



Composants

Principes de base

Philippe Genoud

Philippe.Genoud@univ-grenoble-alpes.fr

Dernière mise à jour : 05/02/2026

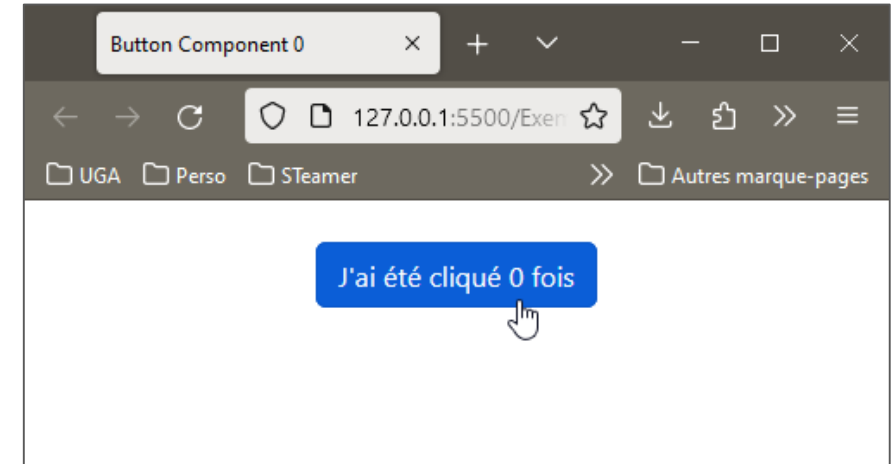


This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

Exemple introductif

- Bouton comptant le nombre de clics

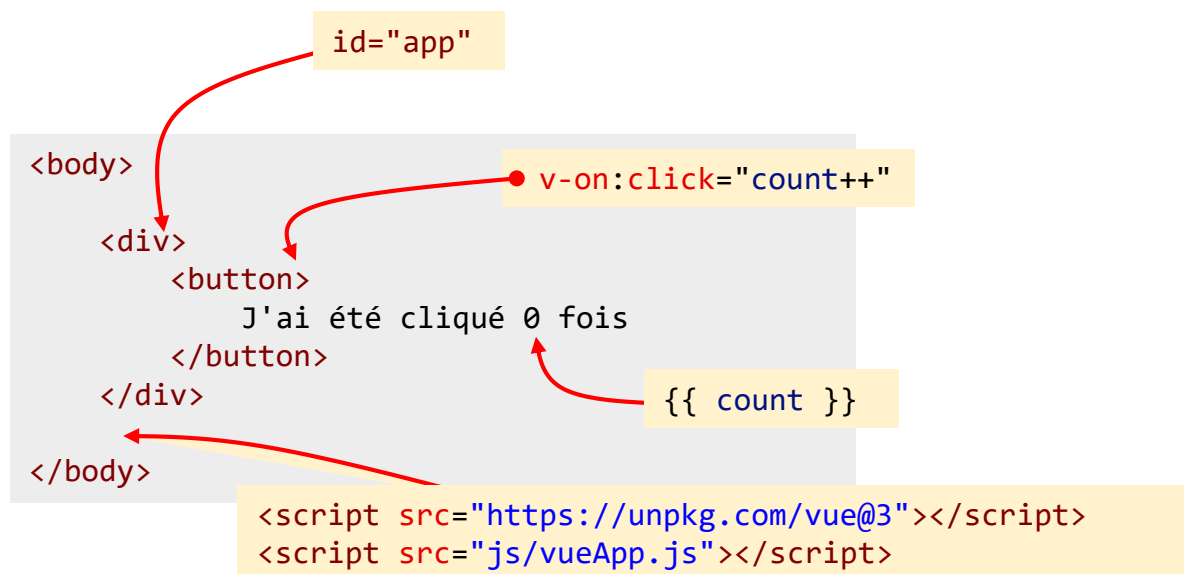
```
<body>
  <div>
    <button>
      J'ai été cliqué 0 fois
    </button>
  </div>
</body>
```



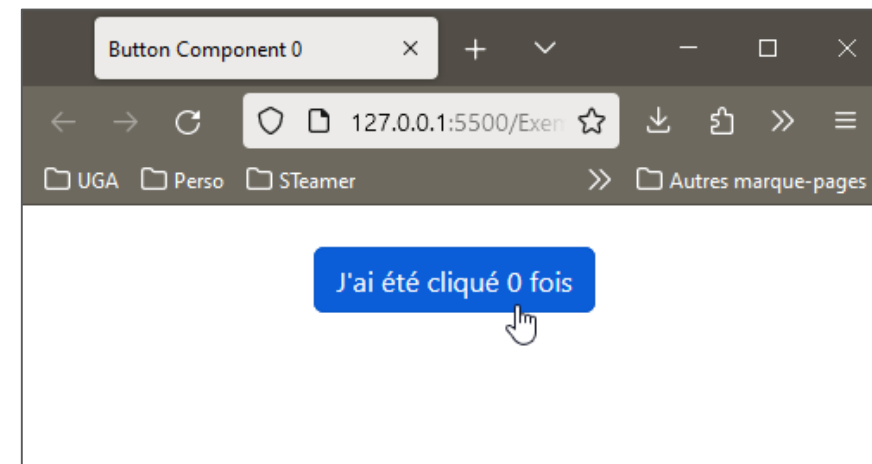
chaque fois que l'utilisateur clique le compteur est augmenté

Exemple introductif

- Bouton comptant le nombre de clics



```
const vueApp = Vue.createApp({
  data() {
    return {
      count: 0,
    }
  }
});
const rootComponent = vueApp.mount('#app');
```



chaque fois que l'utilisateur clique le compteur est augmenté

1. Définir avec Vue un composant dont l'état (`count`) est le nombre de clics sur le bouton
2. Lier cet état au bouton
 - a) le bouton affiche la valeur de `count`
 - b) quand on clique sur le bouton, `count` est incrémenté

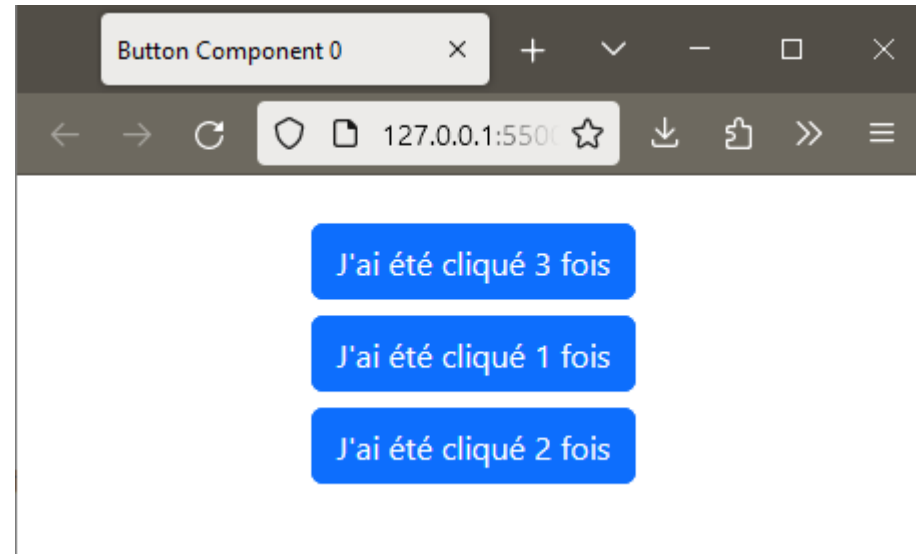
Exemple introductif

- 3 boutons comptant leur nombre de clics respectifs

```
<body>
  <div>
    <button>
      J'ai été cliqué 0 fois
    </button>
  </div>
</body>
```

```
const vueApp = Vue.createApp({
  data() {
    return {
      count: 0,
    }
  }
});

const rootComponent = vueApp.mount('#app');
```



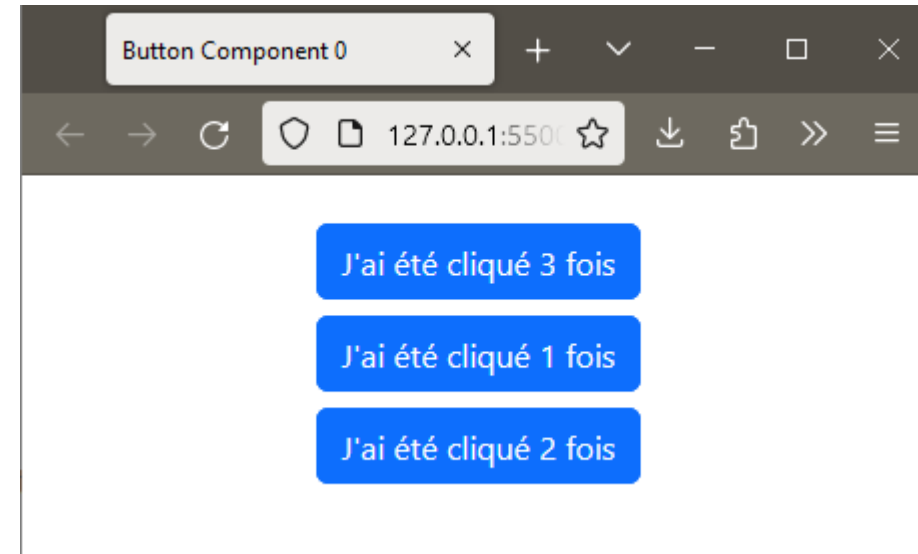
Exemple introductif

- 3 boutons comptant leur nombre de clics respectifs

```
<div id="app">
  <button v-on:click="count1++">
    J'ai été cliqué {{ count1 }} fois
  </button>
  <br>
  <button v-on:click="count2++">
    J'ai été cliqué {{ count2 }} fois
  </button>
  <br>
  <button v-on:click="count3++">
    J'ai été cliqué {{ count3 }} fois
  </button>
</div>
```

```
const vueApp = Vue.createApp({
  data() {
    return {
      count1: 0,
      count2: 0,
      count3: 0
    }
  }
});

const rootComponent = vueApp.mount('#app');
```



Une solution : gérer l'état de chaque bouton dans le composant Vue

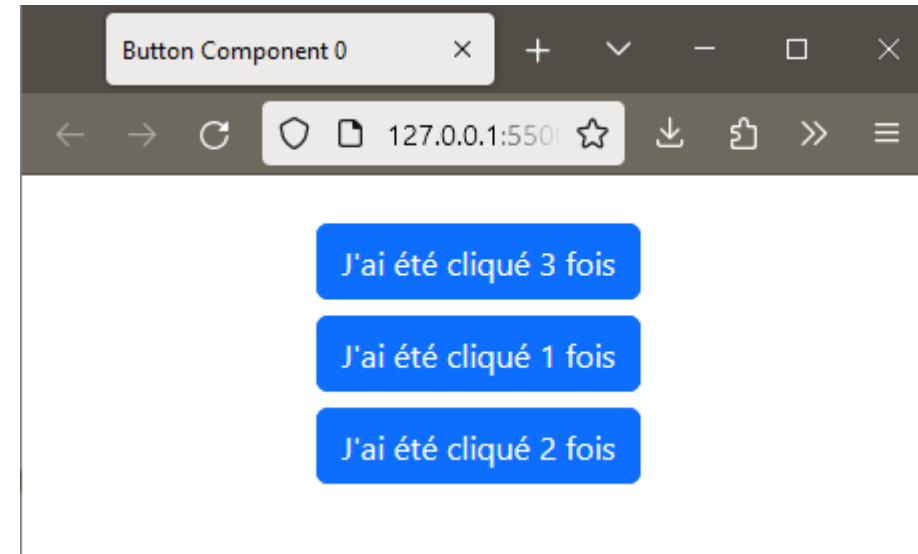
Exemple introductif

- 3 boutons comptant leur nombre de clics respectifs

```
<div id="app">
  <button v-on:click="count1++">
    J'ai été cliqué {{ count1 }} fois
  </button>
  <br>
  <button v-on:click="count2++">
    J'ai été cliqué {{ count2 }} fois
  </button>
  <br>
  <button v-on:click="count3++">
    J'ai été cliqué {{ count3 }} fois
  </button>
</div>
```

```
const vueApp = Vue.createApp({
  data() {
    return {
      count1: 0,
      count2: 0,
      count3: 0
    }
  }
});

const rootComponent = vueApp.mount('#app');
```



Une solution : gérer l'état de chaque bouton dans le composant Vue

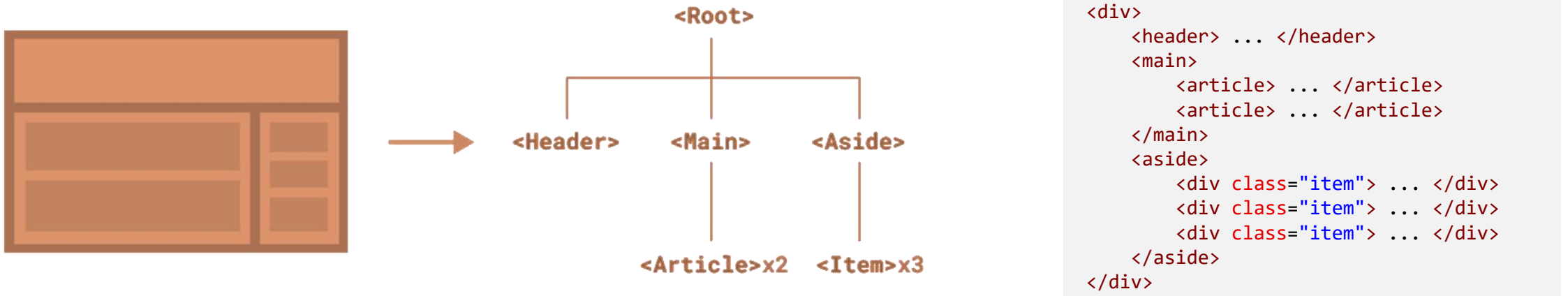


- Duplication de code (dans HTML, dans JavaScript)
- Augmente la complexité
- Perte de lisibilité
- Difficulté à faire évoluer de manière cohérente

Vue3 propose une solution simple et efficace pour ces problèmes : définir le bouton comme un **composant réutilisable**

Composants réutilisables

- définir l'UI d'une l'application de manière hiérarchique en un arbre de composants imbriqués



- Vue propose un modèle de composants qui permet
 - d'étendre le jeu d'éléments natifs (balises) HTML
 - d'encapsuler le contenu et la logique de ces nouveaux éléments au sein d'une même entité (le composant) sur laquelle il est possible de réfléchir et travailler de manière isolée
 - indépendance des composants
 - réutilisation des composants

Définir un composant

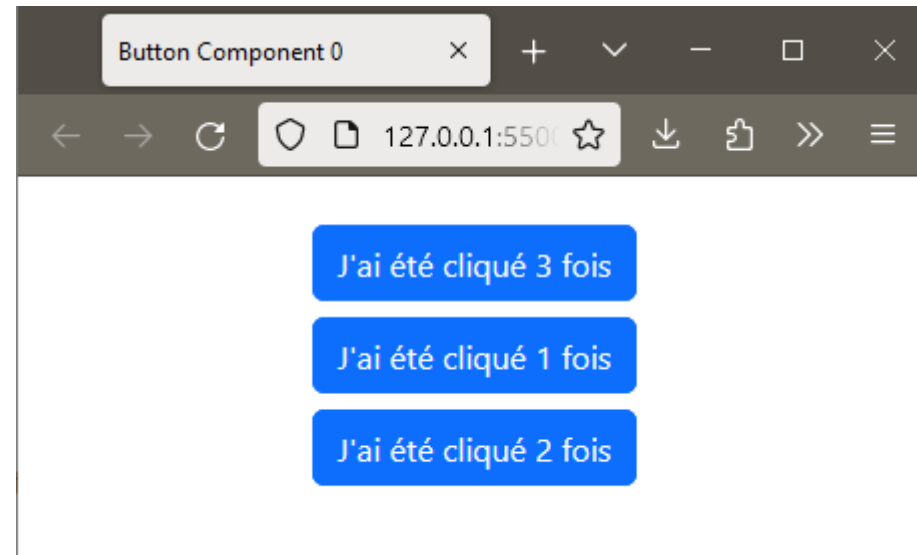
- Dans l'Options API de Vue un composant peut être défini comme un simple objet JavaScript contenant des options spécifiques à Vue

L'objet d'options de l'application définit une propriété **components** dont la valeur est un objet où seront définis les différents objets définissant les composants utilisés

```
const vueApp = Vue.createApp({  
  components: {  
    Composant1 : {  
      ...  
    },  
    Composant2 : {  
      ...  
    }  
  }  
});  
  
const rootComponent = vueApp.mount('#app');
```

Nom du composant
sera utilisé comme nom de balise.
Recommandation : utiliser du PascalCase

Objet d'option Vue (**data**, **methods**, ...) définissant le composant



Définir un composant

- Dans l'Options API de Vue un composant peut être défini comme un simple objet JavaScript contenant des options spécifiques à Vue

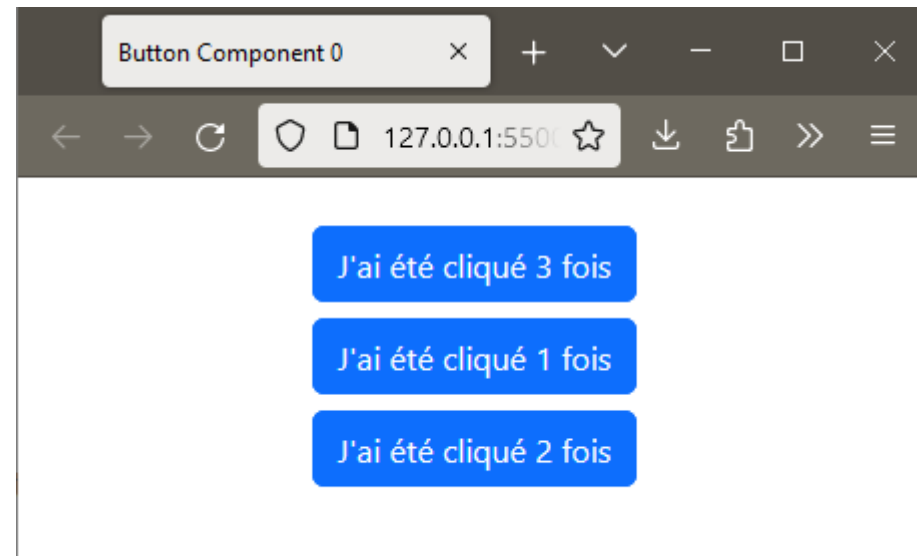
L'objet d'options de l'application définit une propriété **components** dont la valeur est un objet où seront définis les différents objets définissant les composants utilisés

```
const vueApp = Vue.createApp({
  components: {
    CounterButton : {
      data() {
        return {
          count: 0,
        };
      },
      template: `
        <button @click="count++" >
          J'ai été cliqué {{ count }} fois
        </button>`,
    },
  },
});

const rootComponent = vueApp.mount('#app');
```

Données associées au composant

template du composant : une chaîne de caractères définissant le code HTML réactif associé au composant.



Utiliser un composant

- Dans l'Options API de Vue un composant peut être défini comme un simple objet JavaScript contenant des options spécifiques à Vue
- Dans le code HTML le composant peut être utilisé via une simple balise

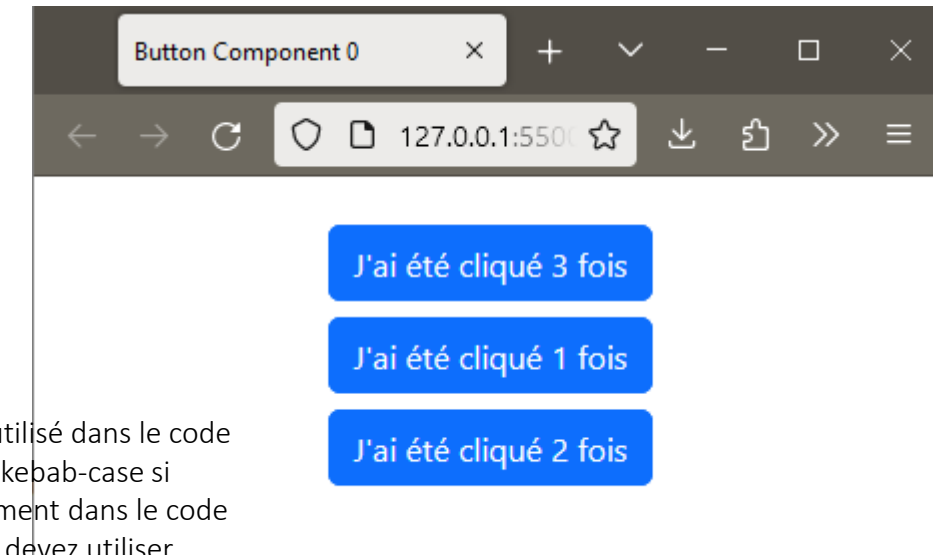
JavaScript

```
const vueApp = Vue.createApp({
  components: {
    CounterButton : {
      data() {
        return {
          count: 0,
        };
      },
      template: `
        <button @click="count++" >
          J'ai été cliqué {{ count }} fois
        </button>`,
    },
  },
});

const rootComponent = vueApp.mount('#app');
```



Attention l'identifiant PascalCase utilisé dans le code JavaScript doit être transformé en kebab-case si vous utilisez le composant directement dans le code HTML de la page*. De même vous devez utiliser systématiquement une balise fermante.



HTML

```
<div id="app">
  <counter-button></counter-button>
  <br>
  <counter-button></counter-button>
  <br>
  <counter-button></counter-button>
</div>
```

* car ce code est transformé en DOM par le parseur du navigateur, il doit donc respecter les conventions de la norme HTML,

Définir un composant

- dans la pratique il vaut mieux définir chaque composant dans un module séparé

vueApp.js

```
const vueApp = Vue.createApp({
  components: {
    CounterButton : {
      data() {
        return {
          count: 0,
        };
      },
      template: `
        <button @click="count++" >
          J'ai été cliqué {{ count }} fois
        </button>`,
    }
  }
});

const rootComponent = vueApp.mount('#app');
```

CounterButton.js

```
export const counterButtonComponent = {
  data() {
    return {
      count: 0,
    };
  },
  template: `
    <button @click="count++" >
      J'ai été cliqué {{ count }} fois
    </button>`,
};
```

vueApp.js

```
import { counterButtonComponent } from "./CounterButton.js";

const vueApp = Vue.createApp({
  components: {
    CounterButton : counterButtonComponent
  }
});

const rootComponent = vueApp.mount('#app');
```



vueApp.js utilisant les modules ES2015 il ne faut pas oublier de spécifier le type **module** dans l'importation du script dans la page

```
<script src="js/vueApp.js" type="module"></script>
```

Définir un composant

- dans la pratique il vaut mieux définir chaque composant dans un module séparé

Le code peut encore être simplifié

Le module CounterButton n'ayant qu'un export on peut utiliser **export default** et on n'a plus besoin de nommer l'entité exportée

Plus besoin des accolades dans l'import

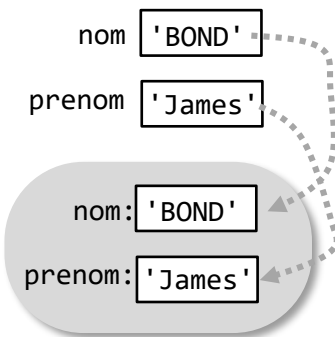
En prenant le même identifiant dans l'import et dans la déclaration **components** on peut utiliser l'écriture simplifiée pour les objets

```
let nom = 'BOND';  
let prenom = 'James';
```

```
const obj = {  
  nom: nom,  
  prenom: prenom  
};
```



```
{  
  nom,  
  prenom  
}
```



Si dans un objet littéral la valeur d'une propriété (attribut) est définie par une variable dont l'identifiant est identique au nom de la propriété, le variable peut être omise

CounterButton.js

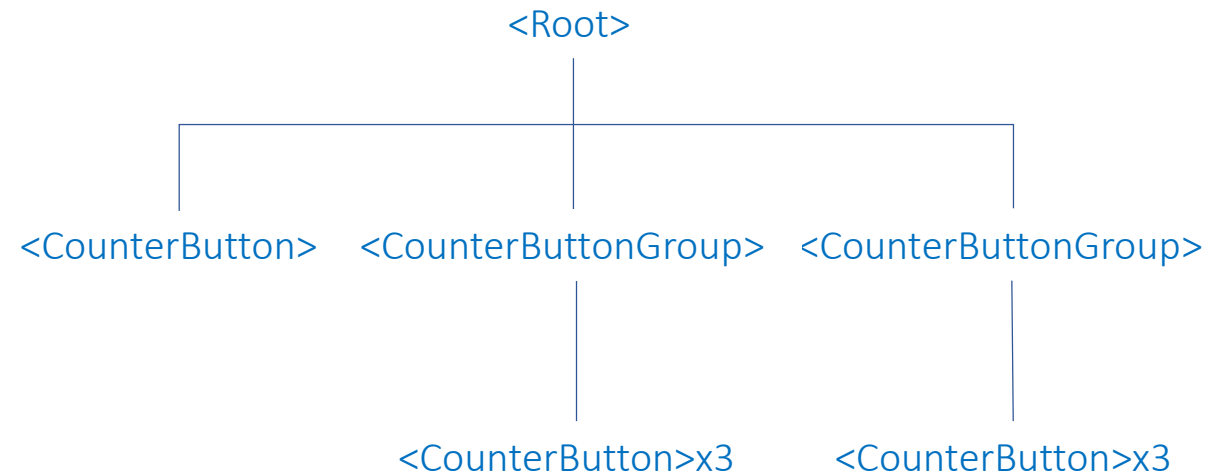
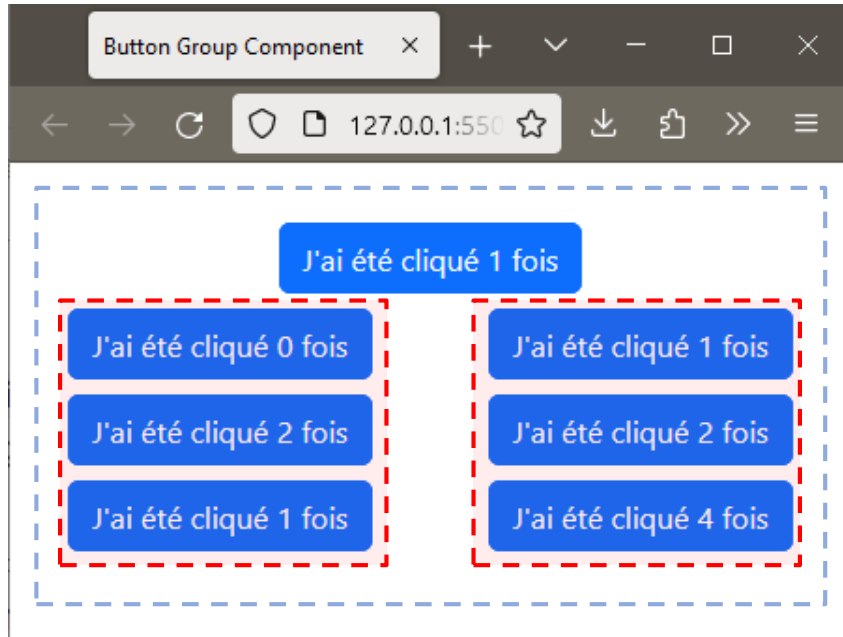
```
export default {  
  data() {  
    return {  
      count: 0,  
    };  
  },  
  template: `  
    <button @click="count++" >  
      J'ai été cliqué {{ count }} fois  
    </button>`,  
};
```

vueApp.js

```
import CounterButton from "./CounterButton.js"  
  
const vueApp = Vue.createApp({  
  components: {  
    CounterButton  
  }  
});  
  
const rootComponent = vueApp.mount('#app');
```

Définir un composant

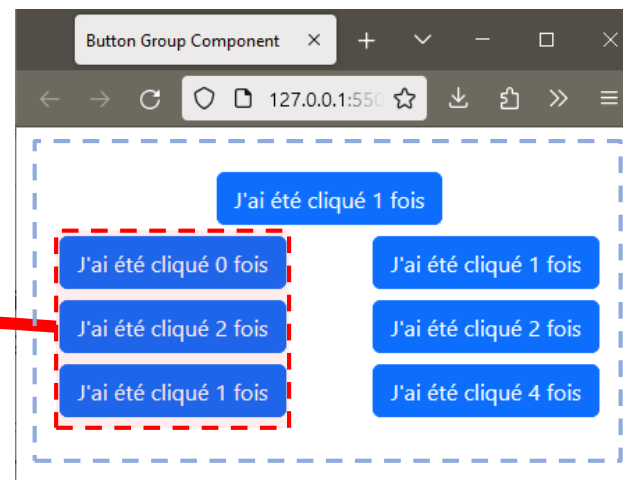
- Un composant peut réutiliser un autre composant



Composant **CounterButtonGroup** regroupant 3 composants **CounterButton**

Définir un composant

- Un composant peut réutiliser un autre composant



CounterButtonGroup.js

```
import CounterButton from "../CounterButton.js";

export default {
  components: {
    CounterButton,
  },
  template: `
    <div class="col">
      <CounterButton />
      <br>
      <CounterButton />
      <br>
      <CounterButton />
    </div>
  `,
};
```

dans chaque composant il faut déclarer les composants qu'il utilise dans son template

dans le **template** on peut utiliser une balise autofermante au nom du composant (PascalCase)

par contre dans le DOM il faut utiliser kebab-case et une balise fermante

vueApp.js

```
import CounterButton from "../CounterButton.js";
import CounterButtonGroup from "../CounterButtonGroup.js";

const vueApp = Vue.createApp({
  components: {
    CounterButton,
    CounterButtonGroup,
  }
});

const rootComponent = vueApp.mount('#app');
```

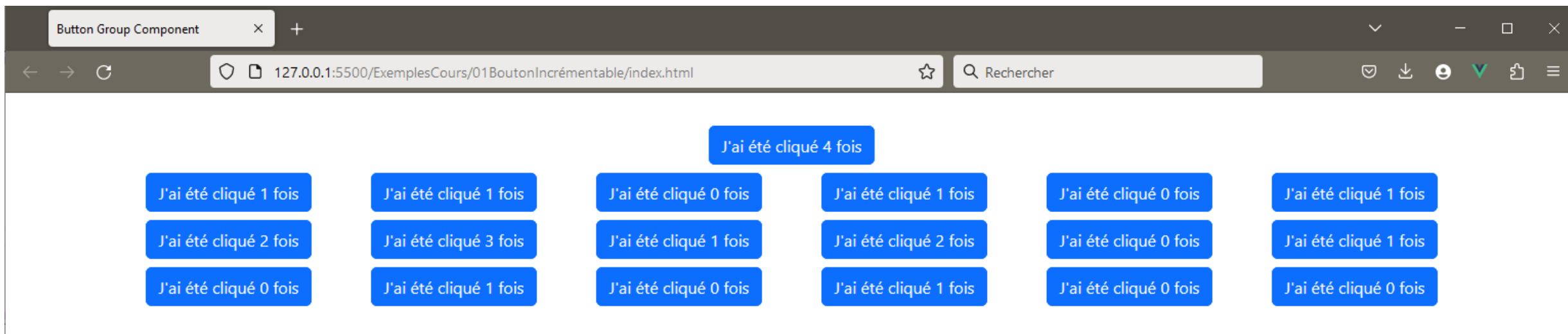
Code JS du composant racine

```
<div id="app" class="container mt-4 text-center">
  <counter-button></counter-button>
  <br>
  <div class="row">
    <counter-button-group></counter-button-group>
    <counter-button-group></counter-button-group>
  </div>
</div>
```

Template du composant racine

Utiliser un composant

- Possibilité d'utiliser les directives Vue sur un composant



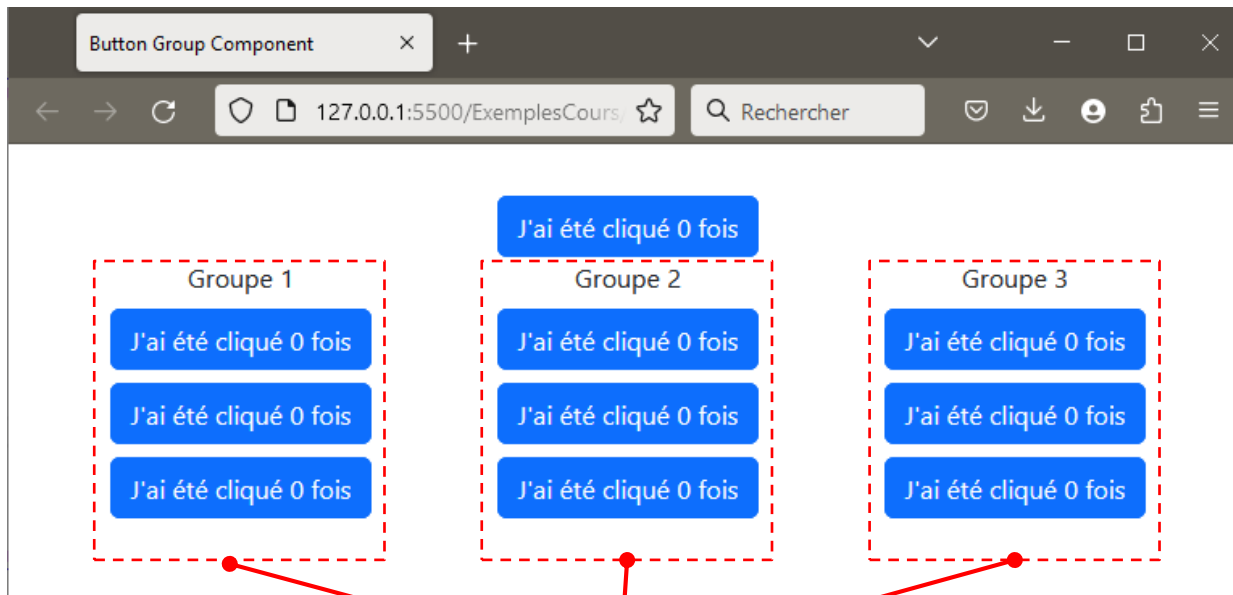
```
<div id="app" class="container mt-4 text-center">
  <counter-button></counter-button>
  <br>
  <div class="row">
    <counter-button-group v-for="n in 6"></counter-button-group>
  </div>
</div>
```



n varie de 1 à 6

Passer des données à un composant : props

- possibilité de transmettre des données au composant à l'aide d'attributs personnalisés (**props**)



```
import CounterButton from "../CounterButton.js";

export default {
  components: {
    CounterButton,
  },
  template: `
    <div class="col">
      <CounterButton />
      <br>
      <CounterButton />
      <br>
      <CounterButton />
    </div>
  `
};
```

Passer des données à un composant : props

- possibilité de transmettre des données au composant à l'aide d'attributs personnalisés (props)



```
<counter-button-group title="Groupe 1"></counter-button-group>
<counter-button-group title="Groupe 2"></counter-button-group>
<counter-button-group title="Groupe 3"></counter-button-group>
```



```
<counter-button-group v-for="n in 3" :title="'Groupe ' + n"></counter-button-group>
```

```
import CounterButton from "../CounterButton.js";
```

```
export default {
  components: {
    CounterButton,
  },
  props: [
    'title'
  ],
  template: `
```

```
<div class="col">
  {{ title }}<br>
  <CounterButton />
  <br>
  <CounterButton />
  <br>
  <CounterButton />
</div>
```

```
</div>
```

```
};
```

l'option **props** permet d'enregistrer la liste des propriétés du composant qui pourront être utilisées comme attributs de la balise pour

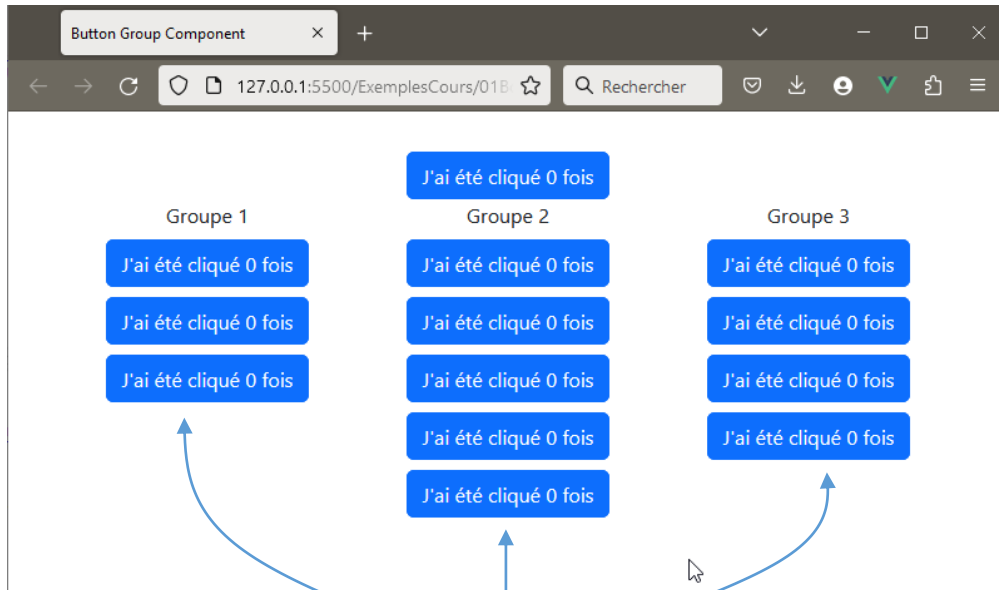
La valeur de cette propriété est accessible à l'intérieur du *template* et dans le contexte **this** du composant, tout comme les données définies dans l'option **data**



directive **v-bind** pour indiquer à Vue que les valeurs passées sont des expressions JavaScript plutôt que des chaînes de caractères.

Passer des données à un composant : props

- possibilité de mettre des **contraintes** sur la propriété (type de données, valeur requise) et de définir une **valeur par défaut**



Paramétrer le nombre de boutons d'un groupe

Pour chaque propriété dans l'objet servant de déclaration, la clé est le nom de la prop.

```
import CounterButton from "../CounterButton.js";

export default {
  components: {
    CounterButton,
  },
  props: {
    title: String,
    nbButtons: {type: Number, default: 3}
  },
  template: `
    <div class="col">
      {{ title }}<br>
      <template v-for="i in nbButtons">
        <CounterButton />
        <br>
      </template>
    </div>
  `
};
```

au lieu de définir les **props** avec un tableau de noms on utilise un objet

fonction constructeur du type attendu

valeur par défaut

Balise **<template>** pour pouvoir utiliser la directive **v-for** sans restituer un élément dans le DOM¹.

¹ voir <https://vuejs.org/api/built-in-special-elements#template>

Passer des données à un composant : props

- utilisation de la propriété `nbButtons`
 - camelCase dans le JavaScript
 - kebab-case dans le HTML

`nb-buttons`

utilisation de la valeur par défaut

```
import CounterButton from "../CounterButton.js";

export default {
  ...
  props: {
    title: String,
    nbButtons: {type: Number, default: 3}
  },
  ...
};
```

```
<counter-button-group title="Groupe 1"></counter-button-group>
<counter-button-group title="Groupe 2" :nb-buttons="5" ></counter-button-group>
<counter-button-group title="Groupe 3" :nb-buttons="4" ></counter-button-group>
```

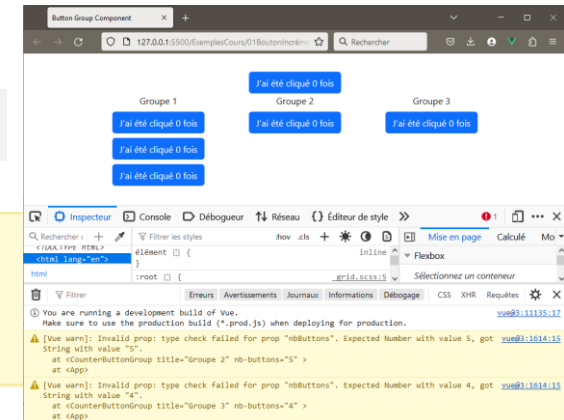


utiliser directive **v-bind** pour indiquer à Vue que les valeurs passées sont des expressions JavaScript plutôt que des chaînes de caractères.

```
<counter-button-group title="Groupe 2" nb-buttons="5"></counter-button-group>
```

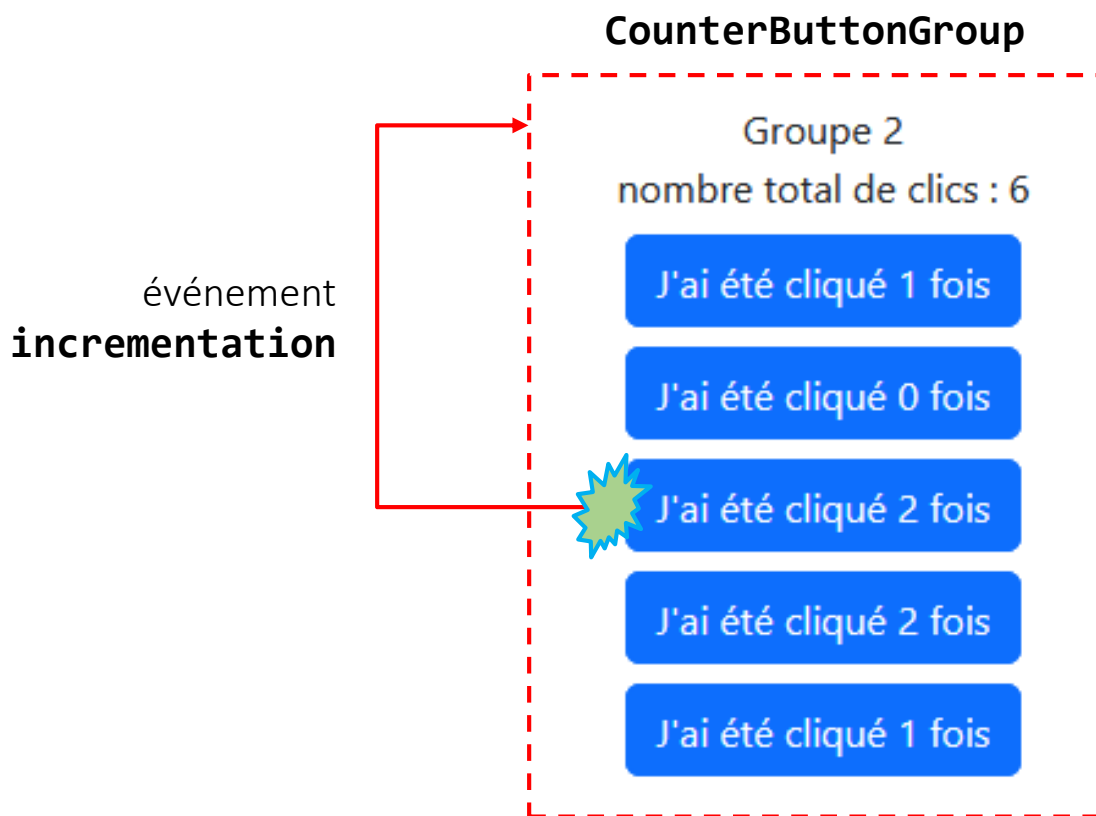
⚠ [Vue warn]: Invalid prop: type check failed for prop "nbButtons". Expected Number with value 5, got String with value "5".

```
at <CounterButtonGroup title="Groupe 2" nb-buttons="5" >
at <App>
```



Événements de composants

- certaines fonctionnalités peuvent nécessiter de communiquer avec le parent
- Vue.js fournit un système d'événements personnalisés pour les composants.
 - Les instances de composants peuvent émettre des événements que le composant parent peut choisir d'écouter comme pour un événement natif du DOM grâce à une directive **v-on** ou **@**,



Le composant **CounterButtonGroup** conserve dans son état le nombre total de clics effectué sur les boutons qu'il contient

A chaque nouveau clic les composants **CounterButton** préviennent le composant parent que leur état a changé

Événements de composants

- A chaque clic un composant **CounterButton** émet un événement **incrémentation**

CounterButton.js

```
export default {  
  data() {  
    return {  
      count: 0,  
    };  
  },  
  emits: [ 'incrémentation' ],  
  methods: {  
    increment() {  
      this.count++;  
      this.$emit('incrémentation');  
    }  
  },  
  template: `  
    <button @click="increment">  
      J'ai été cliqué {{ count }} fois  
    </button>`,  
};
```

Comme pour les *props*, les événements émis par un composant doivent¹ être déclarés dans une option **emits**

¹ Ce n'est pas une obligation, mais est considéré comme une bonne pratique de documentation du composant

Le composant émet l'événement en appelant la méthode intégrée **\$emit**, et en lui passant en argument le nom de l'événement

Un click sur le bouton appelle la méthode **increment()** qui déclenchera l'événement **incrémentation**

Événements de composants

- le composant parent `CounterButtonGroup` écoute les événements `incrementation` des ses composants enfants `CounterButton`

CounterButtonGroup.js

```
import CounterButton from "../CounterButton.js";

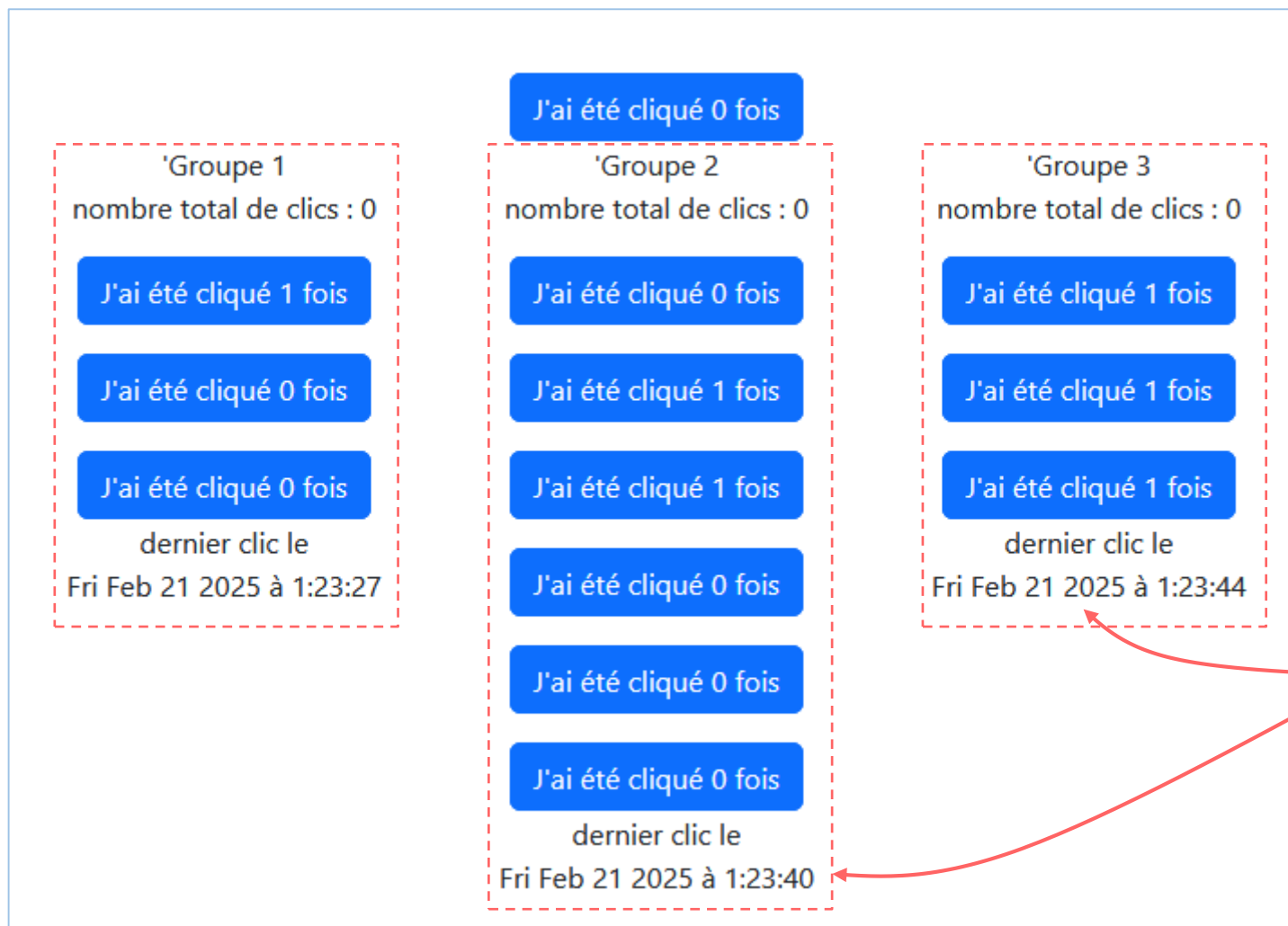
export default {
  components: {
    CounterButton,
  },
  data() {
    return {
      nbClicks : 0 // nombre total de clics
    }
  },
  props : {
    title: String,
    nbButtons: {type: Number, default: 3}
  },
  template: `
    <div class="col">
      {{ title }}<br>
      nombre total de clics : {{ nbClicks }}
      <template v-for="i in nbButtons">
        <CounterButton @incrementation="nbClicks++"/>
        <br>
      </template>
    </div>`,
};
```

ajout d'une propriété `nbClicks` permettant de conserver le nombre de clics total

écoute des événements `incrementation` à l'aide d'une directive `v-on` à chaque événement `nbClicks` est incrémenté

Événements de composants

- Possibilité de transmettre des valeurs avec l'événement
 - exemple le composant `CounterButton` indique l'heure de l'événement `incrementation`



Le groupe de boutons affiche la date et l'heure du dernier click

Événements de composants

- Possibilité de transmettre des valeurs avec l'événement
 - exemple le composant `CounterButton` indique l'heure de l'événement `incrementation`

CounterButton.js

```
export default {
  data() {
    return {
      count: 0,
    };
  },
  emits: [ 'incrementation' ],
  methods: {
    increment() {
      this.count++;
      this.$emit('incrementation', new Date());
    }
  },
  template: `
    <button @click="increment">
      J'ai été cliqué {{ count }} fois
    </button>`,
};
```

Un objet `Date` indiquant la date et l'heure courantes est transmis au composant parent à l'écoute de l'événement `incrementation` en le passant comme argument de la méthode `$emit`

Événements de composants

- Possibilité de transmettre des valeurs avec l'événement
 - au niveau du composant parent la fonction d'écoute récupère en paramètre la valeur transmise comme argument lors de l'appel de `$emit` par le composant enfant

CounterButtonGroup.js

```
import CounterButton from "../CounterButton.js";
```

```
export default {  
  components: {  
    CounterButton,  
  },  
  data() {
```

```
    return {  
      nbclicks: 0, // nombre total de clics,  
      lastClick: null,  
    };  
  },  
  props: {
```

```
    title: String,  
    nbButtons: { type: Number, default: 3 },  
  },  
  methods: {
```

```
    newClick(date) {  
      this.nbclicks++;  
      this.lastClick = date;  
    }  
  },  
};
```

modification
de l'attribut
lastClick

Mise à jour du template HTML

template: `

```
<div class="col">
```

```
  {{ title }}<br>
```

```
  nombre total de clics : {{ nbclicks }}<br>
```

```
  <template v-for="i in nbButtons">
```

```
    <CounterButton @incrementation="newClick"/>
```

```
  </template>
```

```
  <template v-if="lastClick">dernier clic le<br>
```

```
    {{ lastClick.toDateString() }} à  
    {{ lastClick.getHours() }}:{{ lastClick.getMinutes() }}  
    :{{ lastClick.getSeconds() }}
```

```
  </template>
```

```
</div>`,
```

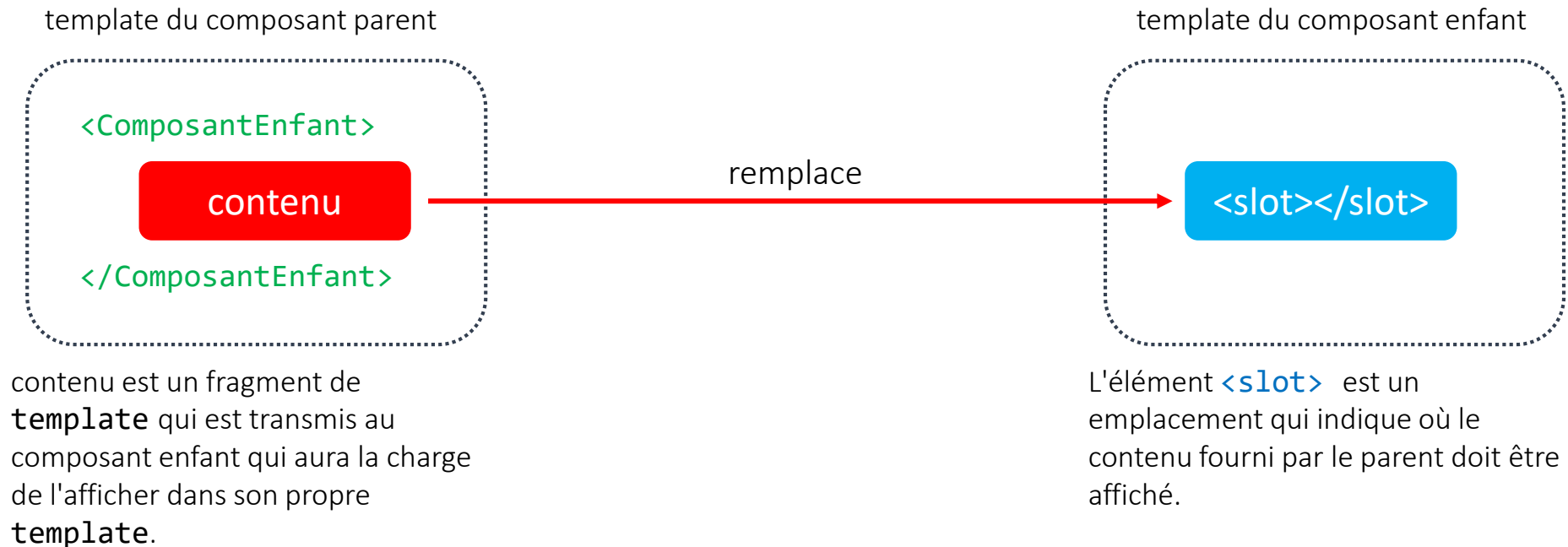
```
};
```

sur un événement `incrementation` appel de la fonction de gestion d'événement avec en paramètre la date du clic

Tous les arguments supplémentaires passés à `$emit()` après le nom de l'événement seront transmis à l'écouteur. Par exemple, avec `$emit('foo', 1, 2, 3)` la fonction d'écoute recevra trois arguments.

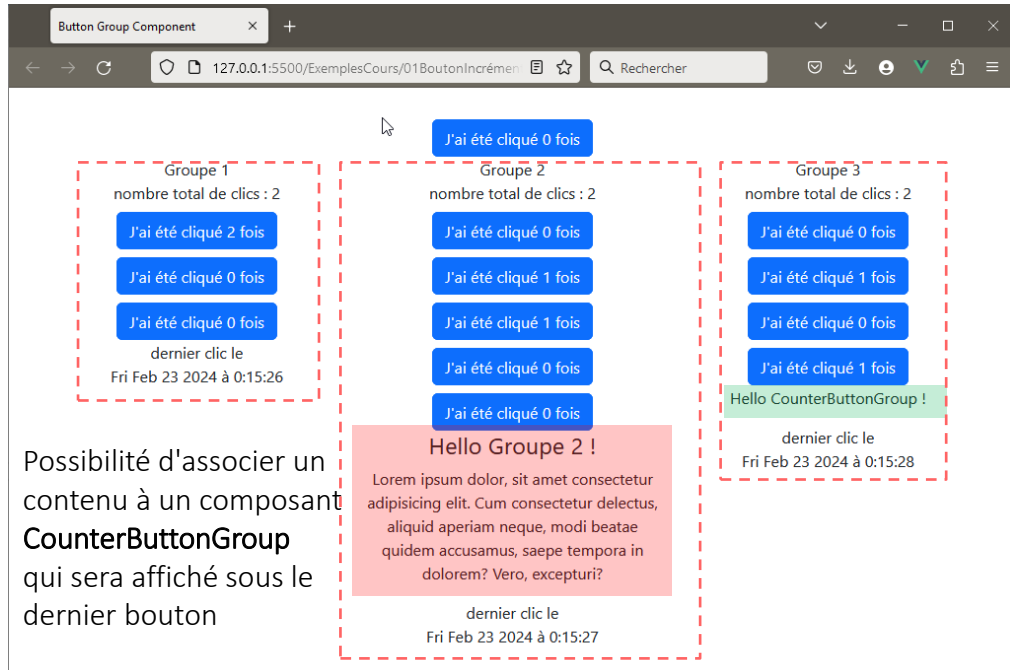
Distribution de contenu : slots

- il peut être utile de pouvoir passer du contenu à un composant, à la manière de ce qui est fait pour les éléments HTML : utilisation d'un élément `<slot>`



Le contenu du slot a accès aux données du composant parent car il est défini dans celui, mais n'a pas accès aux données du composant enfant (similaire à la portée lexicale de JavaScript)

Distribution de contenu : slots



Dans le template de **CounterButtonGroup**

```
template: `
<div class="col">
  {{ title }}<br>
  nombre total de clics : {{ nbclicks }}<br>
  <template v-for="i in nbButtons">
    <CounterButton @incrementation="newClick"/><br>
  </template>
  <slot></slot>
  <template v-if="lastClick">dernier clic le<br>
    {{ lastClick.toDateString() }} à
    {{ lastClick.getHours() }}:
    {{ lastClick.getMinutes() }}:
    {{ lastClick.getSeconds() }}
  </template>
</div>`,
```

Dans le template de la page **index.html**

```
<div id="app" class="container mt-4 text-center">
  <counter-button></counter-button>
  <br>
  <div class="row">
    <counter-button-group title="Groupe 1"></counter-button-group>
    <counter-button-group title="Groupe 2" :nb-buttons="5">
      <h4>Hello Groupe 2 !</h4>
      <p>
        Lorem ipsum dolor, sit amet consectetur adipisicing
        elit.Cum consectetur delectus, aliquid aperiam neque,
        modi beatae quidem accusamus, saepe tempora in dolore?
        Vero, excepturi?
      </p>
    </counter-button-group>
    <counter-button-group title="Groupe 3" :nb-buttons="4">
      <p> {{ message }}</p>
    </counter-button-group>
  </div>
</div>
```

Contenu HTML
statique

Contenu réactif lié à la propriété
message des données du composant
root (si **message** change, le message
affiché sur la page change)

Composant et Gestion d'état

- chaque instance de composant Vue constitue une unité autonome qui "gère" son propre état réactif.

```
export default {  
  data() {  
    return {  
      count: 0,  
    };  
  },  
  emits: [ 'incrementation' ],  
  methods: {  
    increment() {  
      this.count++;  
      this.$emit('incrementation', new Date());  
    }  
  },  
  template: `  
    <button @click="increment" >  
      J'ai été cliqué {{ count }} fois  
    </button>`,  
};
```

Etat

Actions

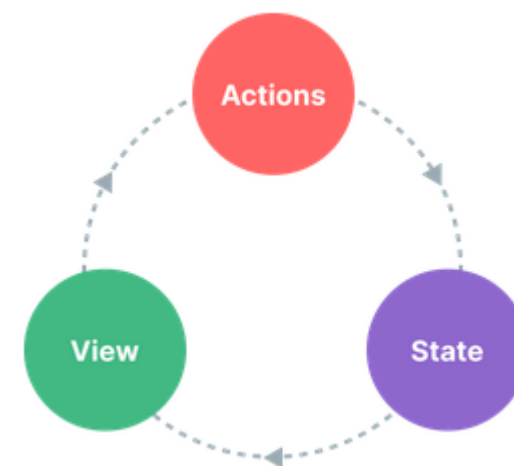
Vue

Composé de

source des données qui dirigent l'application

les différentes manières dont l'état pourrait changer, en réaction aux entrées de l'utilisateur dans la vue

cartographie déclarative de l'état

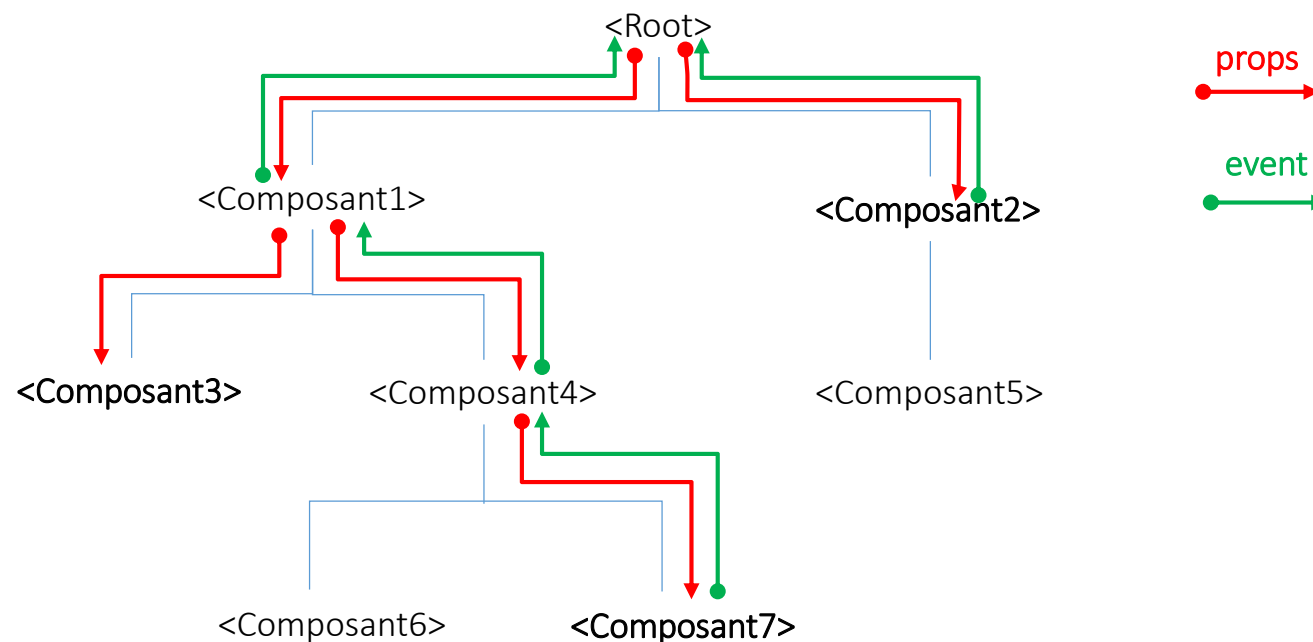


Modèle simple avec un flux de données à sens unique

Composant et Gestion d'état

- Le modèle simple se complexifie quand plusieurs composants doivent partager un état commun
 - Différentes vues dépendent de la même partie d'un état
 - Les actions de différentes vues ont besoin de muter la même partie d'un état.

- définir l'état partagé au niveau d'un composant ancêtre commun, puis le transmettre en tant que *props* aux composants enfants
- faire remonter un événement jusqu'au composant possédant l'état partagé



Solutions fragiles et pouvant conduire rapidement à un code non maintenable.
Créent des dépendances entre composants qui sont difficilement réutilisables dans un contexte différent



Pour des applications complexes utilisation d'une bibliothèque de gestion d'états (store) → Pinia <https://pinia.vuejs.org/>



Cycle de Vie des Composants Vue.js

- Cycle de vie d'un composant :
 - décrit les étapes que le composant traverse depuis sa création à sa destruction
 - Permet au développeur de contrôler le comportement des composants à chaque étape de leur existence
 - Le développeur peut se greffer à un moment précis de l'existence du composant pour y déclencher des actions
 - 3 phases principales
 - Création
 - Mise à jour
 - Destruction
 - Lors de l'exécution de l'application Vue.js invoque des fonctions particulières appelées « crochets du cycle de vie » (*Lifecycle hooks*) dans lesquels le développeur peut mettre son propre code.

Principaux hooks

- Phase de création

Hook	Moment	Rôle
<code>beforeCreate()</code>	Avant l'instanciation du composant	Initialisation des injections de dépendances externes. Initialisation des options de l'instance Vue.js.
<code>created()</code>	Après l'instanciation du composant	Initialisation des données réactives, propriétés calculées, méthodes et observateurs.
<code>beforeMount()</code>	Avant le montage du composant dans le DOM	Configuration des options de rendu et compilation du template.
<code>mounted()</code>	Après le montage du composant dans le DOM	Insertion de l'instance Vue.js dans le DOM. Accès aux éléments du DOM rendu du composant.

Source : <https://laconsole.dev/formations/vue/cycle-de-vie>

Principaux hooks

- Phase de mise à jour

Hook	Moment	Rôle
<code>beforeUpdate()</code>	Avant que le composant mette à jour son arborescence DOM en raison d'un changement d'état réactif.	Accès à l'état du DOM rendu du composant avant sa mise à jour.
<code>updated()</code>	Après que le composant ait mis à jour son arborescence DOM en raison d'un changement d'état réactif.	Accès à l'état du DOM rendu du composant après sa mise à jour.

Source : <https://laconsole.dev/formations/vue/cycle-de-vie>

- Phase de suppression

Hook	Moment	Rôle
<code>beforeUnmount()</code>	Avant le démontage du composant du DOM	Accès à l'état du DOM rendu du composant avant sa suppression.
<code>unmounted()</code>	Après le démontage du composant du DOM	Plus d'accès à l'état du DOM rendu du composant.

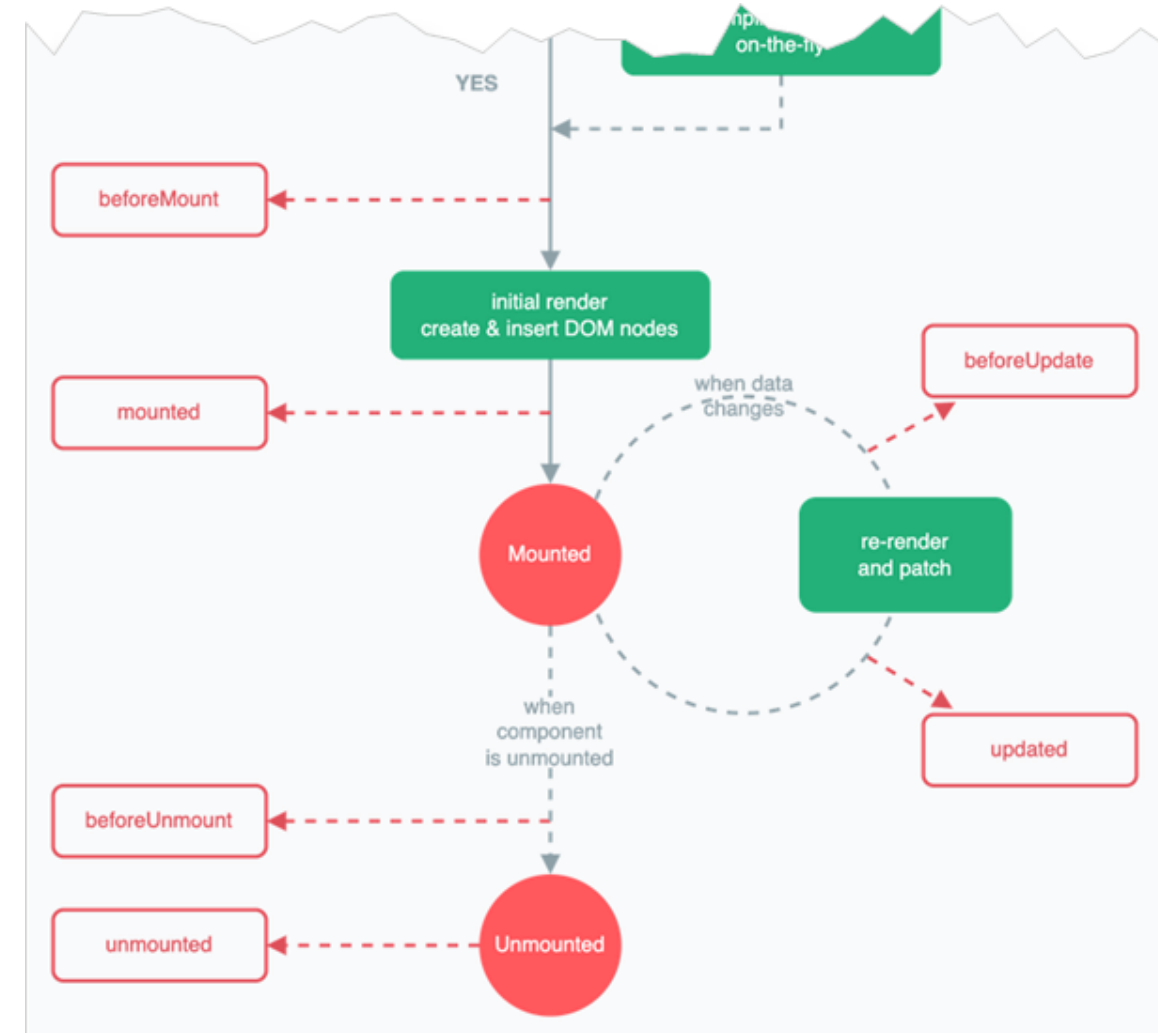
Source : <https://laconsole.dev/formations/vue/cycle-de-vie>

Déclaration des hooks

- Méthodes à déclarer dans l'objet d'options de l'instance de Vues.js (composant racine) ou de vos composants

```
export default {  
  data() {  
    return { ... }  
  },  
  created() {  
    // Le composant est créé, je peux récupérer des données  
    // en provenance d'une API avec fetch()  
  },  
  mounted() {  
    // Le composant a été monté sur le DOM,  
    // c'est par exemple le moment de manipuler la structure HTML  
  },  
  beforeUnmount() {  
    // Le composant a été supprimé du DOM. Il pourrait être intéressant  
    // de supprimer des timers et écouteurs d'événements  
    // (à des éléments DOM non gérés par Vue.js)  
    // afin d'éviter de les cumuler lors de la navigation sur le site.  
  },  
  methods: {  
    ...  
  },  
  ...  
}
```

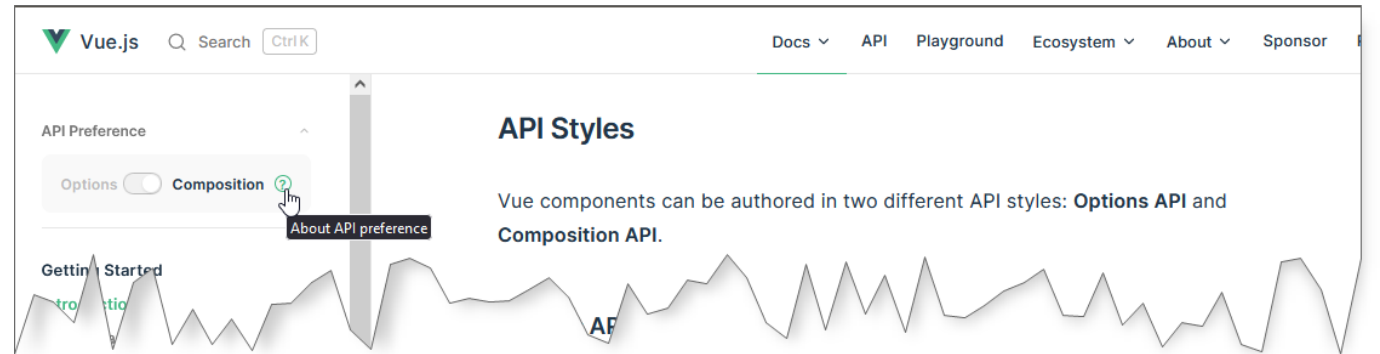
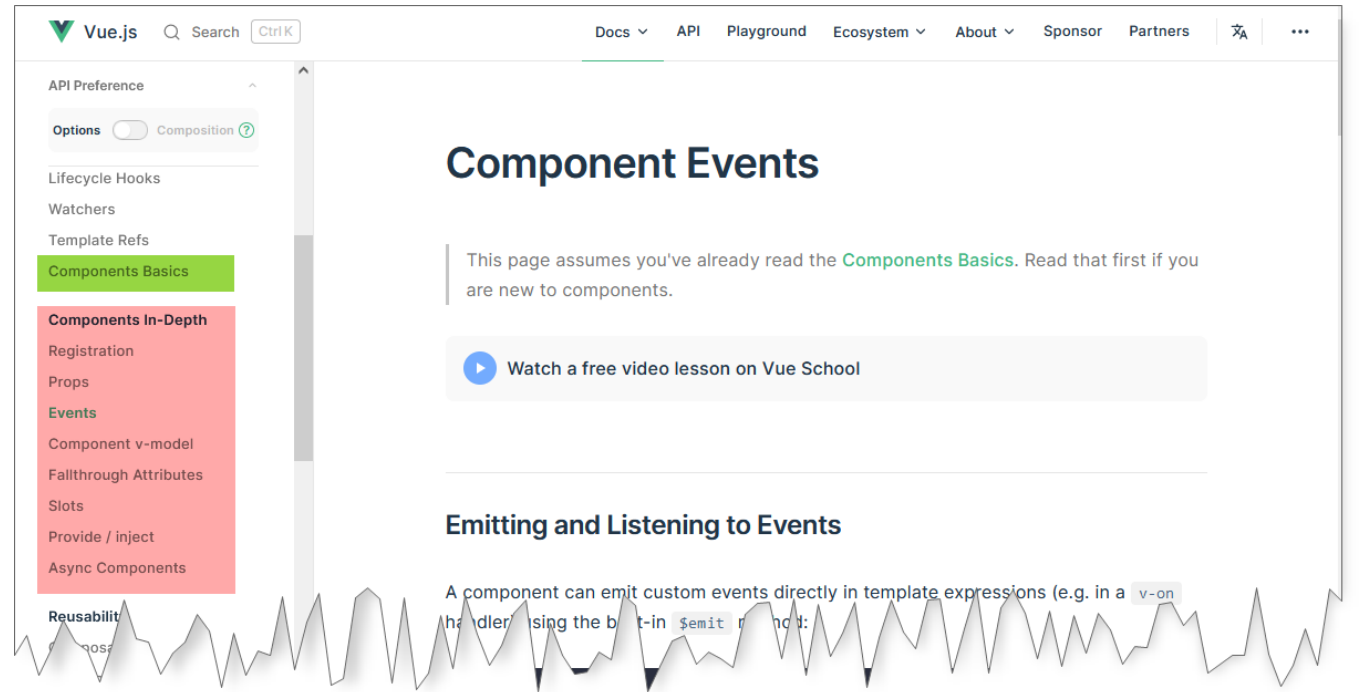
Diagramme du cycle de vie



Pour aller plus loin

- <https://vuejs.org/guide/>
- <https://fr.vuejs.org/guide/>

- l'API de composition



Pour aller plus loin

- Frameworks de composants Vue
 - Vuetify
 - <https://vuetifyjs.com/en/>
 - PrimeVue
 - <https://primevue.org/>
 - Quasar
 - <https://quasar.dev/vue-components>
 - NaiveUI
 - <https://www.naiveui.com/en-US/dark>
 - ...

<https://dev.to/ranaharoon3222/the-ultimate-list-of-7-perfect-vue-3-ui-libraries-for-every-project-1l39>

