Ajout d'options aux components by global mixin. ex. vuex
  - Ajout de méthodes aux instances de Vue en passant par Vue.prototype
  - Une combinaison des méthodes ci-dessus. ex. vue-router.
- Pour utiliser un plugin :

```
Vue.use(MyPlugin);
```

Vue Router

Vue.js permet de créer des applications single page performantes. Pour cela, la gestion des routes est possible via « **vue-router** »

1) Déclarer le plugin VueRouter

```
import VueRouter from 'vue-router';

Vue.use(VueRouter);
```

2) Création des composants

```
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }
```

3) Définition des routes

```
const routes = [
  { path: '/foo', component: Foo },
  { path: '/bar', component: Bar }
]
```

4) Création d'une instance du vue Router

```
const router = new VueRouter({
  routes // short for routes: routes
})
```

5) Création de l'instance Racine Vue avec le router

```
new Vue({
  el: '#app',
  router,
  template: '<app/>',
  components: { App }
});
```

Vue Router

6) Template du composant Racine

```
<template>
  <div id="app">
    <h1>My App</h1>
    <div>
      <router-link to="/foo">Link To Foo</router-link>
      <router-link to="/bar">Link to Bar</router-link>
    </div>
    <!-- route outlet -->
    <!-- component matched by the route will render here -->
    <router-view></router-view>
  </div>
</template>
```

Pour aller plus loin : https://router.vuejs.org/en/

Hand's On

Sujet

Nous souhaitons développer un site web qui permet de partager ses **recettes de cuisines**.
La création de cette application se fera en plusieurs étapes :

1. Création d'un composant affichant une recette
2. Création d'un composant afficher une liste de recette
3. Création d'un composant de détail d'une recette et mise en place du routage
4. Permettre l'ajout d'une nouvelle recette
5. Pouvoir enregistrer en favoris des recettes

Récupération des sources :

https://github.com/xebia-france/xebicon17-vuejs

Ressources

Quelques liens

- La documentation officielle : https://vuejs.org/v2/guide/
- Cheatsheet : https://vuejs-tips.github.io/cheatsheet/
- Bibliothèque de composants : https://vuetifyjs.com/
- Ultime page de ressources : https://github.com/vuejs/awesome-vue
- Bonnes pratiques : https://github.com/pablohpsilva/vuejs-component-style-guide

Astuce

- Pour utiliser SASS : npm install sass-loader node-sass webpack --save-dev
  - puis au niveau de la balise style ajouter : lang=”scss”

```
<style lang="scss">
 #app {
   font-family: 'Avenir', Helvetica, Arial, sans-serif;

   header {
     text-align: center;
   }
 }
</style>
```