

# L3 MIASHS

## FRAMEWORKS

2017-2018

```
L3Miashs.addEventListener( 'frameworkjs', (lecturers) => {  
  console.log( 'This course is started' );  
  lecturers['Blandine_Descamps-Medda'].mail = "blandine.descamps@ling.fr";  
  lecturers['Caroline_Desprat'].mail = "l3miashs@desprat.fr";  
  lecturers['Nathalie_Hernandez'].mail = "hernande@irit.fr";  
  lecturers['Nathalie_Hernandez'].isReferent = true ;  
});
```

**FRAMEWORKS FRONT  
END**

# PLAN

1. Introduction
2. Différents patterns
3. Stack
4. Notions clés

**DIFFÉRENTS PATTERNS**

# MVC

Architecture 3-tiers côté client: Modèle-Vue-Contrôleur

- Modèle : contient les données
- Vue : présentation UI (lit le modèle et envoie les actions utilisateurs)
- Contrôleur : logique des actions utilisateur (traite les actions pour mettre à jour le modèle).

# MVC : CONCLUSION

- Un peu complexe à prendre en main, très dogmatique, fonctionne bien dans de larges applications.
- action → mise à jour → affichage induit par ce patron est bien adapté aux applications web

# MVVM

- Modèle-vue Vue Modèle
- Par rapport au MVC, communication bidirectionnelle entre la Vue et le Modèle, les actions de l'utilisateur entraînent des modifications des données du modèle.séparer la vue de la logique et de l'accès aux données en accentuant les principes de binding et d'événement
- un peu moins dogmatique que MVC

# FLUX

- actions : qu'elles proviennent du serveur ou d'une interaction utilisateur ;
- dispatcher : dans lequel sont envoyées les actions que ce dernier transmet à qui veut, un peu comme un EventEmitter global ;
- stores : (équivalent du modele dans MVC), contiennent les données, et réagissent aux actions que le dispatcher leur transmet ;
- views : s'occupent du rendu des données dans le DOM, et de lancer des actions utilisateur.



# FLUX : CONCLUSION

architecture unidirectionnelle raisonne en actions, déclenchées par la vue ou le serveur Toutes les actions passent par le dispatcher Seuls les stores signalent aux vues qu'il faut se mettre à jour

ref: [PDC : Flux](#)

# FUNCTIONAL REACTIVE PROGRAMMING

*Reactive Programming is an asynchronous programming paradigm concerned with data streams and the propagation of change. -- Wikipedia*

# FUNCTIONAL REACTIVE PROGRAMMING

- State : store immuable . On ne le modifie jamais, on en recrée un nouveau. le State est l'unique source de vérité sur l'état actuel de la TOTALITÉ de l'application côté front.
- Reducer : action + state -> state. (dispatcher simplifié). Le reducer est une fonction PURE, c'est à dire sans aucun effet de bord : les mêmes entrées produisent les mêmes sorties.

Rappel programmation fonctionnelle. Exemple  $f(x) = x+3$ .

# LE DOM ...

DOM = Document Object Model (arbre de noeuds).  
Fait la liaison entre HTML et JS.

Quand faire le nouveau rendu? Doit-on rerendre *tout* le DOM? Quelle partie est prioritaire..

# LE DOM VIRTUEL

Virtual DOM : optimiser les modifications du DOM .  
Algorithme différentiel entre le DOM et le DOM  
Virtual pour savoir quelle partie du DOM re-rendre.

Pros: taux de rafraichissement maximum. On ne touche plus au DOM directement (donc plus besoin de jQuery o/)  
Cons: complexe à prendre en main (cycle de vie des composants)

# RÉSUMÉ

| Framework   | MVC | MVVM | Flux | FRP   |
|-------------|-----|------|------|-------|
| Angular > 2 | x   |      |      | ~     |
| Vue.js      |     | x    |      | vuex  |
| React       |     |      | x    | redux |

# INTRODUCTION À VUE.JS



# PRÉSENTATION

- Vue (prononcé /vju:/ - view)
- Framework (open-source) pour construire des interfaces utilisateur
- Créé en 2014 par Evan You (ancien Google)
- Actuellement en version 2.5 (Javascript)



# OBJECTIFS

- Faciliter le développement d'UI web. Framework moins dogmatique (qu'angular par exemple) -mais quand même : la courbe d'apprentissage est moins sévère.
- Adoption incrémentale de l'architecture (dans le cas d'une web app pré existante par exemple).

# VUE.JS

- Core: declarative rendering + composition de composants
- Extra : routing, gestion de l'état, build (maintenance des bibliothèques et packages supportés officiellement)

**INSTANCE DE VUE**

# CRÉATION

```
var vm = new Vue({  
  // options  
})
```

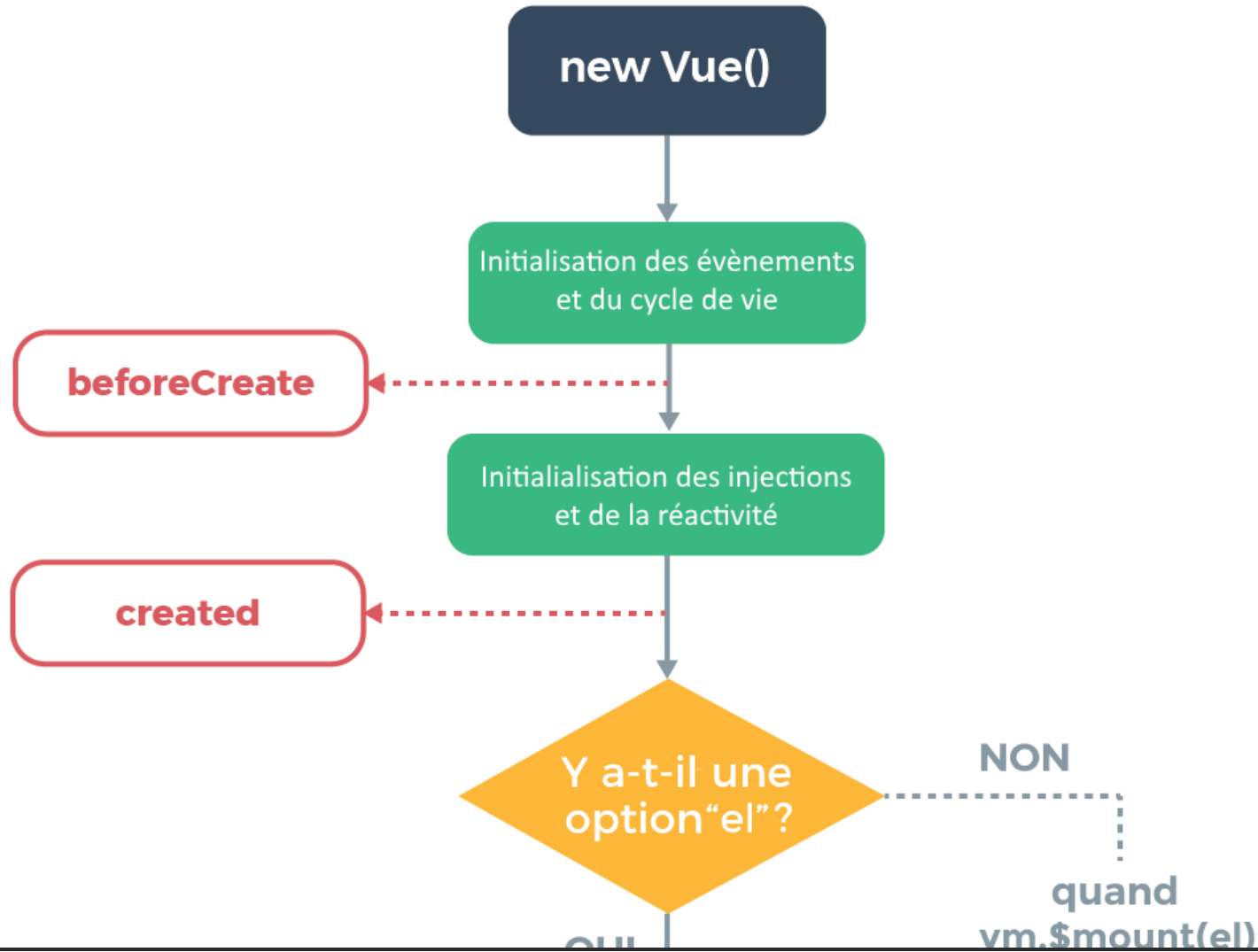
## Le Vue-Modèle dans MVVM

# DONNÉES ET MÉTHODES

```
var data = { a: 1 };  
// L'objet est ajouté à une instance de Vue  
var vm = new Vue({  
  data: data,  
  methods: { // les propriétés calculées  
    sayHello: function() { console.log("hello"); }  
  }  
})  
// La propriété depuis l'instance retourne celle des données originale  
vm.a == data.a; // => true  
vm.a = 2;  
data.a = 3;
```

Quand ces données changent, le rendu de la vue est refait. Il est à noter que les propriétés dans data sont **réactives**

# CYCLE DE VIE D'UNE INSTANCE



# EXEMPLE

```
new Vue({
  data: {
    a: 1
  }, //le cycle de vie
  created: function () {
    // `this` est une référence à l'instance de vm
    console.log('a is: ' + this.a)
  },
  mounted: function () { console.log("i'm mounted") },
  methods: { //les propriétés calculées
  }, computed: { //les propriétés calculées (cache)
  }, watch: { //les propriétés observées
  }
})
```

# PROPRIÉTÉS CALCULÉES ET OBSERVATEURS



# PROPRIÉTÉS CALCULÉES

Opérations plus complexes : utilisation de propriétés calculées (méthodes)

```
new Vue({
  data: {
    table: ["things", "among", "others"]
  },
  methods: {
    //les propriétés calculées
    getTableLength: function() {
      return this.table.length;
    }
  },
  computed: { //les propriétés calculées (cache)
    filterThings: function() {
      this.table.filter((e)=>{ return e == "things"; })
    }
  }
})
```

## COMPUTED VS METHODS :

Dans `computed`, les propriétés sont mises en cache selon leur dépendances. => elles sont réévaluées uniquement si une dépendance (réactive) change.

## PROPRIÉTÉS CALCULÉES VS OBSERVÉES

Pour observer et de réagir aux changements de données sur une instance de Vue : les propriétés `watch`.

# EXEMPLE

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

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

A-t-on vraiment besoin du watch ici .. ?

## EXEMPLE

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

## EXEMPLE DE WATCH

yes / no question

```
<div id="watch-example">
  <p>
    Posez votre question (réponse par Oui ou Non) :
    <input v-model="question">
  </p>
  <p>{{ answer }}</p>
</div>
```

```
var watchExampleVM = new Vue({
  el: '#watch-example',
  data: {
    question: '',
    answer: 'Je ne peux pas vous donner une réponse avant que vou
  },
  watch: {
    // à chaque fois que la question change, cette fonction s'exé
    question: function () {
      this.answer = "J'attends que vous arrêtiez de taper..."
      this.getAnswer()
    }
  },
  methods: {
    getAnswer: function () {
      if (this.question.indexOf('?') === -1) {
```

# RÉSUMÉ

- `methods` sont des méthodes. Elles sont systématiquement toutes appelées lorsque l'une d'elle est appelée. Prend des paramètres.
- `computed` sont des propriétés compilées => la propriété est ré-évaluée seulement si une de ses dépendance change. Ne prend pas de paramètre. Lors de l'instanciation de la vue, elles sont converties sous forme de propriété

Conclusion : si les données doivent être mise en cache, utiliser les propriétés `computed`, sinon `methods`.



- watchest destiné à l'observation du changement de propriétés réactives.

**RENDU DÉCLARATIF  
DANS LE DOM**

# Syntaxe de template (ex. interpolation de texte):

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

```
//script js
let app = new Vue({
  el: "#app",
  data: {
    message: "Hello World !"
  }
});
```

Hello World !

# TEMPLATES

- Les templates sont basés sur la syntaxe HTML (déclaratif)
- Liaison (binding) entre le DOM rendu et les données de l'instance Vue
- HTML valide (spec-compliant)
- Le template est compilé sous la forme de fonctions de rendu dans le *virtual DOM*.

*possibilité d'utiliser JSX (voir cours react)*

# BINDING DE DONNÉES

## Liaison d'un attribut d'un élément

```
<!--template-->
<div id="app">
  <span v-bind:title="message">Hello again !</span>
</div>
```

```
//script js
let app = new Vue({
  el: '#app',
  data: {
    message: 'Je me suis affiché ' + new Date().toLocaleString();
  }
})
```

Que se passe-t-il?

L'attribut `v-bind` est une des **directives** de Vue. Elle permet de lier l'attribut à une valeur dynamique (`v-bind` peut être abrégé par `:`).

```
<!--template-->
<div id="app">
  <span :title="message">Hello again !</span>
</div>
```

# DIRECTIVE

Une **directive** est un attribut spécial fourni par le framework ; elle applique un comportement *réactif* spécifique au DOM après rendu. Dans Vue, les directives sont préfixées par `v-`.

*Remarque : côté un peu magique du framework avec la liaison "automatique" des données et de la vue.*

**RENDU CONDITIONNEL**



## DIRECTIVE v-if, v-else

```
<div id="app-3">
  <p v-if="seen">Maintenant vous me voyez</p>
  <p v-else="">Ou pas</p>
</div>
```

```
let app3 = new Vue({
  el: '#app-3',
  data: {
    seen: true
  }
})
```

et le v-else-if?

## **DIRECTIVE v-show**

v-show : toujours rendu, permutation basée sur du CSS.

Le v-if est un vrai rendu conditionnel (construction/destruction des listeners). Ne se rend que quand la condition est vraie pour la première fois.

## BINDING SUR LES CLASSES (STYLE)

### Liaison de Classes HTML

Il est possible de passer un objet à `v-bind:class` pour permuter les classes automatiquement :

```
<div id="app-3">  
  <p v-if="seen" :class="{active:seen}">Maintenant vous me voyez<  
  <p v-else="">Ou pas</p>  
</div>
```

# RENDU DE LISTES (BOUCLES)

# DIRECTIVE v-for SUR UN TABLEAU

```
<ul id="maliste-1">
  <li v-for="item in items">
    <!-- item est un alias de l'élément en cours d'itération-->
    {{ item.message }}
  </li>
</ul>
```

```
let maliste = new Vue({
  el: "#maliste-1",
  data: {
    items: [{ message: "Foo" }, { message: "Bar" }]
  }
});
```

- Foo
- Bar

## DIRECTIVE v-for SUR UN TABLEAU

```
<ul id="example-2">
  <li v-for="(item, index) in items">
    {{ parentMessage }} - {{ index }} - {{ item.message }}
  </li>
</ul>
```

```
new Vue({
  el: "#iterobject",
  data: {
    parentMessage: "Parent",
    items: [{ message: "Foo" }, { message: "Bar" }]
  }
});
```

Qu'est-ce qui s'affiche?

## DIRECTIVE v-for SUR UN OBJECT

```
<ul id="iterobject">
  <li v-for="value in object">
    {{ value }}
  </li>
</ul>
```

```
new Vue({
  el: "#iterobject",
  data: {
    object: {
      firstName: "John",
      lastName: "Doe",
      age: 30
    }
  }
});
```

Qu'est-ce qui s'affiche?

## DIRECTIVE v-for SUR UN OBJECT

```
<ul id="iterobject">
  <li v-for="(value, key, index) in object">
    {{ index }}. {{ key }}: {{ value }}
  </li>
</ul>
```

```
new Vue({
  el: "#iterobject",
  data: {
    object: {
      firstName: "John",
      lastName: "Doe",
      age: 30
    }
  }
});
```

Qu'est-ce qui s'affiche?



## DIRECTIVE v-for ET LES key

La mise à jour d'une liste peut être coûteuse. Dans Vue pour aider à traquer les modifications, l'attribut `key` permet à Vue de suivre l'identité de chaque noeud (doit être unique). Si `key` est une valeur dynamique, il faut utiliser `v-bind`.

```
<div v-for="item in items" :key="item.id">  
  <!-- contenu -->  
</div>
```

# GESTION DES ÉVÈNEMENTS

Similaire à la couche événementiel du DOM (gestion par callback).

## DIRECTIVE v-on

```
<div id="counter">  
  <button v-on:click="counter += 1">Add 1</button>  
  <p>Le bouton ci-dessus a été cliqué {{ counter }} fois.</p>  
</div>
```

```
let example1 = new Vue({  
  el: "#counter",  
  data: {  
    counter: 0  
  }  
});
```

```
<div id="hello">
  <button v-on:click="greet($event, 'L3') ">Greet</button>
  <span>{{greet2('L3')}}</span>
</div>
```

```
let example2 = new Vue({
  el: "#hello",
  data: { name: "Caroline" },
  methods: {
    greet: function(event, niveau) {
      if (event) {
        alert(
          event.target.tagName,
          this.name + " greets " + niveau + "!"
        );
      }
    },
    greet2: niveau => niveau + " hello"
  }
});
```

## MODIFICATEURS D'ÉVÈNEMENT

Comme pour `event.preventDefault()` ou `event.stopPropagation()` Vue permet d'utiliser des suffixes à de directive pour préserver la logique des données dans les méthodes.

Modificateurs (**ref**): `.stop` , `.prevent`, `.capture`, `.self`, `.once`, `.passive` Exemples:

```
<!-- equivalent à event.stopPropagation() -->  
<a v-on:click.stop="doThis"></a>  
<!-- l'évènement `submit` ne rechargera plus la page -->  
<form v-on:submit.prevent="onSubmit"></form>
```

## DES LISTENER DANS LE HTML ? ET LA SÉPARATION DES RESPONSABILITÉS ?

- Le Modèle-Vue gère la vue courante donc pas de difficulté de maintenance.
- Avantages:
  - Localiser l'implémentation des gestionnaires dans le JS en regardant le HTML
  - Pas d'attache manuel de l'événement (magie du framework)
  - Suppression automatique des listeners à la destruction du Modèle-vue.

**BINDING BI-DIRECTIONNEL**

## DIRECTIVE v-model

Liaison bi-directionnel entre le modèle et la vue sur les champs de formulaire (input, checkbox, select, textarea).

```
<input v-model="message" placeholder="modifiez-moi">
<p>Le message est : {{ message }}</p>
<input type="checkbox" id="checkbox" v-model="checked">
<label for="checkbox">{{ checked }}</label>
```

Possède des modificateurs (.lazy, .number, .trim)



# RÉSUMÉ DES DIRECTIVES

v-bind

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

v-for with keys (arrays and objects)

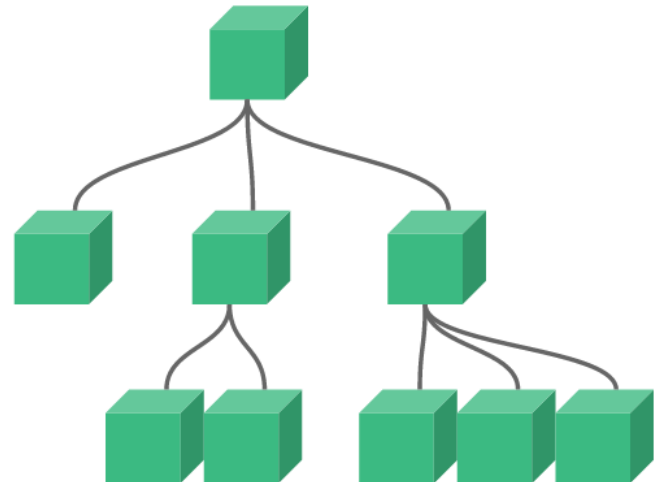
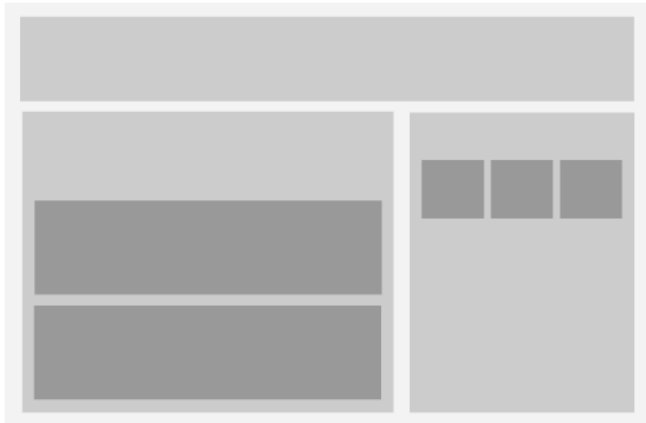
v-on

v-model

# COMPOSANTS

Abstraction permettant de créer des composants réutilisables et autonomes.

*une applications = arbre de composants*



# LE COMPOSANT

Un composant = une instance de Vue

```
Vue.component('todo-item', {  
  template: '<li>Ceci est une liste</li>'  
})
```

Insertion dans un autre composant

```
<ol>  
  <!-- Crée une instance du composant todo-list -->  
  <todo-item></todo-item>  
</ol>
```

et avec des options (propriétés) ?

# LES PROPS

```
Vue.component('todo-item', {  
  props: ['todo'],  
  template: '<li>{{ todo.text }}</li>'  
})
```

Le composant `todo-item` accepte maintenant une «**prop**» (propriété) qui est comme un attribut personnalisé. Cette prop est appelée `todo`.

# LES PROPS : EXEMPLE

```
<div id="maliste">
  <ol>
    <todo-item
      v-for="item in groceryList"
      v-bind:todo="item"
      v-bind:key="item.id">
    </todo-item>
  </ol>
</div>
```

```
var malistedecourses = new Vue({
  el: '#maliste',
  data: {
    groceryList: [
      { id: 0, text: 'Légumes' },
      { id: 1, text: 'Fromage' },
      { id: 2, text: 'Oeufs' }
    ]
  }
})
```

# COMPOSANTS VUE ET CUSTOM ELEMENTS

## Parallèle entre composants Vue et Custom Elements (spec Web Components) :

- Syntaxe proche
- La spec des Web Components est finalisée mais pas implémentée (pas besoin de polyfill avec Vue)
- Un composant de Vue peut être implémenté à l'intérieur d'un élément natif
- Fonctionnalités supplémentaires des composants de Vue (flux de données, événement personnalisé, intégration des outils de build)



**COMPOSANTS**

**MONOFICHIERS**

Composants globaux seront définis en utilisant **Vue.component**, suivi de **new Vue({ el: '#container' })** pour cibler un élément conteneur dans le corps de chaque page.

# PROBLÈME

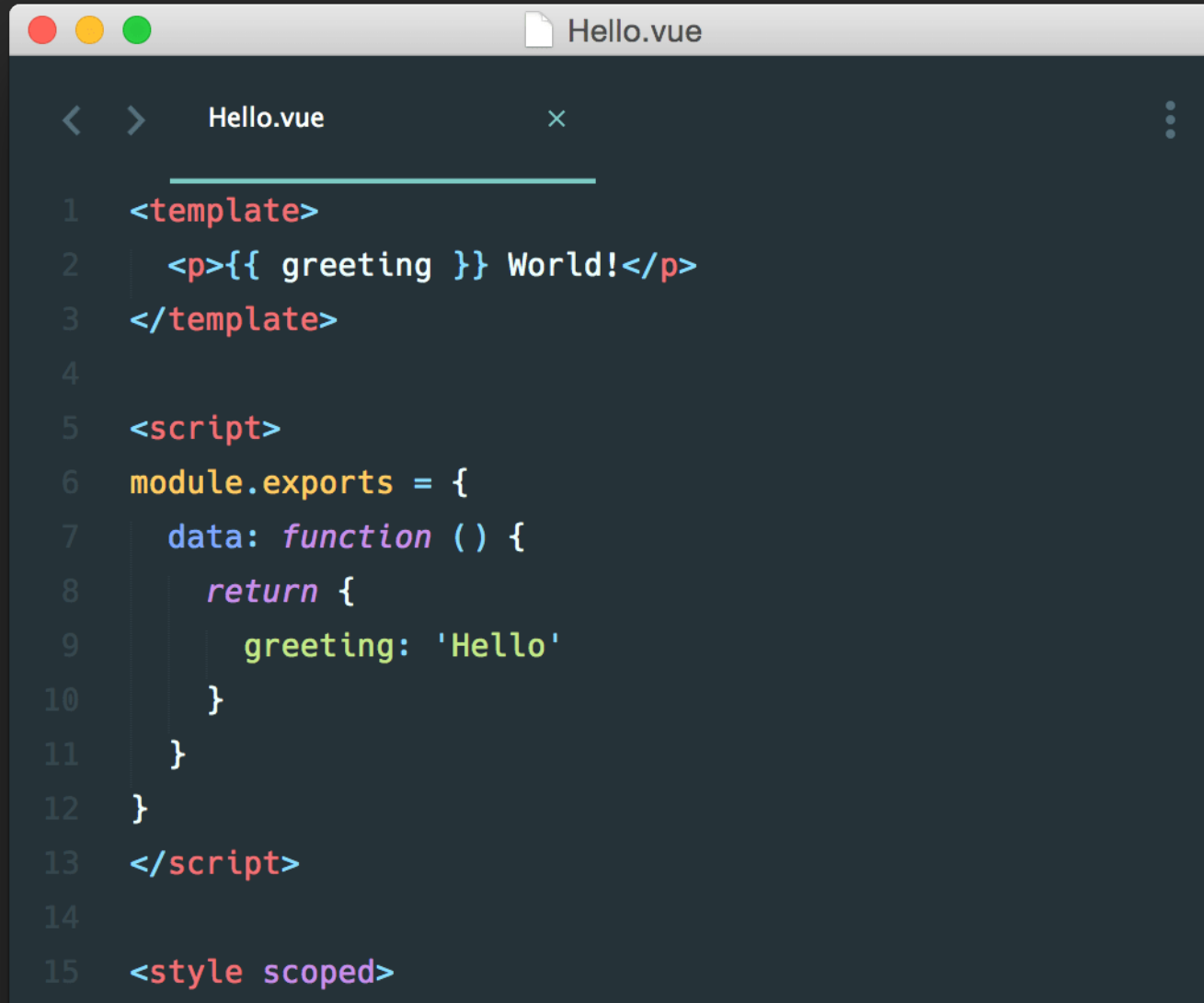
- Les définitions globales forcent à avoir un nom unique pour chaque composant
- (Les templates sous forme de chaînes de caractères ne bénéficient pas de la coloration syntaxique et requièrent l'usage de slashes disgracieux pour le HTML multiligne).

- L'absence de support pour le CSS signifie que le CSS ne peut pas être modularisé comme HTML et JavaScript
- L'absence d'étape de build nous restreint au HTML et à JavaScript ES5 (pas de préprocesseur)

# SOLUTION

- composants monofichiers avec une extension `.vue`, rendus possibles par les outils de build tels que **webpack** ou **Browserify**.

# EXAMPLE HELLO.VUE



A screenshot of a code editor window titled 'Hello.vue'. The editor displays the following Vue.js component code:

```
1 <template>
2   <p>{{ greeting }} World!</p>
3 </template>
4
5 <script>
6   module.exports = {
7     data: function () {
8       return {
9         greeting: 'Hello'
10      }
11    }
12  }
13 </script>
14
15 <style scoped>
```

**AVANTAGES**

- Une coloration syntaxique complète
- Des modules CommonJS
- Du CSS dont la portée est limitée au composant





**EVENEMENTS**

**PERSONNALISÉS**



**LIÉS LES ÉVÉNEMENTS "NATIFS" AUX  
COMPOSANTS**

## Utilisation du modificateur `.native` pour `v-on`

```
<base-input v-on:focus.native="onFocus"></base-input>
```

(attention à la structure du composant)

# MODIFICATEUR `sync`

Comment créer un "two-way binding" (liaison à double sens) pour une prop? Attention maintenant (si les enfants font muter le parent de manière non explicite)

## Deux solutions:

- Emettre un événement de mise à jour explicite pour permettre au parent de l'écouter si besoin.

```
this.$emit('update:title', nouveauTitre)
```

```
<text-document  
v-bind:title="doc.title"  
v-on:update:title="doc.title = $event"  
></text-document>
```

- Utiliser le modificateur `.sync()`

```
<text-document v-bind:title.sync="doc.title"></text-document>
```

# SLOTS

Dans un composant `<slot>` peut être considéré comme une zone de distribution de contenu



```
<navigation-link url="/profile">  
  Mon profil  
</navigation-link>
```

## Le composant navigation-link:

```
<a  
  v-bind:href="url"  
  class="nav-link"  
>  
  <slot></slot>  
</a>
```

Lors du cycle de rendu du composant, l'élément est remplacé par « Mon profil »

**ALLER PLUS LOIN**

# TRANSITIONS (EXEMPLE SIMPLE)

composant conteneur transition pour ajouter des  
transitions entrantes/sortantes

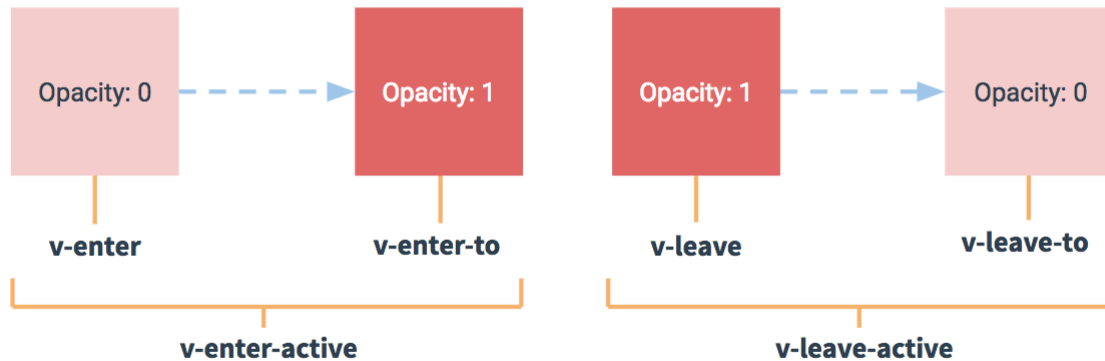
```
<div id="transitionexemple">
  <button v-on:click="show = !show">  Permuter </button>
  <transition name="fade"> <p v-if="show">bonjour</p> </transitio
</div>
```

```
new Vue({
  el: '#demo',
  data: { show: true }
})
```

```
.fade-enter-active, .fade-leave-active {
  transition: opacity .5s;
}
.fade-enter, .fade-leave-to {
  opacity: 0;
}
```

## Entrée

## Sortie



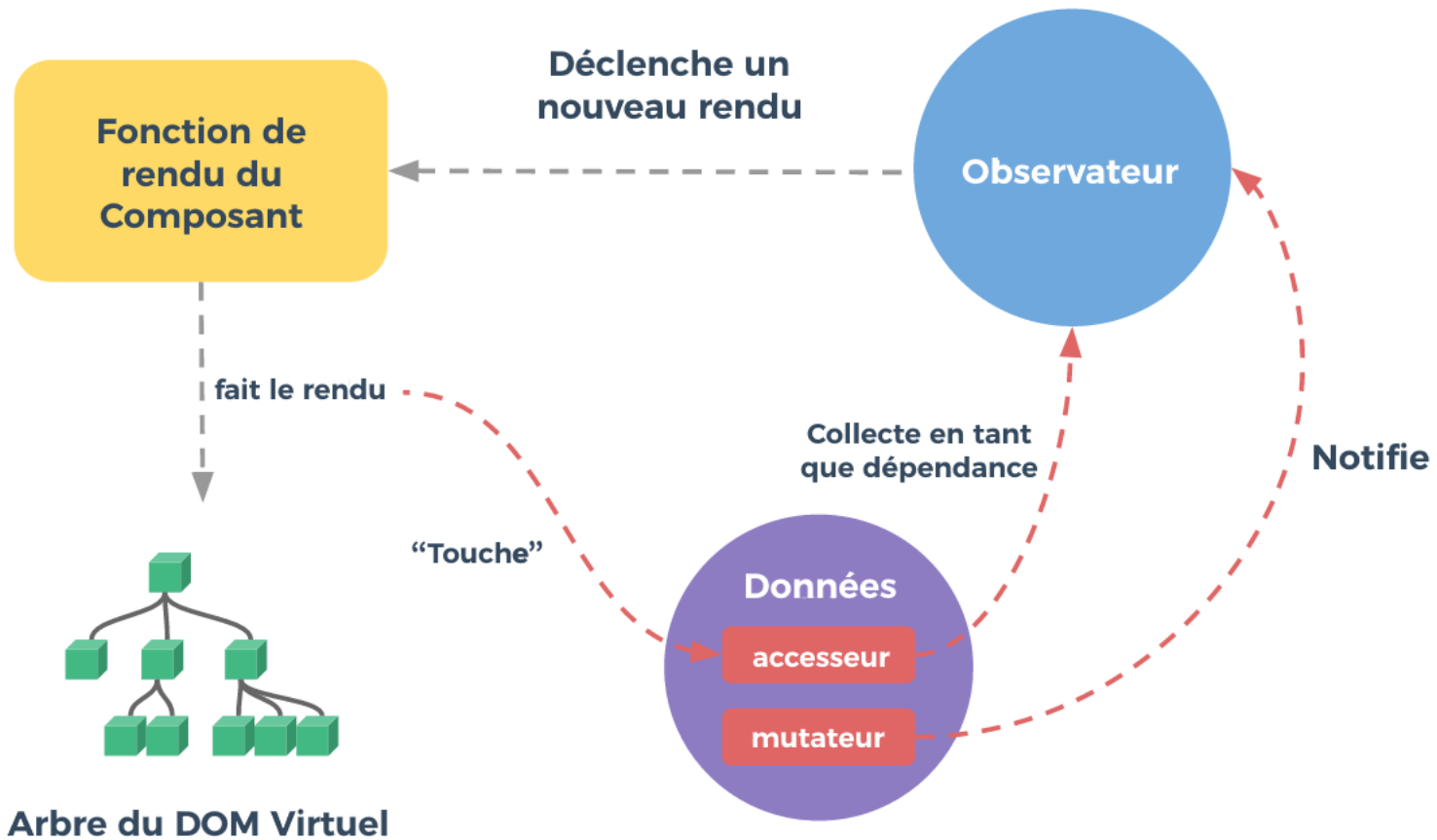
Vue va chercher si l'élément cible a des transition CSS et les applique  
classes de transition

# LE SYSTÈME DE RÉACTIVITÉ

*Un modèle = un object JS*

*une modification = une mise à jour de la  
vue.*

VueJS réactivité





En ES5 le système de getter/setter (accesseur/mutateurs) permet le suivi des dépendances et la notification de changement des propriétés.

Chaque composant a une instance d'observateur qui suit les modifications.

Ajout/suppression :

```
this.$set(this.someObject, 'b', 2)
```

L'object 'data' est comme le schéma de l'état du composant.

## Quelques limitations :

- Pas d'ajout dynamique de nouvelles propriétés réactives dans une instance existante
- Possibilité d'ajouter des ppts réactives a un objet imbriqué.



# RÉSUMÉ VUEJS

- Concepts
  - Virtual dom
  - Reactivité
- Instance de Vue
  - Propriétés réactives (data)
  - Propriétés calculées et observables (methods, computed, watch)

- Templates
  - Directives
  - Transition
  - Evenenements personnalisés
- Composants
  - Slot
  - (monofichier)

# TOOLING

- vue-cli 3.0 : interface en ligne de commandes.  
Initialisation rapide d'un nouveau projet, GUI...
- DevTools 5.0 : débbug des appli Vue et Vuex

# FUTURES ÉVOLUTIONS

- version 3 en typescript (~2019)



# RÉFÉRENCES

- Roadmap vue.js



**EXAMPLE**

Todo List VueJS

[https://jsfiddle.net/yyx990803/4dr2fLb7/?  
utm\\_source=website&utm\\_medium=embed&utm\\_campaign=](https://jsfiddle.net/yyx990803/4dr2fLb7/?utm_source=website&utm_medium=embed&utm_campaign=)