

Les fonctionnalités avancées des composants

Ibrahim ALAME

14/02/2023

1 Introduction au chapitre

1.1 Objectifs du chapitre

Nous allons voir comment utiliser **v-model** sur les composants. Nous allons étudier les cascades d'attributs, les **slots**, l'injection avec **Provide** et **Inject**.

Nous verrons également des fonctions utilitaires permettant de transformer des objets réactifs en références. Enfin, nous verrons les composants asynchrones.

1.2 Code de la vidéo

Pour rappel, la directive **v-model** équivaut à l'utilisation de **v-bind** sur l'attribut `value` et de l'utilisation de **v-on** sur l'événement `input`. Voici l'équivalent de **v-model** :

2 Utilisation de la directive v-model sur des composants

2.1 Fonctionnement de v-model avec un composant

Lorsque vous utilisez **v-model** sur un composant enfant :

```
<Enfant v-model="uneProp" />
```

Cela revient en fait exactement à :

```
<Enfant
  :modelValue="uneProp"
  @update:modelValue="val => uneProp = val"
/>
```

Cela signifie que dans le composant enfant il faut émettre un événement **update:modelValue** et déclarer une **prop modelValue** :

```
<script setup>
defineProps({
  modelValue: string;
```

```

}>();
defineEmits<{
  (e: 'update:modelValue', value: string);
}>();
</script>

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

```

2.2 Changer le nom de la **prop** et de l'événement

Comme nous venons de le voir, par défaut, la **prop** reçue par le composant enfant est **modelValue** et l'événement qu'il doit ressentir est **update:modelValue**. Il est possible de modifier le nom en passant un argument à **v-model** :

```
<Enfant v-model:unNom="uneProp" />
```

Vous pouvez ensuite utiliser ce nom dans le composant enfant :

```

<script setup>
defineProps<{
  unNom: string;
}>();
defineEmits<{
  (e: 'update:unNom', value: string);
}>();
</script>

<template>
  <input
    type="text"
    :value="unNom"
    @input="$emit('update:unNom', $event.target.value)"
  />
</template>

```

2.3 Utiliser plusieurs directives **v-model**

Vous pouvez sans problème utiliser plusieurs directives v-model sur un composant :

```
<Enfant v-model:une-prop="uneProp" v-model:une-autre-prop="uneAutreProp" />
```

Par exemple :

```
<Modal v-model:prenom="prenom" v-model:nom="nom" />
```

Et nous aurions dans le composant enfant :

```
<template>
  <input
    type="text"
    :value="prenom"
    @input="$emit('update:prenom', $event.target.value)"
  />
  <input
    type="text"
    :value="nom"
    @input="$emit('update:nom', $event.target.value)"
  />
</template>

<script setup lang="ts">
defineProps<{
  prenom: string;
  nom: string;
}>();

defineEmits<{
  (e: 'update:prenom', value: string);
  (e: 'update:nom', value: string);
}>();
</script>

<style scoped lang="scss"></style>
```

Voici l'exemple exécutable :

2.4 Utiliser des modificateurs

Vous pouvez utiliser des modificateurs personnalisés ou définis par [Vue.js](#), comme par exemple `trim`, `number` etc. Par exemple, pour supprimer les espaces possibles avant ou après :

```
<Modal v-model:prenom.trim="prenom" v-model:nom.trim="nom" />
```

Pour utiliser un modificateur personnalisé, il faut utiliser côté parent :

```
<Modal v-model:prenom="prenom" v-model:nom.exemple="nom" />
```

Et côté enfant :

```
<template>
  <input
    type="text"
    :value="prenom"
    @input="$emit('update:prenom', $event.target.value)"
  />
  <input
    type="text"
    :value="nom"
    @input="emitName"
  />
</template>

<script setup lang="ts">
defineProps<{
  prenom: string;
  nom: string;
  nameModifiers?: { [s: string]: boolean };
}>();

defineEmits<{
  (e: 'update:prenom', value: string);
  (e: 'update:nom', value: string);
}>();

function emitName(event) {
  let value = (event.target as HTMLInputElement).value;
  if (props.modelModifiers.exemple) {
```

```

    // Modifier la valeur comme on souhaite si le modificateur est présent :
    value = value.toUpperCase();
  }
  emit('update:modelValue', value);
}
</script>

<style scoped lang="scss"></style>

```

2.5 Code de la vidéo

Pour rappel, la directive `v-model` équivaut à l'utilisation de `v-bind` sur l'attribut `value` et de l'utilisation de `v-on` sur l'événement `input`. Voici l'équivalent de `v-model` :

3 Cascade d'attributs

3.1 Les attributs en cascade

Les attributs en cascade sont rarement utilisés, vous pouvez passer rapidement pour une première approche de [Vue.js](#) et y revenir plus tard.

Un attribut en cascade est un attribut ou un écouteur d'événement déclaré avec `v-on` qui est passé à un composant sans être déclaré précisé, c'est-à-dire sans utiliser de `props`. Par exemple, si nous avons le composant enfant :

```

<template>
  <h1>Hello</h1>
</template>

```

Et que nous déclarons un attribut sur celui-ci dans le composant parent :

```

<template>
  <Enfant class="large" />
</template>

```

Le composant enfant rendu sur le DOM sera :

```

<h1 class="large">Hello</h1>

```

3.2 Fusionnement d'attributs

Comme vu précédemment, si le composant enfant a déjà un attribut déclaré, les valeurs seront fusionnées. En reprenant l'exemple avec l'attribut `class` :

```
<template>
  <h1 class="titre">Hello</h1>
</template>
```

Et dans le composant parent :

```
<template>
  <Enfant class="large" />
</template>
```

Nous aurons :

```
<h1 class="titre large">Hello</h1>
```

3.3 Ecouteurs déclarés avec **v-on**

C'est exactement le même principe avec les écouteurs d'événements déclarés en utilisant **v-on**. Si le **template** du composant parent est :

```
<template>
  <Enfant @click="onClick" />
</template>
<script>
function onClick() {
  console.log("clac");
}
</script>
```

L'écouteur sera placé automatiquement sur l'élément racine du composant enfant. Un clic sur le composant enfant déclenchera donc le gestionnaire déclaré dans le composant parent. A noter que si l'élément racine du composant enfant a déjà un ou plusieurs gestionnaires d'événements déclarés, ils seront également exécutés. Cela ne les remplacera pas.

3.4 Utilisation de **\$attrs** côté **template**

Vue met à disposition automatiquement une propriété **\$attrs** que vous pouvez utiliser dans le **template** du composant enfant. Elle contient tous les attributs et les écouteurs d'événements passés en cascade. C'est par exemple utile si le composant enfant n'a pas d'élément racine, pour décider de placer les attributs reçus sur un élément particulier :

```
<template>
  <h1>Hello</h1>
  <h2 v-bind="$attrs">Hello 2</h2>
</template>
```

3.5 Utilisation de `useAttrs()` côté `script`

Il est également possible d'accéder aux attributs en utilisant `useAttrs()` côté `script` :

```
<script setup>

import { useAttrs } from 'vue';

const attrs = useAttrs();
</script>
```

3.6 Désactivation du passage automatique des attributs

Il est possible de désactiver le passage automatique des attributs dans le cas où par exemple, vous avez bien un élément racine sur le composant enfant mais que vous souhaitez appliquer les attributs sur d'autres éléments :

```
<template>
  <div class="uneClasse">
    <h1>Hello</h1>
    <h2 v-bind="$attrs">Hello 2</h2>
  </div>
</template>

<script lang="ts">
export default {
  inheritAttrs: false,
};
</script>

<script setup lang="ts"></script>
```

Notez bien l'utilisation de deux balises `scripts` ! Elles seront fusionnées automatiquement avec la configuration par `Vite`.

3.7 Code de la vidéo

Voici le code de la vidéo :

4 Présentation des slots

4.1 Les `slots`

Les `slots` permettent de passer des parties de `template` d'un composant parent à un composant enfant. Nous avons vu les `props` qui permettent de passer des valeurs JavaScript d'un composant parent à un composant enfant.

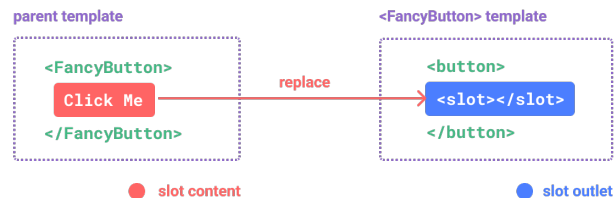
Les **slots** permettent de passer du HTML. Pour utiliser les **slots**, il faut utiliser le composant enfant avec deux balises et mettre le fragment HTML à passer, c'est-à-dire le **slot**, entre ces deux balises :

```
<Enfant>
  J'insère du contenu ici pour le passer au composant enfant.
</Enfant>
```

Dans le composant enfant, il faut utiliser une balise `<slot>` pour indiquer où afficher le contenu passé :

```
<template>
  // Le contenu sera projeté à la place de slot :
  <slot></slot>
</template>
```

Voici le schéma officiel pour les **slots** avec un autre exemple simple :



Dans le **template** du parent nous déclarons le fragment HTML à passer : en l'occurrence simplement le texte **Click Me**. Nous déclarons le contenu entre les balises du composant enfant **FancyButton**.

Dans le composant enfant, nous utilisons les balises `<slot>` pour indiquer où insérer le contenu passé, dans l'occurrence le texte **Click Me**. Le rendu final sur le DOM sera donc :

```
<button>
  Click Me
</button>
```

Il est possible de passer n'importe quel contenu HTML comme par exemple une **div** :

```
<template>
  <Enfant>
    <div>
      J'insère du contenu ici pour le passer au composant enfant.
    </div>
  </Enfant>
</template>
```

Il est même possible de passer un autre composant :


```
<template>
  <Enfant>
    <AutreComposant></AutreComposant>
  </Enfant>
</template>
```

Le composant sera également remplacé à l'endroit de **slot**.

4.2 Portée des variables

Le contenu déclaré dans le composant parent a accès aux propriétés du composant parent :

```
<template>
  <span>{{ message }}</span>
  <Enfant>{{ message }}</Enfant>
</template>
```

Ici si le composant parent a une propriété **message** côté **script**, aucun problème pour l'utiliser dans le contenu à passer en **slot** au composant enfant.

En revanche, le contenu n'a pas accès par défaut aux propriétés du composant enfant. Donc dans notre exemple, si la propriété **message** était déclarée dans le composant enfant, nous aurions une erreur.

4.3 Les **slots** nommés

Il est parfois nécessaire de projeter plusieurs contenus à des endroits précis. Dans ce cas, il faut utiliser plusieurs éléments **slot** et les identifier de manière unique. Pour nommer un **slot** côté composant parent, il suffit d'utiliser la directive **v-slot** sur l'élément projeté et de lui passer l'identifiant :

```
<template>
  <Enfant>
    <h1 v-slot:header>Titre de mon site</h1>
    <p v-slot:footer>Qui sommes-nous ?</p>
  </Enfant>
</template>
```

La notation raccourcie de la directive **v-slot** est **#**, nous pouvons donc faire :

```
<template>
  <Enfant>
    <h1 #header>Titre de mon site</h1>
    <p v-#footer>Qui sommes-nous ?</p>
  </Enfant>
</template>
```

Dans notre exemple nous avons défini deux **slot header** et **footer** qui nous permettent de référencer les contenus à projeter. Dans le composant enfant, il suffit d'utiliser l'attribut **name** sur un **slot** pour indiquer que le contenu référencé doit être projeté à cet endroit :

```
<template>
  <header>
    <slot name="header"></slot>
  </header>
  <footer>
    <slot name="footer"></slot>
  </footer>
</template>
```

Il est également possible de créer des identifiants dynamiques pour les `slots` dans les composants parents :

```
<template v-slot:[nomDynamique]>
  Du contenu HTML
</template>
```

Vous pouvez également utiliser la notation raccourcie :

```
<template #[nomDynamique]>
  Du contenu HTML
</template>
```

4.4 Contenu par défaut

Il est également possible de définir un contenu par défaut si aucun contenu n'est passé.

Il suffit de mettre du contenu par défaut entre les balises `<slot>` dans le composant enfant pour que celui-ci soit considéré comme contenu par défaut :

```
<template>
  <slot name="header">Mon titre par défaut si aucun contenu n'est passé</slot>
</template>
```

4.5 Code de la vidéo

Voici le code de la vidéo :

5 Portes des machines à sous

5.1 Passer des données aux slots

Nous avons vu que par défaut, le contenu des `slots` ne peut accéder aux propriétés déclarées dans le composant enfant.

Il est cependant possible de passer des valeurs aux contenus des `slots` depuis le composant enfant. Pour ce faire, il faut passer des attributs aux `slots`, exactement de la même manière que pour les `props` aux composants. Bien sûr, cette fois-ci nous sommes dans le composant enfant :

```
<div>
  <slot :prop="uneValeur" :autreProp="42"></slot>
</div>
```

Dans le composant parent, nous pouvons utiliser ces propriétés passées depuis le composant enfant dans les contenus :

```
<Enfant v-slot="slotProps">
  {{ slotProps.prop }} {{ slotProps.autreProp }}
</Enfant>
```

Cela fonctionne également avec les **slots** nommés :

```
<Enfant>
  <template #header="headerProps">
    {{ headerProps }}
  </template>

  <template #default="defaultProps">
    {{ defaultProps }}
  </template>

  <template #footer="footerProps">
    {{ footerProps }}
  </template>
</Enfant>
```

Dans le composant enfant :

```
<slot name="header" :uneProp="uneValeur"></slot>

<slot name="footer" :uneProp2="autreVal"></slot>
```

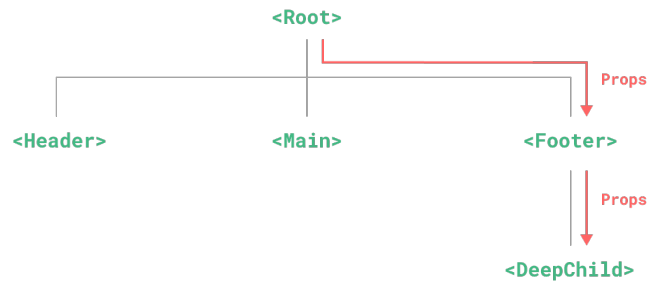
5.2 Code de la vidéo

Voici le code de la vidéo :

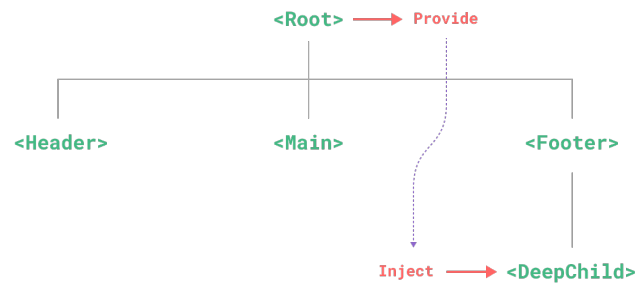
6 Provide et Inject

6.1 Passer des données à travers plusieurs composants

Supposons que vous vouliez passer des données à des composants enfants éloignés dans l'arbre. Vous pouvez utiliser des **props** sur plusieurs niveaux :



C'est d'ailleurs ce que nous avons fait dans le projet. Mais supposons que les composants ciblés soient à 3 ou 4 voir plus niveaux du composant contenant les données et que les composants intermédiaires n'étaient pas besoin de ces données. Comment faire pour passer des données sans devoir déclarer les mêmes **props** dans tous les composants ancêtres des composants cibles ? C'est là qu'interviennent **provide** et **inject** :



6.2 La fonction **provide()**

La fonction **provide()** permet de mettre à disposition des données réactives ou non à tous les composants descendants du composant l'utilisant. La syntaxe est :

```

<script setup>
import { provide } from 'vue';

provide('clé', valeur);
</script>
  
```

La clé d'injection doit être une chaîne de caractères mais la valeur peut être une valeur en dur ou une propriété réactive :

```

import { ref, provide } from 'vue';

const compteur = ref(0);
provide('compteur', compteur);
  
```

6.3 Fournir des valeurs globalement

Pour fournir des données à toute l'application et non seulement aux composants descendants d'un composant, vous pouvez utiliser la méthode `provide()` sur `app` dans `main.ts` :

```
import { createApp } from 'vue';
import App from './App.vue';

const app = createApp(App);

app.provide('API_URL', 'https://restapi.fr/api');

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

6.4 Injecter les données avec la fonction `inject()`

La fonction `inject()` permet de récupérer les données fournies par `provide` dans un composant descendant :

```
<script setup>
import { inject } from 'vue';

const compteur = inject('compteur');
</script>
```

Comme nous utilisons `TypeScript`, il faut taper la valeur reçue en utilisant un type générique :

```
const compteur = inject<number>('compteur');
```

En effet, sans ça `TypeScript` n'a aucun moyen de savoir le type de la valeur retournée par la fonction `inject()`.

Lorsque vous êtes certain que la valeur fournie par `provide()` sera présente, et qu'elle ne peut donc pas être `undefined`, vous pouvez l'indiquer à `TypeScript` en utilisant `!` :

```
const compteur = inject<number>('compteur')!;
```

6.5 Modification des propriétés réactives

Lorsque vous utilisez `provide/inject`, il est recommandé de laisser toutes les modifications des propriétés réactives dans le composant qui utilise `provide()`. Autrement dit, retenez la règle qu'il est interdit de modifier une propriété réactive fournie par `provide()` dans un composant descendant et qu'il faut le faire dans le composant parent.

`Vue.js` met à disposition la fonction `readonly()` pour ne pas oublier de respecter cette règle. Si vous voulez modifier des propriétés réactives depuis un composant enfant il faut passer une fonction avec `provide` qui sera appelée par le composant enfant :

```

<script setup>
import { provide, ref, readonly } from 'vue';

const ville = ref('Paris');

function majVille() {
  ville.value = 'Nice';
}

provide('ville', readonly({
  ville,
  majVille
}));
</script>

```

Et dans le composant descendant :

```

<script setup>
import { inject } from 'vue';

const { ville, majVille } = inject<{ ville: string; majVille: () => void }>('ville');
</script>

<template>
  <button @click="majVille">{{ ville }}</button>
</template>

```

6.6 Code de la vidéo

Voici le code de la vidéo :