Les bases des composants

Ibrahim ALAME

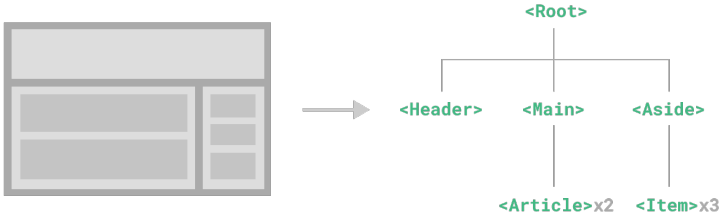
14/02/2023

1 Introduction aux composants

1.1 Les composants

Nous avions déjà commencé à aborder les composants : ils permettent de créer des briques d'UI réutilisables et indépendantes. Nous avions également vu qu'ils étaient organisés de façon hiérarchique en arbre de composants.

Le composant au sommet de l'arbre étant le composant racine (root en anglais). Ce schéma montre que l'interface utilisateur de gauche est découpée en plusieurs composants : un composant pour l'en-tête, un composant pour la partie principale à gauche et un composant pour la partie droite :



Remarquez que des composants sont réutilisés plusieurs fois : un composant Article pour les deux articles affichés et un composant Item pour les trois informations affichées à droite.

1.2 Syntaxe des composants

Nous avons vu que les fichiers des composants étaient écrits en PascalCase et devaient terminer par l'extension .vue.

Cette syntaxe permet de créer des composants monofichiers (SFCs) avec une partie template, une partie script et une partie style. Cette syntaxe est rendue possible grâce à Vite et plus précisément au plugin Vue qui va transformer les SFCs lors d'une étape de build.

1.3 Utiliser des composants enfants

Pour utiliser un composant enfant dans un composant parent, il suffit d'importer le ou les composants dans la partie script et de les utiliser dans la partie template.

La syntaxe recommandée pour utiliser un composant enfant est `<ComposantEnfant />`. Cela permet de distinguer en un coup d'œil les composants Vue.js des éléments HTML. Dans notre exemple, nous aurions par exemple dans le composant Root :

```
<script setup>
import Header from './Header.vue';
import Main from './Main.vue';
import Aside from './Aside.vue';
</script>

<template>
  <Header />
  <Main />
  <Aside />
</template>
```

Les composants sont automatiquement exportés grâce à l'utilisation de l'attribut setup sur les balises scripts. Aussi, il n'y a rien à exporter manuellement dans les composants enfants.

2 Composants locaux et globaux

2.1 Enregistrement des composants

Tous les composants Vue.js ont besoin d'être enregistrés pour que Vue puisse savoir où les chercher lorsque vous les utilisez. Il est possible d'enregistrer les composants soit localement, soit globalement.

2.2 Enregistrement global

Pour enregistrer des composants globalement, il faut modifier le fichier main.ts et utiliser la méthode component() sur l'instance retournée par createApp() :

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

```
const app = createApp(App);

app
  .component('ComponentA', ComponentA)
  .component('ComponentB', ComponentB)
  .component('ComponentC', ComponentC)

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

Le premier argument est le nom à donner au composant dans l'application et le deuxième argument est le composant lui-même qui est importé depuis le fichier où il est situé. Ces composants peuvent être utilisés par n'importe quel composant de votre application, d'où le nom d'enregistrement global. Il existe cependant deux problèmes à l'enregistrement global :

1. Le tree-shaking n'est pas possible. Le tree-shaking permet d'enlever automatiquement du build les composants qui ne sont pas utilisés.

2. La relation entre les composants devient difficilement maintenable. Le fait de déclarer tous les composants au même endroit rend difficile le fait de retrouver un composant enfant utilisé dans un composant. Nous n'avons en effet pas accès au chemin dans la partie script du composant parent. Il faut regarder pour chaque composant dans le fichier main.ts où il est déclaré. Pour les grandes applications, cela devient rapidement illisible.

2.3 Enregistrement local

Les composants enregistrés localement sont disponibles uniquement dans le composant qui les importe. Il est donc nécessaire d'importer le même composant enfant dans tous les composants parents qui l'utilisent. Mais c'est beaucoup plus clair de cette manière : vous savez en un coup d'œil quels composants enfants sont utilisés par un composant en regardant ses imports.

Avant l'apparition de la syntaxe script setup dans la version 3.2, les composants enregistrés localement devaient être enregistrés comme ceci :

```
import ComponentA from './ComponentA.vue'

export default {
  components: {
    ComponentA
  },
  setup() {
    // ...
  }
}
```

Aujourd'hui c'est beaucoup plus simple :

```
<script setup>
import ComponentA from './ComponentA.vue'
</script>
```

Le composant est automatiquement enregistré localement !

3 Les props

3.1 Communications entre les composants

Nous avons vu dans les chapitres précédents comment utiliser des composants : comment les définir, comment les instancier, comment fonctionne une architecture en arbre de composants monofichiers etc. Mais il reste un problème : comment faire passer des données le long de notre arbre de composants ?

Comment communiquer des données d'un composant parent vers un composant enfant et d'un composant enfant vers un composant parent ? C'est ce que nous allons voir maintenant.

3.2 Utilisation des attributs props

Vous pouvez passer des valeurs dans le sens parent → enfant en utilisant des props. Nous allons partir d'un exemple simple pour bien comprendre.

Supposons que nous avons deux composants : Parent.vue et Enfant.vue. Voici notre composant Enfant.vue :

```
<template>
  <h3>{{ prenom }}</h3>
</template>

<script setup>
const props = defineProps(['prenom']);
console.log(props.prenom);
</script>
```

La fonction defineProps() permet de déclarer les propriétés récupérées sur le composant enfant depuis le composant parent. Ici nous récupérons prenom. Elles sont directement utilisables sur le template et nous pouvons les récupérer dans la partie script en déclarant une variable props. Dans notre composant Parent.vue :

```
<template>
    <Enfant prenom="Paul" />
</template>

<script>
```

```
import Enfant from './Enfant.vue';
</script>
```

Nous passons de la donnée de manière unidirectionnelle de Parent.vue vers Enfant.vue. Pour ce faire, nous utilisons simplement un attribut prenom et lui donnons pour l'instant une valeur fixe. Nous pouvons définir n'importe quel attribut personnalisé de cette manière, du moment qu'il n'entre pas en collision avec le nom de d'un attribut HTML natif (par exemple style).

Il ne faut jamais modifier une props dans un composant enfant. Nous verrons comment passer des données du composant enfant vers le composant parent.

3.3 Utilisation d'une liaison dynamique

Nous souhaitons maintenant que les données que nous passons soient liées de manière dynamique. C'est-à-dire que nous voulons que le composant enfant puisse recevoir la nouvelle valeur de l'attribut que nous lui passons depuis le composant parent lorsqu'elle change. Pour ce faire il suffit d'utiliser la directive v-bind :

```
<template>
    <Enfant :prenom="prenom" />
</template>

<script>
import Enfant from './Enfant.vue';
import { ref } from 'vue';

const prenom = ref('Jean');
</script>
```

Nous définissons une variable réactive sur notre composant parent prenom. Nous utilisons la directive v-bind pour lier cette propriété réactive à la prop passée au composant enfant.

3.4 Typage des propriétés props

Nous avons vu que nous pouvions utiliser sur le composant enfant une fonction defineProps() et lui passer un tableau avec le nom des props que nous passons depuis le composant parent. Pour plus de sécurité lors du développement, nous pouvons typer les propriétés sur props afin de définir le type de valeur attendu.

Les types possibles sont String, Number, Boolean, Array, Object, Function, Date et Symbol. Par exemple, pour notre attribut prenom nous souhaitons que ce dernier soit de type String. Nous allons donc typer la props en utilisant un objet au lieu d'un tableau dans la fonction defineProps() sur le composant enfant :

```
...
const props = defineProps({
  prenom: String,
```

```
});
</script>
...
```

Si la valeur de prenom passée n'est pas de type String, alors Vue nous affichera un message d'erreur dans la console JavaScript de notre navigateur. En outre, grâce à Volar et TypeScript nous aurons l'auto-complétion des props et le contrôle des types. Vous pouvez également typer une propriété en permettant plusieurs types, par exemple [String, Number] :

```
<script>
//...
const props = defineProps({
  prenom: [String, Number],
});
</script>
...
```

Dans ce cas, cette propriété pourra être une chaîne de caractères ou un nombre.

3.4.1 Utilisation d'un objet pour plus d'options de type

Vous pouvez également utiliser un objet pour accéder à plus d'options de typage et à des validateurs : Vous pouvez rendre un attribut obligatoire avec required :

```
<script>
//...
const props = defineProps({
  prenom: {
        type: String,
        required: true
    }
});
</script>
```

Vous pouvez attribuer une valeur par défaut à un attribut :

```
<script>
//...
const props = defineProps({
  prenom: {
        type: String,
        required: true,
        default: 'Jean'
```

```
    }
});
</script>
```

3.4.2 Valeur par défaut des propriétés props de type Object

Si vous souhaitez passer un objet comme donnée depuis un composant parent à un composant enfant, vous pouvez utiliser les mêmes fonctionnalités, cependant vous devrez effectuer une adaptation. La raison est toujours la même : les objets et les tableaux sont passés par référence et non par valeur en JavaScript. Il est donc nécessaire d'utiliser des fonctions pour fabriquer de nouveaux objets pour chaque instance. Dans le cas contraire, le même objet serait utilisé pour toutes les instances d'un composant enfant. Il faut donc utiliser une fonction pour définir une valeur par défaut à notre objet :

```
titre: {
      type: Object,
      default() {
        return { title: 'Mon super titre' }
      }
},
```

3.5 Nommage des props

Si vos noms de props sont longs, il est recommandé d'utiliser côté composant parent le kebab-case :

```
<Enfant une-longue-prop="test" />
```

Côté composant enfant il faut utiliser le camelCase pour récupérer la prop :

```
<script setup lang="ts">
defineProps({
  uneLongueProp: String
})
</script>
```

3.6 Passer des types statiques autres que des chaînes de caractères

Pour passer des props statiques autre que des chaînes de caractères, il faut utiliser v-bind :

```
<Enfant :nombre="42" :booleen="false" />
```

De même pour les tableaux ou les objets :

```
<Enfant :tableau="[5, 2, 1]" />
```

3.7 Passer un objet de props

Si vous avez de nombreuses props vous pouvez passer un objet en utilisant la notation longue de v-bind :

```
<script>
const objet = {
  id: 1,
  title: 'Un titre'
};
</script>

<template>
  <Enfant v-bind="objet" />
</template>
```

4 Validation des props et utilisation de TypeScript

4.1 Utilisation des validateurs

Vous pouvez créer un validateur personnalisé pour la valeur de votre attribut. Celui-ci doit être une fonction prenant en paramètre la valeur de l'attribut et doit tester cette valeur puis retourner un booléen :

```
defineProps({
  title: {
    type: String,
    validator(value) {
      return value.length > 2;
    }
  }
}
```

4.2 Notation raccourcie pour les booléens

Les props de type booléen ont une notation raccourcie. Par exemple si vous définissez sur le composant enfant :

```
defineProps({
  available: Boolean
})
```

Vous pouvez directement passer le prop de cette manière côté parent :

```
<Enfant available />
```

4.3 Les génériques TypeScript

Les génériques sont une fonctionnalité de TypeScript permettant une grande flexibilité combinée à une sécurité du typage. Lorsque vous débutez avec TypeScript vous rencontrez deux travers :

1. Trop typer en any. Cela vous donne une grande flexibilité mais vous perdez totalement en sécurité et en autocomplétion.
2. Tout typer correctement sans utiliser de génériques. Cela conférera à votre code la sécurité et l'autocomplétion permise par le typage fort mais vous perdrez en flexibilité, ce qui provoque beaucoup de duplications et de lourdes dans votre code.

En résumé, l'objectif des génériques est de permettre d'utiliser des éléments (fonctions, classes, interfaces etc) qui peuvent fonctionner avec une diversité de types tout en conservant l'utilité de TypeScript, à savoir la sécurité et l'autocomplétion. Nous allons donner deux exemples de types génériques utilisés nativement.

4.3.1 Le typage des tableaux

Nous avons vu comment typer des tableaux pour qu'ils n'acceptent qu'un seul type en faisant :

```
const tableau: string[] = ['Des', 'chaînes', 'de', 'caractères'];
```

La notation alternative est :

```
const tableau: Array<string> = ['Des', 'chaînes', 'de', 'caractères'];
```

Si vous passez la souris sur Array vous verrez `interface Array<T>`. Cela signifie que Array prend en fait un argument appelé T par convention pour type. L'interface `Array<T>` utilise donc un type générique, nous pouvons passer n'importe quel type comme argument et le tableau devra comporter le type défini. Exemple avec une union de types :

```
const tableau: Array<string | boolean> = ['Des', 'chaînes', 'de', 'caractères', true];
```

Autre exemple en utilisant une interface :

```
interface User {
  name: string;
}
```

```
const tableau: Array<User> = [{name: 'Paul'}, {name: 'Jean'}];
```

Vous commencez à percevoir l'idée de générique : nous passons en argument n'importe quel type à l'interface `Array<T>`, et TypeScript nous obligera à le respecter.

4.3.2 Autre exemple : les promesses

Prenons un exemple avec une promesse :

```
let condition: boolean;

const promesse = new Promise((resolve, reject) => {
  if (condition) {
    resolve(42)
  } else {
    reject('Erreur');
  }
});
```

Dans VS Code, si vous passez la souris sur la constante, vous aurez `Promise<unknown>`. Ici, TypeScript utilise également un générique pour typer la valeur remboursée par la promesse. Autrement dit, il utilise une `interface Promise<T>`, et donc les génériques. Ici, il ne peut pas détecter le type par inférence et préciser donc que la valeur retournée est de type inconnu : unknown. En revanche, si nous passons un argument, TypeScript saura que les valeurs retournées dans les promesses seront du type spécifié :

```
let condition: boolean;

const promesse: Promise<number | string> = new Promise((resolve, reject) => {
  if (condition) {
    resolve(42)
  } else {
    reject('Erreur');
  }
});
```

Ici, nous indiquons à TypeScript les valeurs retournées dans la promesse ce qui permet la sécurité et l'autocomplétion. A savoir que si vous essayez d'utiliser par exemple une méthode disponible sur les tableaux ou les objets sur la valeur de retour, TypeScript vous renverra une erreur. Par exemple :

```
let condition: boolean;
```

```
const promesse: Promise<number | string> = new Promise((resolve, reject) => {
  if (condition) {
    resolve(42)
  } else {
    reject('Erreur');
  }
});

promesse.then((val) => {
  val.map()
})
```

Ici, vous aurez une erreur Property 'map' does not exist on type 'string — number'.

De même, VS Code vous proposera en autocomplétion sur val les méthodes et propriétés partagées par les types nombre et chaînes de caractères. Si vous mettiez `Promise<any>`, TypeScript ne feriez plus de contrôle sur les valeurs de retour, sur les méthodes que l'on tente d'accéder etc.

```
let condition: boolean;

const promesse: Promise<any> = new Promise((resolve, reject) => {
  if (condition) {
    resolve(42)
  } else {
    reject('Erreur');
  }
});

promesse.then((val) => {
  val.map()
})
```

Ici aucun problème lors de la compilation et dans VS Code, pourtant l'accès à la méthode map() provoquera une erreur lors de l' ;exécution.

4.4 Utilisation de TypeScript avec les props

Vous pouvez également utiliser la syntaxe TypeScript au lieu de passer un objet à props, dans ce cas on utilise un type générique

```
<script setup lang="ts">
defineProps<{
```

```
  prenom?: string
  age?: number
}>()
</script>
```

Remarquez que nous utilisons un type générique et non plus un argument passé à definePropos(). Le plus souvent nous utiliserons des interfaces de cette manière ;

```
<script setup lang="ts">
interface Props {
  prenom: string
  age?: number
}

const props = definePropos<Props>()
</script>
```

5 Les événements

5.1 Communiquer des composants enfants vers les composants parents

Nous allons maintenant nous intéresser au sens de communication inverse : des composants enfants vers les composants parents. Pour communiquer dans ce sens, nous utilisons les événements personnalisés. Il faut déclarer l'événement sur le composant enfant et l'écouter sur le composant parent. Dans le template du composant parent, nous enregistrons l'écoute d'un événement personnalisé avec la directive v-on :

```
<Enfant
  ...
  @un-evenement="gestionnaire"
 />
```

Dans la partie script du composant enfant, nous déclarons l'événement avec defineEmits() :

```
const emit = defineEmits(['un-evenement']);
```

Nous pouvons émettre l'événement côté script en utilisant :

```
emit('un-evenement');
```

5.2 Émettre des événements sans utiliser defineEmits()

Côté composant enfant, vous pouvez émettre directement des événements sans passer par defineEmits(), en utilisant `$emit()` dans le template :

```
<button @click="$emit('unEvenement')">Cliquer</button>
```

Il faut utiliser le camelCase pour le nom de l'événement. Côté composant, parent, cela ne change pas, nous utilisons v-on pour écouter l'événement écrit en kebab-case :

```
<Enfant @un-evenement="gestionnaire" />
```

5.3 Passer des arguments

Vous pouvez passer des arguments lors de l'émission d'un événement. Il suffit de les passer en argument dans le composant enfant :

```
<button @click="$emit('unEvenement', 42, 'unEvenement')">Cliquer</button>
```

Côté composant parent vous pouvez les récupérer dans le gestionnaire d'événement :

```
<Enfant @un-evenement="gestionnaire" />
```

Par exemple :

```
function gestionnaire(arg1, arg2) {
  console.log(arg1, arg2);
}
```

5.4 Validation des évenements

Comme pour les props, il est possible de valider les événements, même si c'est assez peu utilisé. Par exemple :

```
<script setup>
const emit = defineEmits({
  submit: ({ email, password }) => {
    if (email && password) {
      return true;
    } else {
      return false;
    }
  }
```

```
})

function submitForm(email, password) {
  emit('submit', { email, password })
}
</script>
```

Ici notre événement submit prend en argument un objet { email, password }. La fonction de validation définie dans defineEmits() reçoit cet argument dans une fonction de rappel. Cette fonction de rappel retourne true si l'événement est valide et false sinon. Ici, il est considéré que l'événement est valide si email et password sont définis.

Nous ne détaillerons pas davantage car ce type de validation est très rarement utilisé, et seulement en développement. Nous verrons comment valider des formulaires plus tard.

5.5 Syntaxe recommandée : utilisation de defineEmits avec TypeScript

Dans la suite de la formation nous utiliserons ce qui est recommandé par Vue officiellement : déclarer explicitement les événements utilisés avec defineEmits() en utilisant TypeScript. Comme pour les props, il suffit de passer un générique à la fonction pour déclarer les types des événements utilisés :

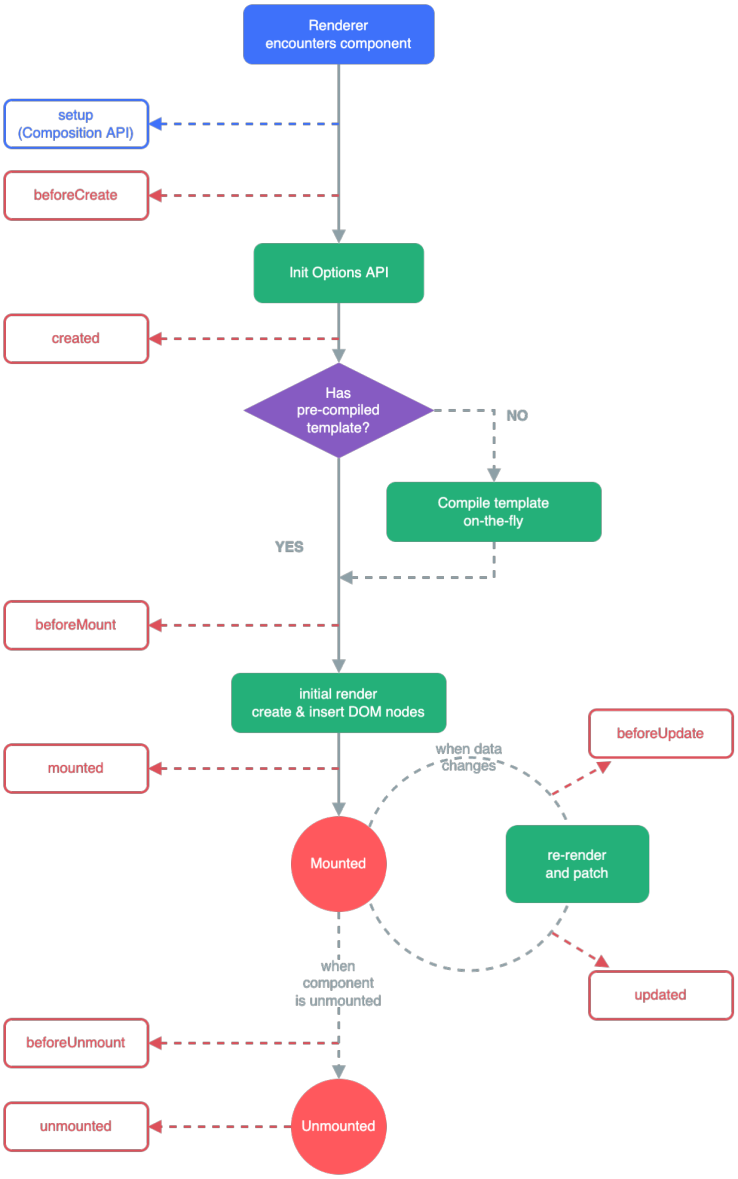
```
const emit = defineEmits<{
  (e: 'change', id: number): void;
  (e: 'update', value: string): void;
}>()
```

Ici nous déclarons un événement change qui contient un argument id de type number et un événement update qui contient un argument value de type chaîne de caractères.

6 Le cycle de vie d'un composant

6.1 Le cycle de vie des composants

Les composants Vue.js passent par différents stades de leur initialisation, à l'affichage sur le DOM, puis à leurs mises à jour et enfin à leur destruction et enlèvement du DOM.



Lors de ces étapes de leur cycle, des fonctions peuvent être appelées pour effectuer des tâches. Ces fonctions sont appelées des lifecycle hooks (littéralement des accroches du cycle de vie).

Les hooks les plus utilisés sont :

- onMounted() : appelée juste après que le composant soit monté (donc lorsque le composant et ses descendants sont affichés sur le DOM).
- onUpdated() : appelée lorsque le composant a été mis à jour et qu'un changement sur le DOM a été effectué.

- onUnmounted() : appelée après que tous les effets réactifs aient été stoppés et lorsque le composant va être retiré du DOM. On l'utilise pour faire des opérations de nettoyage.

Les autres hooks sont :

- onBeforeMount() : qui est appelée lorsque le composant a été initialisé (notamment son état réactif) mais lorsqu'aucun élément n'a encore été créé sur le DOM.
- onBeforeUpdate() : appelée juste avant que le composant n'ait mis à jour son DOM suite à un changement de son état réactif.
- onBeforeUnmount() : appelée lorsque le composant est encore fonctionnel mais va être détruit.
- onErrorCaptured() : appelée lorsqu'une erreur se propage depuis un composant enfant.

Nous verrons les autres hooks qui ont des cas avancés d'utilisations : onRenderTracked(), onRenderTriggered(), onActivated(), onDeactivated() et onServerPrefetch(). Ils sont relatifs au développement, aux composants keepalive et au SSR.

6.2 Utiliser un hook

Pour utiliser un hook c'est très simple, il suffit d'utiliser la fonction de hook adéquate et de lui passer une fonction de rappel contenant les tâches à exécuter lorsque l'état est atteint :

```
<script setup>
import { onMounted } from 'vue'

onMounted(() => {
  console.log('Le composant est présent sur le DOM.')
})
</script>
```

7 Les références de template

7.1 Les références de templates

Parfois, vous voudrez accéder directement à un élément du DOM pour effectuer des opérations particulières. Dans ce cas, Vue.js propose de créer une référence côté template avec l'attribut ref :

```
<input ref="input">
```

L'élément du DOM référencé devient alors accessible depuis le script :

```
<script setup>
import { ref, onMounted } from 'vue';

```

```
const input = ref(null);

onMounted(() => {
  input.value.focus();
})
</script>
```

Le code précédent récupère le champ en utilisant la référence déclarée et va ensuite utiliser la méthode focus() permettant de cibler l'élément. Nous sommes sûr que le champ a été créé sur le DOM, car nous utilisons le hook onMounted() que nous avons vu dans la leçon précédente. Nous sommes obligé d'utiliser le hook onMounted() car nous manipulons un élément du DOM et il faut donc que celui-ci soit créé sur le DOM avant que nous y accédions.

7.2 Utilisation des refs avec v-for

Lorsque nous utilisons ref avec la directive v-for, il faut que la référence côté script contienne un tableau qui contiendra les éléments :

```
<script setup>
import { ref, onMounted } from 'vue';

const list = ref([1, 2, 3, 4]);

const itemRefs = ref([]);

onMounted(() => console.log(itemRefs.value));
</script>

<template>
  <ul>
    <li v-for="item in list" ref="itemRefs">
      {{ item }}
    </li>
  </ul>
</template>
```

7.3 Utilisation des refs avec des composants

L'attribut ref peut également être utilisé sur les composants enfants. Par exemple :

```
<script setup>
import { ref, onMounted } from 'vue';
import Enfant from './Enfant.vue';

const enfant = ref(null);

onMounted(() => {
  // enfant.value contient l'élément Enfant
});
</script>

<template>
  <Enfant ref="enfant" />
</template>
```

Attention ! Il n'est pas possible par défaut d'accéder aux propriétés des composants enfants en utilisant une référence car ces propriétés sont privées par défaut. Il faut exposer, avec defineExpose(), les propriétés auxquelles vous souhaitez accéder dans le composant parent, depuis le composant enfant :

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

const a = 21;
const b = ref(42);

defineExpose({
  a,
  b
})
</script>
```

En utilisant defineExpose() nous rendons accessibles les deux variables depuis le composant parent qui peut ensuite les récupérer sur enfant.value : enfant.value.a et enfant.value.b.

7.4 Utilisation de TypeScript

Bien entendu, il faut également typer avec TypeScript les références. Nous allons voir comment le faire.

7.4.1 Typer les références contenant des éléments HTML

Les références doivent être typées avec une union de types entre null et le type d'élément HTML contenu dans la ref. Par exemple, pour un champ :

```
<script setup lang="ts">
import { ref, onMounted } from 'vue';

const el = ref<HTMLInputElement | null>(null);

onMounted(() => {
  el.value?.focus();
});
</script>

<template>
  <input ref="el" />
</template>
```

Ici nous passons en type générique `<HTMLInputElement | null>` à ref() pour lui indiquer que la référence peut être null (lorsque l'élément n'est pas encore sur le DOM) ou alors un élément de type HTMLInputElement. Notez également qu'il faut indiquer que la référence est null lors de l'initialisation du composant en passant null comme valeur initiale à ref().

L'opérateur de chaînage optionnel ?. est un opérateur JavaScript qui permet de lire la valeur d'une propriété sans avoir à valider expressément que chaque référence dans la chaîne n'est ni null ni undefined.

Ici nous avons besoin d'utiliser cet opérateur car el est null avant le montage du composant sur le DOM. La référence peut également être null dans d'autres cas, par exemple en utilisant la directive v-if et si la condition n'est pas remplie.

7.4.2 Typer les références contenant des composants

Pour les composants, il faut typer de cette manière :

```
<script setup lang="ts">
import Enfant from './Enfant.vue';

const modal = ref<InstanceType<typeof Enfant> | null>(null);

const openModal = () => {
  modal.value?.open();
};
</script>
```

- `typeof` : permet de récupérer le type d'un élément, ici le type du composant enfant.
- `InstanceType` : permet de créer un type à partir d'un type d'une instance de fonction constructrice.
- `ref<InstanceType<typeof Enfant> | null>` : nous récupérons l'instance du composant, récupérons son type avec typeof, construisons le type de la fonction constructrice du composant à partir de cette instance du composant. Nous indiquons également que la référence peut être null.

8 Utilisation des liaisons et des directives sur les composants

8.1 Utilisation des classes sur les composants

Lorsque vous utilisez l'attribut class avec un composant enfant, ces classes seront ajoutées sur l'élément racine, c'est-à-dire l'élément imbriquant tous les autres éléments du template du composant enfant. Par exemple, si le composant enfant a pour template :

```
<div class="classe1">
  <p>Bonjour !</p>
</div>
```

Et que nous utilisons l'attribut class dans le template du composant parent, sur le composant enfant :

```
<Enfant class="classe2" />
```

Vue.js résolvera les classes en les fusionnant :

```
<div class="classe1 classe2">
  <p>Bonjour !</p>
</div>
```

Attention ! Si le composant enfant n'a pas d'élément racine imbriquant les autres éléments du template, il faudra utiliser la valeur spéciale $attrs.class mise à disposition par Vue.js. Ainsi, nous devrons faire :

```
<p :class="$attrs.class">Bonjour </p>
<span>C'est le template du composant enfant</span>
```

Ici nous disons à Vue.js d'appliquer les classes définies dans le composant parent sur l'élément paragraphe. C'est tout à fait logique : Vue.js ne sait pas où appliquer les classes si nous ne lui indiquons pas, et donc par défaut il ne va rien appliquer.

8.2 Utilisation de la directive v-for avec des composants

La directive v-for est utilisable directement sur tout composant :

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

Pour passer des données au composant enfant, il faut utiliser la liaison de données avec v-bind :

```
<Composant v-for="(item, index) in items" :item="item" :index="index" :key="item.id"  />
```

8.3 Utilisation de la directive v-if avec des composants

Il n'y a rien de particulier pour la directive v-if que vous pouvez appliquer sur tous les composants :

```
<Composant v-if="active" />
```