

# Vue.js : composants

**Achref El Mouelhi**

Docteur de l'université d'Aix-Marseille  
Chercheur en programmation par contrainte (IA)  
Ingénieur en génie logiciel

[elmouelhi.achref@gmail.com](mailto:elmouelhi.achref@gmail.com)



# Plan

- 1 Composant inline
- 2 Composant SFC
  - Structure d'un composant SFC
  - Avant de commencer
- 3 v-bind **dans** style
- 4 Importation d'un composant
  - Importer globalement un composant
  - Importer localement un composant
- 5 Interaction entre template/script : binding

# Plan

6

## Interaction entre composants parent/enfant

- Balise d'interaction : <slot/>
- Balise d'interaction : <slot>...</slot>
- Directive v-slot
- slot et #identifiant
- slotProps
- props
- Fonction \$emit ()
- Interaction avec v-model

7

## Cycle de vie d'un composant

8

## Template ref

9

## Valeurs réactives

10

## Bonnes pratiques

# Vue.js

## Remarque

- Pour une meilleure restructuration du projet, utilisons les composants.
- Application **Vue.js** = { composants }

# Vue.js

Déplaçons le contenu de la balise `<div id='app'>` dans une nouvelle section template

```
const App = Vue.createApp({
  data() {
    return {
      // les attributs précédents
    };
  },
  methods: {
    // les méthodes précédentes
  },
  template : `
    <!--
      déplacer ici le contenu de la balise <div id='app'> ici entre backquote
    -->
  `);
}

App.mount('#app');
```

# Vue.js

Déplaçons le contenu de la balise `<div id='app'>` dans une nouvelle section template

```
const App = Vue.createApp({
  data() {
    return {
      // les attributs précédents
    };
  },
  methods: {
    // les méthodes précédentes
  },
  template : `
    <!--
      déplacer ici le contenu de la balise <div id='app'> ici entre backquote
    -->
  `;
});

App.mount('#app');
```

## Remarque

Un composant **Vue.js** = Objet JavaScript (composant inline).

# Vue.js

Nouveau contenu d'index.html

```
<!DOCTYPE html>
<html lang="fr">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Cours Vue.js</title>
  <link rel="stylesheet" href="style.css">
</head>

<body>
  <div id='app'>
  </div>
  <script src="https://unpkg.com/vue@next"></script>
  <script src="script.js">
  </script>
</body>

</html>
```

# Vue.js

Nouveau contenu d'index.html

```
<!DOCTYPE html>
<html lang="fr">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Cours Vue.js</title>
  <link rel="stylesheet" href="style.css">
</head>

<body>
  <div id='app'>
  </div>
  <script src="https://unpkg.com/vue@next"></script>
  <script src="script.js">
  </script>
</body>

</html>
```

Relancez l'application et vérifiez que tout fonctionne comme avant.

# Vue.js

Créons un fichier `HomeView.js` dans `components` pour les données concernant le composant et gardons le reste dans `script.js`

```
export default {
    data() {
        return {
            // les attributs précédents
        };
    },
    methods: {
        // les méthodes précédentes
    },
    template : 
        <!-- les balises précédentes -->
    ;
};
```

# Vue.js

Dans script.js (le point d'entrée), importons puis déclarons le composant

```
import HomeView from "./components/HomeView.js";

const App = Vue.createApp({
  components: {
    'home-view': HomeView
  }
});

App.mount('#app');
```

Dans index.html nous pouvons désormais utiliser le composant

```
<!DOCTYPE html>
<html lang="fr">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale
        =1.0">
    <title>Cours Vue.js</title>
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <div id='app'>
        <home-view></home-view>
    </div>
    <script src="https://unpkg.com/vue@next"></script>
    <script src="script.js" type="module">
    </script>
</body>

</html>
```

# Vue.js

## Remarque

Si le composant est directement attaché au **DOM** (dans une page **HTML**) , il est recommandé de l'écrire en **Kebab Case**.

© Achref EL  
Achref EL

# Vue.js

## Remarque

Si le composant est directement attaché au **DOM** (dans une page **HTML**) , il est recommandé de l'écrire en **Kebab Case**.

Relancez l'application et vérifiez que tout fonctionne comme avant.

# Vue.js

## Rappel

- Application **Vue.js** = { composants }
- Théoriquement : un composant = { code **HTML** + code **CSS** + code **JS** }

© Achref EL MOUADJI

# Vue.js

## Rappel

- Application **Vue.js** = { composants }
- Théoriquement : un composant = { code **HTML** + code **CSS** + code **JS** }

## Composant SFC : Single File Component

- Amélioration des composants inline
- Composant défini dans un fichier .vue
- Conçu pour encapsuler la logique, la structure et les styles dans un seul fichier.

# Vue.js

**Structure d'un composant SFC (Avec VSC, saisissez vueinit et cliquez sur Entrée pour le générer)**

```
<template>
  ...
</template>

<script>
  ...
</script>

<!-- Add scoped attribute to limit CSS to this component only -->
<style scoped>
  ...
</style>
```

# Vue.js

Si nous souhaitons décomposer un composant SFC en trois fichiers HTML, CSS et JS

```
<template src="./template.html"></template>
<style src="./style.css"></style>
<script src="./script.js"></script>
```

# Vue.js

## Remarque

Dans ce cours, nous considérons le projet vue-vite.

© Achref EL MOUADJI

# Vue.js

## Remarque

Dans ce cours, nous considérons le projet vue-vite.

## Rappel : pour créer un projet Vue.js avec Vite

```
npm init vue@latest
```

# Vue.js

## Arborescence d'un projet **Vue.js** créé avec **Vite**

- `node_modules` : contenant les fichiers **Node.js** nécessaires pour un projet **Vue.js**
- `src` : contenant les fichiers sources de l'application
- `package.json` : contenant l'ensemble de dépendance de l'application
- `public` : contenant les fichiers statiques à intégrer systématiquement dans `dist`
- `vite.config.js` : fichier de configuration de **Vite** (outil de construction de projet **Vue.js**)
- `index.html` : point d'entrée de l'application
- `README.md` : permet de fournir une vue d'ensemble de l'application, décrire son objectif et expliquer les fonctionnalités principales.

# Vue.js

## Arborescence d'un projet **Vue.js** créé avec **CLI**

- `node_modules` : contenant les fichiers **Node.js** nécessaires pour un projet **Vue.js**
- `src` : contenant les fichiers sources de l'application
- `package.json` : contenant l'ensemble de dépendance de l'application
- `public` : contenant le point d'entrée de l'application `index.html`
- `vue.config.js` : fichier chargeant les services de **Vue-CLI**
- `README.md` : permet de fournir une vue d'ensemble de l'application, décrire son objectif et expliquer les fonctionnalités principales.

# Vue.js

## Que contient `src` ?

- `assets` : unique dossier accessible aux visiteurs et contenant logo, CSS, images, sons...
- `components` : contient un composant : `HelloWorld.vue`
- `main.js` (ou `main.ts`) : point d'entrée permettant de charger l'application **Vue.js** dans l'élément ayant l'identifiant `app`
- `App.vue` : composant principal référencé par `main.js` et utilisant le composant `HelloWorld.vue`

# Vue.js

Considérons le code suivant pour `src/style.css`

```
#app {  
  font-family: Avenir, Helvetica, Arial, sans-serif;  
  -webkit-font-smoothing: antialiased;  
  -moz-osx-font-smoothing: grayscale;  
  text-align: center;  
  color: #2c3e50;  
}
```

# Vue.js

Considérons aussi le code simplifié suivant HelloWorld.vue

```
<template>
  <div class="hello">
    Hello world
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
}
</script>

<style scoped>
</style>
```

# Vue.js

Considérons aussi le code simplifié suivant `HelloWorld.vue`

```
<template>
  <div class="hello">
    Hello world
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
}
</script>

<style scoped>
</style>
```

## Explication

- La partie `template` contient le code **HTML** à afficher.
- La partie `script` exporte le composant pour pouvoir l'utiliser ailleurs : l'attribut `name` permet de définir le nom à utiliser pour l'importation.

# Vue.js

Et le code suivant pour App.vue

```
<template>
  
  
  <HelloWorld />
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
</script>

<style scoped>
.logo {
  height: 6em;
  will-change: filter;
  transition: filter 300ms;
}
</style>
```

# Vue.js

Relancer l'application et vérifier que Hello world s'affiche au démarrage de l'application.

# Vue.js

## v-bind dans style

Depuis **Vue.js 3.2**, il est possible d'utiliser `v-bind dans style` pour lier la valeur d'une propriété **CSS** à une variable définie dans `data` de la partie `script`.

# Vue.js

## v-bind dans style

Depuis **Vue.js 3.2**, il est possible d'utiliser `v-bind dans style` pour lier la valeur d'une propriété **CSS** à une variable définie dans `data` de la partie `script`.

Ajoutons la propriété suivante dans `data`

```
bgColor: 'red'
```

# Vue.js

## v-bind dans style

Depuis **Vue.js 3.2**, il est possible d'utiliser **v-bind dans style** pour lier la valeur d'une propriété **CSS** à une variable définie dans **data de la partie script**.

Ajoutons la propriété suivante dans **data**

```
bgColor: 'red'
```

Utilisons **v-bind dans style** pour récupérer la valeur

```
div {  
  background-color: v-bind('bgColor');  
}
```

# Vue.js

## Deux types de composant

- Prédéfinis : comme le composant racine `App` et les composants intégrés fournis par **Vue.js** tels que `<transition>`, `<component>`...
- Personnalisés : à définir par le développeur et dont le nom doit être composé de plusieurs mots (`HelloWorld` par exemple)

# Vue.js

## Deux types de composant

- Prédéfinis : comme le composant racine `App` et les composants intégrés fournis par **Vue.js** tels que `<transition>`, `<component>`...
- Personnalisés : à définir par le développeur et dont le nom doit être composé de plusieurs mots (`HelloWorld` par exemple)

### Remarque

Pour utiliser un composant, il faut l'importer.

# Vue.js

## Deux types d'importation

- Locale : le composant ne pourra être utilisé que dans le composant qui l'importe.
- Globale : le composant peut être utilisé dans toute l'application

© Achref EL MOUADJI

# Vue.js

## Deux types d'importation

- Locale : le composant ne pourra être utilisé que dans le composant qui l'importe.
- Globale : le composant peut être utilisé dans toute l'application

## Terminologie

- **Composant parent** : composant qui importe un autre composant.
- **Composant enfant** : composant importé par un autre composant.

# Vue.js

Dans App.vue, gardons l'utilisation de <HelloWorld/> dans template et supprimons (ou commentons) son importation et déclaration dans script

```
<template>
  
  
  <HelloWorld />
</template>

<script>
// import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    // HelloWorld
  }
}
</script>

<style scoped>
.logo {
  height: 6em;
  will-change: filter;
  transition: filter 300ms;
}
</style>
```

# Vue.js

Importons et déclarons globalement le composant HelloWorld dans main.js

```
import { createApp } from 'vue'  
import App from './App.vue'  
import HelloWorld from './components/HelloWorld.vue'  
  
createApp(App)  
  .component('HelloWorld', HelloWorld)  
  .mount('#app')
```

# Vue.js

Importons et déclarons globalement le composant `HelloWorld` dans `main.js`

```
import { createApp } from 'vue'  
import App from './App.vue'  
import HelloWorld from './components/HelloWorld.vue'  
  
createApp(App)  
  .component('HelloWorld', HelloWorld)  
  .mount('#app')
```

## Remarques

- Le composant `HelloWorld` peut être utilisé par tous les composants de l'application sans l'importer localement.
- Possibilité d'importer globalement plusieurs composants.

# Vue.js

**Remettons dans main.js le code initial (en supprimant l'importation et la déclaration de HelloWorld)**

```
import { createApp } from 'vue'  
import App from './App.vue'  
  
createApp(App).mount('#app')
```

# Vue.js

Dans App.vue, importons localement HelloWorld et déclarons le dans components

```
<template>
  
  
  <HelloWorld />
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
</script>

<style scoped>
.logo {
  height: 6em;
  will-change: filter;
  transition: filter 300ms;
}
</style>
```

# Vue.js

Relancer l'application et vérifier que Hello world s'affiche au démarrage de l'application.

# Vue.js

Rappelons les 2 modes de liaison template/script d'un même composant

- One way binding
  - Interpolation {{ ... }}
  - Attribute binding v-bind
  - Event binding v-on
- Two way binding : v-model

# Vue.js

Déclarons la fonction **data** dans la partie script du composant HelloWorld

```
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world"
    }
  }
}
```

# Vue.js

Déclarons la fonction `data` dans la partie script du composant `HelloWorld`

```
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world"
    }
  }
}
```

Utilisons l'interpolation pour afficher le contenu de `msg` dans la partie template du même composant

```
<div class="hello">
  {{ msg }}
</div>
```

# Vue.js

Relancer l'application et vérifier que Hello world s'affiche au démarrage de l'application.

# Vue.js

## Formes d'interaction entre composants parent-enfant

- Ajouter le sélecteur d'un premier composant dans le template d'un deuxième composant
  - on appelle le premier composant : composant fils
  - on appelle le deuxième composant : composant parent
- Plusieurs formes de transmission de données par
  - Du parent vers l'enfant : via la balise `slot` (transclusion) ou la propriété `props`
  - De l'enfant vers le parent : via la méthode `$emit()`

# Vue.js

## Les slots en Vue.js

- Mécanisme de **transclusion** : contenu injecté dans un composant enfant.
- Très utile pour créer des composants réutilisables et flexibles.
- Types principaux :
  - Slot par défaut
  - Slot nommé
  - Slot dynamique
  - Scoped slot

# Vue.js

## Types de slots

### Slot par défaut :

- Permet d'injecter du contenu de base.
- Utilisation : `<slot></slot>` dans l'enfant, contenu direct dans le parent.

### Slot nommé :

- Permet de définir plusieurs zones personnalisables.
- Utilisation : `<slot name="...">` côté enfant et `<template #nom>` côté parent.

### Slot dynamique :

- Le nom du slot est une variable.
- Utilisation : `<slot :name="slotName">`.

### Scoped slot (slot à portée) :

- Le composant enfant expose des données au parent.
- Le parent accède à ces données via une structure `<template #default="{ data }">`.

# Vue.js

**Vue.js nous permet d'utiliser une balise auto-fermante ou une balise fermante pour nos composants (template de App.vue)**

```
<template>
  
  
  <HelloWorld/>
  <HelloWorld></HelloWorld>
</template>
```



# Vue.js

**Vue.js nous permet d'utiliser une balise auto-fermante ou une balise fermante pour nos composants** (`template de App.vue`)

```
<template>
  
  
  <HelloWorld/>
  <HelloWorld></HelloWorld>
</template>
```



Relancer l'application et vérifier que `Hello world` s'affiche deux fois au démarrage de l'application.

# Vue.js

Et si le composant parent (ici App) envoie des données entre les balises ouvrante et fermante de HelloWorld

```
<template>
  
  
  <HelloWorld/>
  <HelloWorld>Wick</HelloWorld>
</template>
```

# Vue.js

Et si le composant parent (ici App) envoie des données entre les balises ouvrante et fermante de HelloWorld

```
<template>
  
  
  <HelloWorld/>
  <HelloWorld>Wick</HelloWorld>
</template>
```

## Constat

Le contenu ne s'affiche pas.

# Vue.js

Et si le composant parent (ici App) envoie des données entre les balises ouvrante et fermante de HelloWorld

```
<template>
  
  
  <HelloWorld/>
  <HelloWorld>Wick</HelloWorld>
</template>
```

## Constat

Le contenu ne s'affiche pas.

## Question

Comment récupérer et afficher les données envoyées par le composant parent ?

# Vue.js

Pour récupérer des données envoyées par le composant parent entre les balises ouvrante et fermante, on utilise la balise slot (contenu du template du composant HelloWorld)

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot/>
  </div>
</template>
```

# Vue.js

Pour récupérer des données envoyées par le composant parent entre les balises ouvrante et fermante, on utilise la balise slot (contenu du template du composant HelloWorld)

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot/>
  </div>
</template>
```

## Question

Comment définir un contenu par défaut pour le cas où le parent n'envoie pas de données ?

# Vue.js

Pour définir un contenu par défaut pour la balise slot (**contenu du template du composant HelloWorld**)

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot>Doe</slot>
  </div>
</template>
```



# Vue.js

Pour définir un contenu par défaut pour la balise slot (**contenu du template du composant HelloWorld**)

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot>Doe</slot>
  </div>
</template>
```



Relancer l'application et vérifier que Hello Doe et Hello Wick s'affichent au démarrage de l'application.

# Vue.js

## Question

Comment envoyer plusieurs données du parent vers l'enfant entre les balises ouvrante et fermante ?

© Achref EL MOUELHI ©

# Vue.js

## Question

Comment envoyer plusieurs données du parent vers l'enfant entre les balises ouvrante et fermante ?

Pour cela, il faut les séparer dans des templates différents et utiliser `v-slot` (contenu du template du composant App)

```
<template>
  
  
  <HelloWorld/>
  <HelloWorld>
    <template v-slot:nom>Wick</template>
    <template v-slot:age>45</template>
  </HelloWorld>
</template>
```

# Vue.js

Pour récupérer les données envoyées par le parent, on utilise la balise slot avec l'attribut name (contenu du template du composant HelloWorld)

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot name="nom">Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
</template>
```

# Vue.js

L'écriture précédente peut être simplifiée avec les identifiants (commençant par #)

```
<template>
  
  
  <HelloWorld/>
  <HelloWorld>
    <template #nom>Wick</template>
    <template #age>45</template>
  </HelloWorld>
</template>
```

# Vue.js

Rien à changer dans template de HelloWorld

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot name="nom">Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
</template>
```

# Vue.js

On peut aussi utiliser un identifiant default avant de simplifier la récupération

```
<template>
  
  
  <HelloWorld/>
  <HelloWorld>
    <template #default>Wick</template>
    <template #age>45</template>
  </HelloWorld>
</template>
```

# Vue.js

Le default sera récupéré sans avoir besoin d'utiliser name dans slot

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot>Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
</template>
```

## Exercice : primeur-produit

### • Première partie

- Créez deux composants PrimeurComponent et ProduitComponent : PrimeurComponent est le composant parent des composants ProduitComponent
- Le composant PrimeurComponent a un attribut produits : à déclarer dans la fonction data (voir ci-dessous).
- Utilisez v-for pour créer autant de composants ProduitComponent que d'éléments dans le tableau produits : chaque composant produit reçoit le nom qu'il doit afficher.

### • Deuxième partie

- Dans PrimeurComponent, ajoutez un bouton ajouter et trois zones de saisie : une pour le nom, une pour le prix et une pour la quantité.
- En cliquant sur le bouton ajouter, un nouveau composant produit s'ajoute (s'affiche) au composant (dans la page) avec le nom saisi par l'utilisateur.

Attribut à déclarer dans la fonction data dans PrimeurComponent.vue

```
produits: [
  { nom: "banane", prix: 3, quantite: 10 },
  { nom: "fraise", prix: 10, quantite: 20 },
  { nom: "poivron", prix: 5, quantite: 10 }
]
```

# Vue.js

## slotProps

Permettent de définir des noms pour les données passées du composant parent au composant enfant via un `slot` pour qu'elles soient utilisées d'une manière dynamique.

© Achref EL MOUELLI

# Vue.js

## slotProps

Permettent de définir des noms pour les données passées du composant parent au composant enfant via un `slot` pour qu'elles soient utilisées d'une manière dynamique.

### Exemple avec slotProps (contenu d'`App.vue`)

```
<template>
  
  
  <HelloWorld v-slot="slotProps">
    {{ slotProps.nom }} {{ slotProps.prenom }}
  </HelloWorld>
</template>
```

# Vue.js

## Contenu du template du composant HelloWorld

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot :nom="lastname" :prenom="firstname"></slot>
  </div>
</template>
```



# Vue.js

## Contenu du template du composant HelloWorld

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot :nom="lastname" :prenom="firstname"></slot>
  </div>
</template>
```

Sans oublier de déclarer firstname et lastname dans data de HelloWorld

```
firstname: 'John',
lastname: 'Abruzzi',
```

# Vue.js

Le contenu d'App.vue peut être simplifié en utilisant la déstructuration

```
<template>
  
  
  <HelloWorld v-slot="{ nom, prenom }">
    {{ nom }} {{ prenom }}
  </HelloWorld>
</template>
```

# Vue.js

## Récapitulons

Grâce à la balise `<slot/>`, on récupère une valeur envoyée par le parent entre la balise ouvrante et fermante de l'enfant.

© Achref EL MOUELHI

# Vue.js

## Récapitulons

Grâce à la balise `<slot/>`, on récupère une valeur envoyée par le parent entre la balise ouvrante et fermante de l'enfant.

## Question

Et si nous voulions passer des valeurs comme valeur d'attribut ?

# Vue.js

## Récapitulons

Grâce à la balise `<slot/>`, on récupère une valeur envoyée par le parent entre la balise ouvrante et fermante de l'enfant.

## Question

Et si nous voulions passer des valeurs comme valeur d'attribut ?

## Réponse

Nous pourrons utiliser la propriété `props`.

# Vue.js

Ajoutons un attribut ville à la balise HelloWorld dans le template de App.vue

```
<template>
  
  
  <HelloWorld ville="Marseille"/>
  <HelloWorld>
    <template v-slot:nom>Wick</template>
    <template v-slot:age>45</template>
  </HelloWorld>
</template>
```

# Vue.js

Pour récupérer la valeur de la propriété ville, il faut commencer par la déclarer dans une section props de script du composant HelloWorld

```
<script>
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world"
    }
  },
  props: ['ville']
}
</script>
```

# Vue.js

Pour afficher la valeur de la propriété ville, on utilise l'interpolation dans la partie template

```
<template>
  <div class="hello">
    {{ msg }} from {{ ville }}
  </div>
  <div>
    Hello <slot name="nom">Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
</template>
```

# Vue.js

On peut aussi typer la props

```
<script>
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world"
    }
  },
  props: {
    ville: String
  }
}
</script>
```

# Vue.js

Les types autorisés sont les types prédéfinis en **JavaScript**

- Number
- String
- Boolean
- Array
- Date
- ...

# Vue.js

On peut aussi spécifier une valeur par défaut pour la props

```
<script>
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world"
    }
  },
  props: {
    ville: {
      type: String,
      default: 'Paris'
    }
  }
}
</script>
```

# Vue.js

## Remarque

- Impossible de modifier la valeur d'une variable déclarée `props`
- Cependant, il est possible de déclarer une variable dans `data` et d'initialiser sa valeur avec la valeur de la variable déclarée dans `props`

# Vue.js

## Exercice : primeur-produit

Reprendons l'exercice précédent et affichons le prix et la quantité disponible de chaque produit en utilisant props.

# Vue.js

## Objectif

Transmettre de données depuis un composant enfant vers un composant parent.

# Vue.js

Dans le template de HelloWorld, ajoutons le contenu suivant

```
<div>
  <label for="nom">Nom</label>
  <input type="text" id="nom" v-model="nom">
  <button @click="$emit('sendData', this.nom)">
    Envoyer
  </button>
</div>
```



# Vue.js

Dans le template de HelloWorld, ajoutons le contenu suivant

```
<div>
  <label for="nom">Nom</label>
  <input type="text" id="nom" v-model="nom">
  <button @click="$emit('sendData', this.nom)">
    Envoyer
  </button>
</div>
```



Sans oublier de déclarer nom comme attribut de la fonction data

```
nom: null
```

# Vue.js

## Explication

- En cliquant sur le bouton, le composant enfant (HelloWorld) émet un évènement à son parent (App).
- Par conséquent, l'évènement sendData (à définir dans la balise HelloWorld de App déclenchera l'exécution d'une fonction à définir dans App..
- Le nom sera envoyé comme paramètre.

# Vue.js

Dans le template de App, ajoutons un évènement de type sendData (écrit en kebab case) dans HelloWorld qui, une fois déclenché, la fonction afficherBonjour() sera exécutée

```
<template>
  
  
  <p>Bonjour {{ nom ?? "Doe" }}</p>
  <HelloWorld @send-data="afficherBonjour" ville="Marseille"/>
  <!-- <HelloWorld />
  <HelloWorld>
    <template v-slot:nom>Wick</template>
    <template v-slot:age>45</template>
  </HelloWorld> -->
  <PrimeurComponent />
</template>
```

# Vue.js

N'oublions pas de définir nom **dans** data et afficherBonjour **dans** methods

```
<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld,
  },
  data() {
    return {
      nom: null
    }
  },
  methods: {
    afficherBonjour(nom) {
      this.nom = nom;
    }
  }
}
</script>
```

# Vue.js

Dans le template de HelloWorld, nous pouvons aussi remplacer

```
<button @click="$emit('sendData', this.nom)">Envoyer</button>
```

© Achref EL MOUELHI ©

# Vue.js

Dans le template de HelloWorld, nous pouvons aussi remplacer

```
<button @click="$emit('sendData', this.nom)">Envoyer</button>
```

Par

```
<button @click="envoyer">Envoyer</button>
```

# Vue.js

Dans le template de HelloWorld, nous pouvons aussi remplacer

```
<button @click="$emit('sendData', this.nom)">Envoyer</button>
```

Par

```
<button @click="envoyer">Envoyer</button>
```

Et utiliser \$emit dans envoyer (à définir dans script)

```
methods: {
  envoyer() {
    this.$emit('sendData', this.nom)
  }
}
```

# Vue.js

Dans la console, si un warn s'affiche (Extraneous non-emits event listeners), alors ajouter le code suivant dans script de HelloWorld

```
emits: ['sendData']
```

# Vue.js

Dans la console, si un warn s'affiche (**Extraneous non-emits event listeners**), alors ajouter le code suivant dans script de HelloWorld

```
emits: ['sendData']
```

## Explication

Ceci permettra d'écouter uniquement les évènements de type sendData.

# Vue.js

## Question

Pourquoi faut-il minimiser les données émises de l'enfant au parent ?

© Achref EL MOUELHI ©

# Vue.js

## Question

Pourquoi faut-il minimiser les données émises de l'enfant au parent ?

## Réponse

- **Encapsulation et Responsabilités** : un composant enfant ne doit pas connaître ou gérer plus de données qu'il ne doit.
- **Performance** : envoyer des objets volumineux ou des structures complexes inutilement peut impacter les performances, surtout dans les composants fréquemment réutilisés.
- **Facilité de maintenance** : Limiter les données transmises réduit le risque de couplage fort entre parent et enfant.

# Vue.js

## Exercice 1 : primeur-produit

- Dans PrimeurComponent, déclarez un attribut total dans data.
- Dans ProduitComponent, ajoutez une zone de saisie et un bouton.
- En choisissant une quantité et appuyant sur le bouton,
  - le total sera recalculé et affiché,
  - la quantité en stock sera mise à jour
  - et le bouton sera désactivé.

# Vue.js

## Exercice 2

- Considérons deux composants ClasseComponent et EleveComponent.
- Chaque composant EleveComponent aura un champ texte pour saisir une note et un bouton pour envoyer la valeur au composant ClasseComponent.
- Le bouton sera désactivé après envoi.
- Chaque fois que le composant ClasseComponent reçoit la note d'un EleveComponent, il recalcule la moyenne et il l'affiche.
- Il y aura autant de EleveComponent (composant enfant) dans ClasseComponent (composant parent) que d'éléments dans le tableau noms (voir ci-dessous).

**Attribut noms à déclarer dans data de ClasseComponent**

```
noms: ['Wick', 'Hoffman', 'Abruzzi']
```

# Vue.js

## Exercice 3 : clavier-touche (Simulation d'un clavier virtuel)

- Créez deux composants ClavierComponent et ToucheComponent.
- Le composant ClavierComponent a un attribut lettres (voir ci-dessous)
- Le composant ToucheComponent a un attribut value recevant une valeur du tableau lettres (un composant fils ToucheComponent pour chaque valeur du tableau lettres).
- Chaque ToucheComponent affiche la lettre qu'il a reçue sur un bouton.
- En cliquant sur ce bouton, la lettre s'affiche (à la suite des autres) dans une balise textarea définie dans le composant ClavierComponent.

### Contenu du tableau lettres

```
lettres: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

# Vue.js

## Exercice : pays-ville

### • Première partie

- Créez deux composants PaysComponent et VilleComponent : PaysComponent est le composant parent des composants VilleComponent.
- Le composant PaysComponent a un attribut villes : à déclarer dans la fonction data (voir ci-dessous). Le contenu du tableau villes est affiché dans une balise div dans le template de PaysComponent.
- Utilisez v-for pour créer autant de composants VilleComponent que d'éléments dans le tableau villes : chaque composant VilleComponent reçoit le nom d'une ville qu'il doit afficher dans une balise input.
- En modifiant la valeur de l'input, la valeur doit être mise à jour dans le template de PaysComponent.



Attribut à déclarer dans la fonction data dans PaysComponent.vue

```
villes: ['Marseille', 'Lyon', 'Paris']
```

# Vue.js

Première solution : code de PaysComponent

```
<template>
  <h1>Pays</h1>
  <div>
    <template v-for="(item) in villes">
      {{ item }} &nbsp;
    </template>
  </div>
  <VilleComponent v-for="(item, index) in villes" :ville="item" @send-data="updateVille(index
    , $event)" />
</template>
<script>
import VilleComponent from './VilleComponent.vue'
export default {
  components: {
    VilleComponent
  },
  data() {
    return {
      villes: ['Marseille', 'Lyon', 'Paris']
    }
  },
  methods: {
    updateVille(ind, ville) {
      this.villes[ind] = ville
    }
  }
}
</script>
```

# Vue.js

Première solution : code de VilleComponent

```
<template>
  <h2>Ville</h2>
  <div>
    <input :value="ville" @input="$emit('sendData', $event.target.value)" />
  </div>
</template>

<script>
export default {
  props: ['ville'],
  emits: ['sendData'],
}
</script>
```

# Vue.js

## Deuxième solution

- Utiliser `v-model` pour communiquer des données aux composants enfants
- Utiliser `modelValue` pour référencer la valeur envoyée via `v-model`
- Utiliser `update:modelValue` pour émettre le changement de valeur au parent

# Vue.js

## Deuxième solution : code de PaysComponent

```
<template>
  <h1>Pays : {{ villes }}</h1>
  <template v-for="(item, i) in villes">
    <VilleComponent v-model="villes[i]" />
  </template>
</template>

<script>
import VilleComponent from './VilleComponent.vue'

export default {
  components: {
    VilleComponent
  },
  data() {
    return {
      villes: ['Marseille', 'Lyon', 'Paris']
    }
  },
</script>
```

# Vue.js

Rappel : l'écriture suivante, en JavaScript, ne permet pas de modifier le tableau mais ne génère pas d'erreur

```
villes = ['Marseille', 'Lyon', 'Paris']

for (let item in villes) {
    item = ''
}

console.log(villes);
// affiche [ 'Marseille', 'Lyon', 'Paris' ]
```

# Vue.js

Rappel : l'écriture suivante, en JavaScript, ne permet pas de modifier le tableau mais ne génère pas d'erreur

```
villes = ['Marseille', 'Lyon', 'Paris']

for (let item in villes) {
    item = ''
}

console.log(villes);
// affiche [ 'Marseille', 'Lyon', 'Paris' ]
```

En revanche, les boucles indexées permettent de modifier le tableau

```
for (let i = 0; i < villes.length; i++) {
    villes[i] = ''

}

console.log(villes);
// affiche [ '', '', '' ]
```

# Vue.js

Par conséquent, l'écriture suivante en Vue.js déclenche une erreur car v-model ne peut jamais modifier la valeur d'une boucle non indexée

```
<template v-for="item in villes">
  <VilleComponent v-model="item" />
</template>
```

© Achref EL MOUADJI

# Vue.js

Par conséquent, l'écriture suivante en Vue.js déclenche une erreur car v-model ne peut jamais modifier la valeur d'une boucle non indexée

```
<template v-for="item in villes">
  <VilleComponent v-model="item" />
</template>
```

Il suffit donc de la remplacer par

```
<template v-for="(item, i) in villes">
  <VilleComponent v-model="villes[i]" />
</template>
```

# Vue.js

## Deuxième solution : code de VilleComponent

```
<template>
  <h2>Ville</h2>
  <div>
    <input :value="modelValue" @input="$emit('update:modelValue', $event.target.value)" />
  </div>
</template>

<script>
export default {
  props: ['modelValue'],
  emits: ['update:modelValue'],
}
</script>
```

# Vue.js

## Exercice : pays-ville

### • Deuxième partie

- Dans le composant PaysComponent, ajoutons un attribut codesPostaux : à déclarer dans la fonction data (voir ci-dessous). Chaque élément d'indice *i* du tableau codesPostaux sera affiché avec l'élément d'indice *i* de villes.
- Le composant PaysComponent transmet à chaque enfant (VilleComponent) une ville et un code postal. Les deux valeurs seront affichées dans deux <input> différents.
- En modifiant la valeur de l'input, la valeur doit être mise à jour dans le template de PaysComponent.



Attributs à déclarer dans la fonction data dans PaysComponent.vue

```
ville: ['Marseille', 'Lyon', 'Paris'],
codesPostaux: ['13000', '69000', '75000']
```

# Vue.js

Solution : code de PaysComponent

```
<template>
  <h1>Pays</h1>
  <div>
    <template v-for="(item, i) in villes">
      {{ item }} - {{ codesPostaux[i] }} &nbsp;
    </template>
  </div>
  <template v-for="(item, i) in villes">
    <VilleComponent v-model:ville="villes[i]" v-model:codePostal="codesPostaux[i]" />
  </template>
</template>

<script>
import VilleComponent from './VilleComponent.vue'
export default {
  components: {
    VilleComponent
  },
  data() {
    return {
      villes: ['Marseille', 'Lyon', 'Paris'],
      codesPostaux: ['13000', '69000', '75000']
    }
  },
}
</script>
```

# Vue.js

## Solution : code de VilleComponent

```
<template>
  <h2>Ville</h2>
  <div>
    Ville : <input :value="ville" @input="$emit('update:ville', $event.target.value)" />
    Code postal : <input :value="codePostal" @input="$emit('update:codePostal', $event.
      target.value)" />
  </div>
</template>

<script>
export default {
  props: ['ville', 'codePostal'],
  emits: ['update:ville', 'update:codePostal'],
}
</script>
```

# Vue.js

## Lifecycle hooks

- Tout composant Vue.js a un cycle de vie qui commence à la création.
- Vue.js nous a préparé une méthode pour chaque phase du cycle de vie : **Lifecycle hooks**.

# Vue.js

## Différentes méthodes (hooks) de cycle de vie d'un composant **Vue.js** (dans l'ordre)

- `setup` (**Vue.js 3**) : appelé avant la création du composant et après la résolution de toutes les `props` (à voir dans un prochain chapitre).
- `beforeCreate` : appelée avant la création du composant.
- `created` : appelée après la création du composant.
- `beforeMount` : appelée avant chaque que le composant ne soit attaché au **DOM**.
- `mounted` : appelée après attachement du composant et tous ses composants enfants au **DOM**.
- `beforeUpdate` : appelée avant une modification d'un élément du composant ou de ses enfants.
- `updated` : appelée après modification.
- `beforeUnmount` : appelée avant destruction du composant.
- `unmounted` : appelée après destruction du composant.

# Vue.js

## Remarques

- `created` suffit pour manipuler les données dès qu'elles sont prêtes.
- `mounted` suffit pour manipuler le **DOM** réel, utiliser des `refs`...
- Mais, `beforeCreate` et `beforeMount` existent pour des cas de diagnostic, optimisation avancée, ou frameworks/plugins qui doivent intervenir avant certaines phases internes (appliquer des transformations avant même l'initialisation des données, enregistrer des logs ou démarrer un timer avant affichage...).

# Vue.js

Ajoutons toutes les méthodes hooks dans la partie script de HelloWorld.vue (ne les placez pas dans methods)

```
beforeCreate() {
  console.log('before create')
},
created() {
  console.log('created')
},

beforeMount() {
  console.log('before mount')
},
mounted() {
  console.log('mounted')
},

beforeUpdate() {
  console.log('before update')
},
updated() {
  console.log('updated')
},

beforeUnmount() {
  console.log('before unmount')
},
unmounted() {
  console.log('unmounted')
},
```

# Vue.js

**Allez à la route associée au composant CycleVieComponent et vérifier l'affichage des messages suivants**

**before create**

**created**

**before mount**

**mounted**

# Vue.js

**Allez à la route associée au composant CycleVieComponent et vérifier l'affichage des messages suivants**

**before create**

**created**

**before mount**

**mounted**

**Saisissez une valeur dans la zone de saisie et vérifiez l'apparition des deux messages suivants**

**before update**

**updated**

# Vue.js

## Question 1

Quelle est la différence entre `created` et `mounted` ?

© Achref EL MOUELHI ©

# Vue.js

## Question 1

Quelle est la différence entre `created` et `mounted` ?

### created

- S'exécute au tout début du cycle de vie,
- S'exécute une seule fois,
- Impossible de manipuler le **DOM**,
- Utilisé généralement pour récupérer des données à partir d'une API backend.

### mounted

- S'exécute après `created`,
- Peut-être exécutée plusieurs fois,
- La manipulation de **DOM** est possible,
- N'est pas utilisée pour la récupération de données.

## Manipulation de DOM : exemple

Placer le curseur dans un `input` du composant en question.

© Achref EL MOUADJI

# Vue.js

## Manipulation de **DOM** : exemple

Placer le curseur dans un `input` du composant en question.

Comment faire pour manipuler le **DOM** dans la section `script` ?

Utiliser `ref` (à voir dans la section suivante)

# Vue.js

## ref

- Attribut spécial pour les éléments **HTML** définis dans un composant.
- Permettant d'obtenir une référence directe sur un élément **HTML**.

Dans le template de HelloWorld, définissons une référence sur la zone de saisie

```
<template>
  <div class="hello">
    {{ msg }} from {{ ville }}
  </div>
  <div>
    Hello <slot name="nom">Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
  <div>
    <label for="nom">Nom</label>
    <input type="text" id="nom" v-model="nom" ref="name">
    <button @click="$emit('sendData', this.nom)">Envoyer</button>
  </div>
</template>
```

Dans le template de HelloWorld, définissons une référence sur la zone de saisie

```
<template>
  <div class="hello">
    {{ msg }} from {{ ville }}
  </div>
  <div>
    Hello <slot name="nom">Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
  <div>
    <label for="nom">Nom</label>
    <input type="text" id="nom" v-model="nom" ref="name">
    <button @click="$emit('sendData', this.nom)">Envoyer</button>
  </div>
</template>
```

## Objectif

Placer le curseur et attacher un placeholder à la zone de saisie.

# Vue.js

Ajoutons la méthode `mounted` pour placer le curseur au chargement du DOM

```
mounted() {  
    this.$refs.name.placeholder = 'Votre nom';  
    this.$refs.name.focus();  
}
```

© Achref EL MOUELHI ©

# Vue.js

Ajoutons la méthode `mounted` pour placer le curseur au chargement du DOM

```
mounted() {  
    this.$refs.name.placeholder = 'Votre nom';  
    this.$refs.name.focus();  
}
```

## Constat

`$refs` est un objet contenant toutes références.

# Vue.js

Ajoutons la méthode `mounted` pour placer le curseur au chargement du DOM

```
mounted() {  
    this.$refs.name.placeholder = 'Votre nom';  
    this.$refs.name.focus();  
}
```

## Constat

`$refs` est un objet contenant toutes références.

## Remarque

La méthode `created` ne pourra pas être utilisé car le composant ne sera pas encore attaché au DOM.

# Vue.js

Nous pouvons définir une référence sur un composant enfant (code à ajouter dans template de App.vue)

```
<HelloWorld @send-data="afficherBonjour" ville="Marseille" ref="enfant"/>
```

© Achref EL MOUELHI ©

# Vue.js

Nous pouvons définir une référence sur un composant enfant (code à ajouter dans template de App.vue)

```
<HelloWorld @send-data="afficherBonjour" ville="Marseille" ref="enfant"/>
```

Ajoutons également un bouton qui nous permettra d'accéder à une des data de l'enfant et la modifier

```
<div>
  <button @click="traduire">
    Traduire en français
  </button>
</div>
```

# Vue.js

Nous pouvons définir une référence sur un composant enfant (code à ajouter dans template de App.vue)

```
<HelloWorld @send-data="afficherBonjour" ville="Marseille" ref="enfant"/>
```

Ajoutons également un bouton qui nous permettra d'accéder à une des data de l'enfant et la modifier

```
<div>
  <button @click="traduire">
    Traduire en français
  </button>
</div>
```

Pour modifier une propriété de l'enfant, définissons la méthode traduire dans methods de App.vue

```
traduire() {
  this.$refs.enfant.msg = 'Bonjour tout le monde'
}
```

# Vue.js

Créons un composant `ReactiveValue` avec le code suivant

```
<template>
    <p>{{ valeur1 }} + {{ valeur2 }} = {{ resultat }}</p>
</template>

<script>
export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
        this.resultat = this.valeur1 + this.valeur2
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

# Vue.js

Créons un composant `ReactiveValue` avec le code suivant

```
<template>
    <p>{{ valeur1 }} + {{ valeur2 }} = {{ resultat }}</p>
</template>

<script>
export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
        this.resultat = this.valeur1 + this.valeur2
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

## Question

Que sera le résultat de ce code ?

# Vue.js

## Résultat attendu

- Au chargement de la page  $2 + 3 = 5$
- 5 secondes plus tard, 3 sera remplacé par 13
- Contenu attendu :  $2 + 13 = 15$

## Résultat obtenu

- Au chargement de la page  $2 + 3 = 5$
- 5 secondes plus tard, 3 est remplacé par 13
- Contenu affiché :  $2 + 13 = 5$

# Vue.js

## Résultat attendu

- Au chargement de la page  $2 + 3 = 5$
- 5 secondes plus tard, 3 sera remplacé par 13
- Contenu attendu :  $2 + 13 = 15$

## Résultat obtenu

- Au chargement de la page  $2 + 3 = 5$
- 5 secondes plus tard, 3 est remplacé par 13
- Contenu affiché :  $2 + 13 = 5$

## Conclusion

Quand `valeur1` ou `valeur2` change, `resultat` ne se met pas à jour.

# Vue.js

## Rappel

- Les valeurs de `data()` sont réactives.
- Mais la liaison entre des propriétés ne se fait pas automatiquement.

© Achref EL MOUELHIDI

# Vue.js

## Rappel

- Les valeurs de `data()` sont réactives.
- Mais la liaison entre des propriétés ne se fait pas automatiquement.

## Plusieurs solutions possibles

- Première solution : utiliser la fonction `watchEffect` (**Vue.js 3**)
- Deuxième solution : utiliser la fonction `computed` (**Vue.js 3**)
- Troisième solution : utiliser l'option `computed` (**Vue.js 2 et 3**)
- Quatrième solution : utiliser `watch` (**Vue.js 2 et 3**)
- Cinquième solution : utiliser le hook `updated` (**Vue.js 2 et 3**)

**Première solution : utiliser watchEffect (aucun changement dans le template)**

```
<script>
import { watchEffect } from 'vue'

export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
        watchEffect(() => {
            this.resultat = this.valeur1 + this.valeur2
        })
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

Première solution : utiliser `watchEffect` (aucun changement dans le template)

```
<script>
import { watchEffect } from 'vue'

export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
        watchEffect(() => {
            this.resultat = this.valeur1 + this.valeur2
        })
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

### Remarque

On utilise `watchEffect` pour relancer le calcul après chaque modification constatée.

## Deuxième solution : utiliser computed (aucun changement dans le template) [Syntaxe Vue.js 3]

```
<script>
import { computed } from 'vue'

export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
        this.resultat = computed(() => this.valeur1 + this.valeur2)
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

Deuxième solution : utiliser `computed` (aucun changement dans le template) [Syntaxe Vue.js 3]

```
<script>
import { computed } from 'vue'

export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
        this.resultat = computed(() => this.valeur1 + this.valeur2)
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

## Remarque

On utilise `computed` pour relancer le calcul après chaque modification constatée.

# Vue.js

On peut aussi utiliser computed ainsi : sans déclarer résultat dans data [Syntaxe Vue.js 2]

```
<template>
  <p>{{ valeur1 }} + {{ valeur2 }} = {{ resultat }}</p>
</template>

<script>
export default {
  name: 'ReactiveValue',
  data() {
    return {
      valeur1: 2,
      valeur2: 3,
    }
  },
  computed: {
    resultat() {
      return this.valeur1 + this.valeur2
    }
  },
  mounted() {
    setInterval(() => this.valeur2 += 10, 5000);
  },
}
</script>
```

# Vue.js

	<b>watch</b>	<b>watchEffect</b>	<b>computed</b>
Observe une seule valeur	✓		
A accès à la valeur précédente	✓		
S'exécute immédiatement		✓	✓
S'exécute après changement	✓	✓	✓
Ne prend pas de paramètre		✓	✓

# Vue.js

Le template, doit-il contenir une seule balise enfant ?

- Dans **Vue.js 2**, oui.
- Dans **Vue.js 3**, non.
- Source (Documentation officielle) :

<https://v3-migration.vuejs.org/new/fragments.html#overview>.

# Vue.js

## Quelques bonnes pratiques

- Utiliser le **Camel case** pour émettre un évènement dans un composant enfant.
- Utiliser le **Kebab case**, dans les composants parents, pour intercepter les évènements émis par les composants enfants.
- Limiter les données transmises avec `$emit` à ce qui est strictement nécessaire.
- Utiliser `:key` dans `v-for`
- Pour changer la valeur d'une props, pensez à la cloner