Vue.js : API Composition

Achref El Mouelhi

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

elmouelhi.achref@gmail.com

Plan

Plan

Plan

Vue.js

Remarque

Depuis le début de ce cours, on utilisait l'**Option API**.

Vue.js

Remarque

Depuis le début de ce cours, on utilisait l'**Option API**.

Composition API

- Simplification de l'écriture des composants
- Apparu dans **Vue.js 3**
- Alternative à l'**Option API**
- Proposant deux nouvelles écritures

Vue.js

Exemple

- Créons un composant `CompteurView`
- Associons une route `/compteur` à ce composant

Commençons par le contenu suivant pour `CompteurView`

```
<template>
    <h1>Compteur </h1>
    <button @click="decrementer">-</button>
    {{ counter }}
    <button @click="incrementer">+</button>
</template>

<script>
export default {
    data() {
        return {
            counter: 0
        }
    },
    methods: {
        incrementer() {
            this.counter++
        },
        decrementer() {
            this.counter--
        }
    }
}
</script>
```

Commençons par le contenu suivant pour `CompteurView`

```
<template>
    <h1>Compteur </h1>
    <button @click="decrementer">-</button>
    {{ counter }}
    <button @click="incrementer">+</button>
</template>

<script>
export default {
    data() {
        return {
            counter: 0
        }
    },
    methods: {
        incrementer() {
            this.counter++
        },
        decrementer() {
            this.counter--
        }
    }
}
</script>
```

Lancez l'application et vérifiez que le compte s'incrémente et se décrémente en cliquant sur les boutons + et −.

Vue.js

Objectif

Transformer le construction du composant d'**Option API** vers **Composition API**.

Vue.js

Dans la partie `script`, commençons par exporter le composant et déclarer une fonction `setup()`

```
<script>
export default {
    setup() {
    }
}
</script>
```

Vue.js

Dans la partie `script`, commençons par exporter le composant et déclarer une fonction `setup()`

```
<script>
export default {
    setup() {
    }
}
</script>
```


Explication

- La fonction `setup()` sera exécutée avant la création du composant.
- Le mot-clé `this` est donc inutilisable dans `setup` (parce que le composant n'a pas encore été créé).

Vue.js

Tout attribut déclaré dans `data` doit être déclaré initialisé avec `ref` dans `setup()` : `ref` retourne un objet

```
<script>

import { ref } from 'vue';

export default {
    setup() {
        const counter = ref(0)
    }
}
</script>
```

Vue.js

Pour qu'un attribut soit utilisable dans `template`, il faut le retourner

```
<script>

import { ref } from 'vue';

export default {
    setup() {
        const counter = ref(0)

        return {
            counter,
        }
    }
}
</script>
```

Les méthodes `incrementer` et `decrementer` doivent être déclarées dans `setup` et retournées, pour manipuler la valeur de `counter`, on écrit `counter.value`

```
<script>

import { ref } from 'vue';

export default {
    setup() {
        const counter = ref(0)

        const incrementer = () => {
            counter.value++
        }
        const decrementer = () => {
            counter.value--
        }
        return {
            counter,
            incrementer,
            decrementer
        }
    }
}
</script>
```

Rien à changer dans `template`

```
<template>
    <h1>Compteur </h1>
    <button @click="decrementer">-</button>
    {{ counter }}
    <button @click="incrementer">+</button>
</template>
```

Rien à changer dans `template`

```
<template>
    <h1>Compteur </h1>
    <button @click="decrementer">-</button>
    {{ counter }}
    <button @click="incrementer">+</button>
</template>
```

Lancez l'application et vérifiez que le compte s'incrémente et se décrémente en cliquant sur les boutons + et −.

Rien à changer dans `template`

```
<template>
    <h1>Compteur </h1>
    <button @click="decrementer">-</button>
    {{ counter }}
    <button @click="incrementer">+</button>
</template>
```

Lancez l'application et vérifiez que le compte s'incrémente et se décrémente en cliquant sur les boutons + et −.

Remarque

Pas besoin d'écrire `counter.value` dans `template`.

Vue.js

En utilisant la balise <script setup>

- Code encore plus simple, plus lisible
- Plus besoin d'exporter
- Plus besoin de retourner les méthodes et attributs déclarés précédemment dans la fonction `setup()`

Vue.js

Commençons par déclarer <script setup>

```
<script setup>

</script>
```

Déclarons nos attributs et méthodes directement dans <script setup>

```
<script setup>

import { ref } from 'vue';

const counter = ref(0)

const incrementer = () => {
    counter.value++
}
const decrementer = () => {
    counter.value--
}
</script>
```

Déclarons nos attributs et méthodes directement dans <script setup>

```
<script setup>

import { ref } from 'vue';

const counter = ref(0)

const incrementer = () => {
    counter.value++
}
const decrementer = () => {
    counter.value--
}
</script>
```

Lancez l'application et vérifiez que le compte s'incrémente et se décrémente en cliquant sur les boutons + et −.

Déclarons nos attributs et méthodes directement dans <script setup>

```
<script setup>

import { ref } from 'vue';

const counter = ref(0)

const incrementer = () => {
    counter.value++
}
const decrementer = () => {
    counter.value--
}
</script>
```

Lancez l'application et vérifiez que le compte s'incrémente et se décrémente en cliquant sur les boutons + et −.

Remarque

`setup` ⇒ pas accès à `this` ⇒ simplification du code avec les fonctions fléchées.

Vue.js

Exercice

- Déplacez le composant `Calculette` dans `Views` et renommez le `Calculette.vue`.
- Définissez une route pour ce composant et ajoutez le au menu.
- Modifier la partie `script` du composant `Calculette` pour le transformer en **Composition API**.

Vue.js

La partie `script`

```
<script setup >
import { ref } from "vue"

const valeur1 = ref(0)
const valeur2 = ref(0)
const resultat = ref(0)

const calculerResultat = () => {
    resultat.value = valeur1.value + valeur2.value;
}
</script>
```

Vue.js

La partie `script`

```
<script setup >
import { ref } from "vue"

const valeur1 = ref(0)
const valeur2 = ref(0)
const resultat = ref(0)

const calculerResultat = () => {
    resultat.value = valeur1.value + valeur2.value;
}
</script>
```

La partie `template`

```
<template>
    <div>
        <label for="valeur1">Valeur 1</label>
        <input type="number" v-model="valeur1" id="valeur1" @input="calculerResultat">
    </div>
    <div>
        <label for="valeur2">Valeur 2</label>
        <input type="number" v-model="valeur2" id="valeur2" @input="calculerResultat">
    </div>
    <div>
        <label for="resultat">Résultat</label>
        <input type="number" v-model="resultat" id="resultat" readonly>
    </div>
</template>
```

Vue.js

Rappel : cycle de vie d'un composant **API Option**

- `beforeCreate` : appelée avant la création du composant. (remplacé par `setup`)
- `created` : appelée après la création du composant. (remplacé par `setup`)
- `beforeMount` : appelée avant que le composant 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

Cycle de vie d'un composant **API Composition** : méthodes préfixées par `on`

- `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`~~ : n'existe plus, remplacée par `setup`
- ~~`created`~~ : n'existe plus, remplacée par `setup`
- `beforeMount` : appelée avant chaque que le composant 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

Ajoutons les méthodes `hooks` suivantes dans la partie `script` de `CompteurView.vue`

```
onBeforeMount(() => {
    console.log('before mount')
})
onMounted(() => {
    console.log('mounted')
})

onBeforeUpdate(() => {
    console.log('before update')
})
onUpdated(() => {
    console.log('updated')
})

onBeforeUnmount(() => {
    console.log('before unmount')
})
onUnmounted(() => {
    console.log('unmounted')
})
```

Vue.js

Ajoutons les méthodes `hooks` suivantes dans la partie `script` de `CompteurView.vue`

```
onBeforeMount(() => {
    console.log('before mount')
})
onMounted(() => {
    console.log('mounted')
})

onBeforeUpdate(() => {
    console.log('before update')
})
onUpdated(() => {
    console.log('updated')
})

onBeforeUnmount(() => {
    console.log('before unmount')
})
onUnmounted(() => {
    console.log('unmounted')
})
```

N'oublions pas les imports suivants

```
import { onMounted, onBeforeMount, onBeforeUpdate, onUpdated, onUnmounted, onBeforeUnmount }
  from 'vue';
```

Vue.js

Vérifiez l'affichage des messages suivants dans la console

```
before mount
mounted
```

Vue.js

Vérifiez l'affichage des messages suivants dans la console

```
before mount
mounted
```


Remarque

La définition d'une méthode `onCreated` ou `onBeforeCreate` génère une erreur.

Vue.js

ref

- permet de créer un objet réactif à partir d'une valeur primitive : `string`, `number`, `boolean` ou un objet...
- retourne un objet contenant un attribut `value`

Reprenons le composant `CompteurView` et modifions l'objet `counter`

```
import { ref } from 'vue';

const compteur = { valeur: 0, etat: 'nul' }
const counter = ref(compteur)
```

Reprenons le composant `CompteurView` et modifions l'objet `counter`

```
import { ref } from 'vue';

const compteur = { valeur: 0, etat: 'nul' }
const counter = ref(compteur)
```

Modifions l'accès à la valeur des attributs de l'objet

```
const incrementer = () => {
    counter.value.valeur++
}
const decrementer = () => {
    counter.value.valeur--
}
```

Reprenons le composant `CompteurView` et modifions l'objet `counter`

```
import { ref } from 'vue';

const compteur = { valeur: 0, etat: 'nul' }
const counter = ref(compteur)
```

Modifions l'accès à la valeur des attributs de l'objet

```
const incrementer = () => {
    counter.value.valeur++
}
const decrementer = () => {
    counter.value.valeur--
}
```

Remarque

Accès long et compliqué aux attributs.

Vue.js

Question

Et si la valeur, à rendre réactive, était un objet ou un tableau (pas une valeur primitive) ?

Vue.js

Question

Et si la valeur, à rendre réactive, était un objet ou un tableau (pas une valeur primitive) ?

Solution : `reactive`

- permet de créer un objet réactif à partir d'une valeur non-primitive : objet
- plus besoin d'utiliser `value`

Vue.js

Remplaçons `ref` **par** `reactive` **dans** `CompteurView`

```
import { reactive } from 'vue';

const compteur = { valeur: 0, etat: 'nul' }
const counter = reactive(compteur)
```

Vue.js

Remplaçons `ref` **par** `reactive` **dans** `CompteurView`

```
import { reactive } from 'vue';

const compteur = { valeur: 0, etat: 'nul' }
const counter = reactive(compteur)
```

L'accès aux attributs se fait sans `value`

```
const incrementer = () => {
    counter.valeur++
}
const decrementer = () => {
    counter.valeur--
}
```

Vue.js

Remplaçons `ref` par `reactive` dans `CompteurView`

```
import { reactive } from 'vue';

const compteur = { valeur: 0, etat: 'nul' }
const counter = reactive(compteur)
```

L'accès aux attributs se fait sans `value`

```
const incrementer = () => {
    counter.valeur++
}
const decrementer = () => {
    counter.valeur--
}
```

Et le template

```
<template>
    <h1>Compteur </h1>
    <button @click="decrementer">-</button>
    {{ counter.valeur }}
    <button @click="incrementer">+</button>
</template>
```

Vue.js

	ref	reactive
Type de données accepté	Valeurs primitives et objets	Objets et tableaux
Accès aux valeurs	Accès via .value	Accès direct
Réassignation complète de l'objet	Possible	Impossible

ref reste réactive après réassignation

```
import { ref } from 'vue';

let counter = ref({ valeur: 0, etat: 'nul' })

const incrementer = () => {
    counter.value = { valeur: 1, etat: 'positif' }
}
const decrementer = () => {
    counter.value = { valeur: -1, etat: 'négatif' }
}
```

ref reste réactive après réassignation

```
import { ref } from 'vue';

let counter = ref({ valeur: 0, etat: 'nul' })

const incrementer = () => {
    counter.value = { valeur: 1, etat: 'positif' }
}
const decrementer = () => {
    counter.value = { valeur: -1, etat: 'négatif' }
}
```

reactive ne reste pas réactive après réassignation

```
import { reactive } from 'vue';

let counter = reactive({ valeur: 0, etat: 'nul' })

const incrementer = () => {
    counter = { valeur: 1, etat: 'positif' }
}
const decrementer = () => {
    counter = { valeur: -1, etat: 'négatif' }
}
```

Vue.js

Pour déterminer si une variable est une référence, réactive ou simple

```
onMounted(() => {
    console.log(isReactive(counter))
    // affiche true

    console.log(isRef(counter))
    // affiche false
})
```

Vue.js

Pour déterminer si une variable est une référence, réactive ou simple

```
onMounted(() => {
    console.log(isReactive(counter))
    // affiche true

    console.log(isRef(counter))
    // affiche false
})
```


Solution

```
import { reactive, onMounted, isReactive, isRef } from 'vue';
```

Vue.js

Références template

permettent de

- référencer un élément **HTML**
- manipuler ses propriétés **JavaScript**

Vue.js

Définissons une référence sur le bouton du composant `CompteurView`

```
<template>
    <h1>Compteur : {{ counter.etat }}</h1>
    <button @click="decrementer">-</button>
    {{ counter.valeur }}
    <button @click="incrementer" ref="bouton">+</button>
</template>
```

Vue.js

Définissons une référence sur le bouton du composant `CompteurView`

```
<template>
    <h1>Compteur : {{ counter.etat }}</h1>
    <button @click="decrementer">-</button>
    {{ counter.valeur }}
    <button @click="incrementer" ref="bouton">+</button>
</template>
```

Dans `script`**, définissons une référence du même nom**

```
const bouton = ref(null);
```

Vue.js

Définissons une référence sur le bouton du composant `CompteurView`

```
<template>
    <h1>Compteur : {{ counter.etat }}</h1>
    <button @click="decrementer">-</button>
    {{ counter.valeur }}
    <button @click="incrementer" ref="bouton">+</button>
</template>
```

Dans `script`**, définissons une référence du même nom**

```
const bouton = ref(null);
```

Ainsi, nous pouvons manipuler toutes les propriétés JavaScript de ce bouton

```
onMounted(() => {
    console.log(bouton.value.innerHTML)
})
```

Vue.js

Dans Vue.js 3.5, `useTemplateRef` a été introduit pour distinguer les références de template des autres références

```
const bouton = useTemplateRef("bouton");
```

Vue.js

Dans Vue.js 3.5, `useTemplateRef` a été introduit pour distinguer les références de template des autres références

```
const bouton = useTemplateRef("bouton");
```

Sans oublier l'import

```
import { useTemplateRef } from 'vue';
```

Vue.js

toRef

- permet de récupérer une valeur d'un objet réactif
- retourne une référence

Vue.js

Dans `script`, créons deux références `valeur` et `etat` depuis l'objet `counter`

```
const valeur = toRef(counter, 'valeur')
const etat = toRef(counter, 'etat')
```

Vue.js

Dans `script`**, créons deux références** `valeur` **et** `etat` **depuis l'objet** `counter`

```
const valeur = toRef(counter, 'valeur')
const etat = toRef(counter, 'etat')
```

Dans `template`**, nous pouvons utiliser directement** `valeur` **et** `etat`

```
<template>
    <h1>Compteur : {{ etat }}</h1>
    <button @click="decrementer">-</button>
    {{ valeur }}
    <button @click="incrementer" ref="bouton">+</button>
</template>
```

Vue.js

toRefs

- similaire au concept de décomposition introduit dans **ES6**
- permet de décomposer un objet réactif

Vue.js

Dans `script`, remplaçons les deux lignes suivantes

```
const valeur = toRef(counter, 'valeur')
const etat = toRef(counter, 'etat')
```

Vue.js

Dans `script`, remplaçons les deux lignes suivantes

```
const valeur = toRef(counter, 'valeur')
const etat = toRef(counter, 'etat')
```

Par

```
const { valeur, etat } = toRefs(counter)
```

Vue.js

Affichons un objet réactif et vérifions qu'il s'agit d'un proxy avec des méta-données

```
console.log(counter)
```

Vue.js

Affichons un objet réactif et vérifions qu'il s'agit d'un proxy avec des méta-données

```
console.log(counter)
```


Question

Comment afficher que les valeurs que nous avons définies dans l'objet ?

Vue.js

Affichons un objet réactif et vérifions qu'il s'agit d'un proxy avec des méta-données

```
console.log(counter)
```


Question

Comment afficher que les valeurs que nous avons définies dans l'objet ?

Solution avec `toRaw`

```
console.log(toRaw(counter))
```

Exercice

- Modifiez le composant `CompteurView` pour afficher le contenu de l'attribut `etat` (voir ci-dessous).
- `etat` doit contenir
  - `positif` si `valeur` est supérieur à zéro,
  - `négatif` si `valeur` est inférieur à zéro,
  - `nul` sinon.

Le template à utiliser

```
<template>
    <h1>Compteur : {{ counter.etat }}</h1>
    <button @click="decrementer">-</button>
    {{ counter.valeur }}
    <button @click="incrementer">+</button>
</template>
```

Vue.js

Solution avec `onUpdated`

```
<script setup>

import { reactive, onUpdated } from 'vue';

const compteur = { valeur: 0, etat: 'nul' }
const counter = reactive(compteur)

const incrementer = () => {
    counter.valeur++
}

const decrementer = () => {
    counter.valeur--
}

onUpdated(() => {
    if (counter.valeur > 0) {
        counter.etat = 'positif'
    } else if (counter.valeur < 0) {
        counter.etat = 'négatif'
    } else {
        counter.etat = 'nul'
    }
})
</script>
```

Vue.js

Solution avec `watch`

```
<script setup>

import { reactive, watch } from 'vue';

const compteur = { valeur: 0, etat: 'nul' }
const counter = reactive(compteur)

const incrementer = () => {
    counter.valeur++
}

const decrementer = () => {
    counter.valeur--
}

watch(() => counter.valeur, (nouvelleValeur) => {
    if (nouvelleValeur === 0) {
        counter.etat = 'nul';
    } else if (nouvelleValeur > 0) {
        counter.etat = 'positif';
    } else {
        counter.etat = 'négatif';
    }
});
</script>
```

Vue.js

Question 1

Pourrions-nous observer `counter` à la place de `() => counter.valeur` ?

Vue.js

Question 1

Pourrions-nous observer `counter` à la place de `() => counter.valeur` ?

Réponse

Oui, mais le `watch` sera exécuté deux fois après chaque changement de valeur de `counter.valeur` :

- une fois lorsqu'on incrémente ou on décrémente `counter.valeur`
- et une deuxième fois lorsqu'on réaffecte `counter.etat` suite à l'incrémentation ou la décrémentation de `counter.valeur`.

Vue.js

Question 2

Pourrions-nous remplacer `() => counter.valeur` par `counter.valeur` ?

Vue.js

Question 2

Pourrions-nous remplacer `() => counter.valeur` par `counter.valeur` ?

Réponse

Non, car `watch` n'accepte comme source que :

- une `ref`
- une `reactive`
- un `getter` (comme `() => counter.valeur`)
- un tableau contenant un des trois éléments précédents

Vue.js

Solution avec `watchEffect`

```
<script setup>

import { reactive, watch } from 'vue';

const compteur = { valeur: 0, etat: 'nul' }
const counter = reactive(compteur)

const incrementer = () => {
    counter.valeur++
}

const decrementer = () => {
    counter.valeur--
}

watchEffect(() => {
    if (counter.valeur > 0) {
        counter.etat = 'positif'
    } else if (counter.valeur < 0) {
        counter.etat = 'négatif'
    } else {
        counter.etat = 'nul'
    }
})

</script>
```

Vue.js

Question 1

Comment connaître les variables observées par `watchEffect` ?

Vue.js

Question 1

Comment connaître les variables observées par `watchEffect` ?

Réponse

Toutes les valeurs réactives lues dans la callback de `watchEffect` sont des dépendances : elles déclencherons donc la callback de `watchEffect`.

Vue.js

Solution avec `computed`

```
<script setup>

import { reactive, computed } from 'vue';

const compteur = { valeur: 0, etat: 'nul' }
const counter = reactive(compteur)

const incrementer = () => {
    counter.valeur++
}

const decrementer = () => {
    counter.valeur--
}

counter.etat = computed(() => {
    if (counter.valeur == 0) {
        return 'nul'
    } else if (counter.valeur > 0) {
        return 'positif'
    } else {
        return 'négatif'
    }
})
</script>
```

Vue.js

Reprenons le `script` du composant `CalculetteView`, la fonction `calculerResultat` est exécuté à chaque saisie de valeur dans l'une des deux zones de texte

```
<script setup >
import { ref } from "vue"

const valeur1 = ref(0)
const valeur2 = ref(0)
const resultat = ref(0)

const calculerResultat = () => {
    resultat.value = valeur1.value + valeur2.value
}
</script>
```

Vue.js

Nous pourrons utiliser `computed` pour mettre à jour le résultat à chaque saisie de valeur dans l'une des deux zones de texte

```
<script setup >
import { ref, computed } from "vue"

const valeur1 = ref(0)
const valeur2 = ref(0)

const resultat = computed(() => {
    return valeur1.value + valeur2.value
})
</script>
```

Vue.js

Nous pourrons utiliser `computed` pour mettre à jour le résultat à chaque saisie de valeur dans l'une des deux zones de texte

```
<script setup >
import { ref, computed } from "vue"

const valeur1 = ref(0)
const valeur2 = ref(0)

const resultat = computed(() => {
    return valeur1.value + valeur2.value
})
</script>
```


Remarque

Pour accéder à la valeur d'une propriété calculée avec `computed` dans la partie `script`, il faut utiliser `.value`. Par exemple : `resultat.value`.

Vue.js

Plus besoin d'écouter d'évènements dans `template`

```
<template>
    <div>
        <label for="valeur1">Valeur 1</label>
        <input type="number" v-model="valeur1" id="valeur1">
    </div>
    <div>
        <label for="valeur2">Valeur 2</label>
        <input type="number" v-model="valeur2" id="valeur2">
    </div>
    <div>
        <label for="resultat">Résultat</label>
        <input type="number" v-model="resultat" id="resultat"
           readonly>
    </div>
</template>
```

Vue.js

Routage

- Dans les composants de type **Composition API**, on ne peut plus utiliser le mot-clé `this`.
- Donc, plus accès aux services `route` et `router`.

Vue.js

Routage

- Dans les composants de type **Composition API**, on ne peut plus utiliser le mot-clé `this`.
- Donc, plus accès aux services `route` et `router`.

Question

Comment faire pour récupérer les paramètres de route, rediriger vers un autre composant... ?

Vue.js

Reprenons le `script` du composant `AdresseView`

```
<script>
export default {
    name: 'AdresseView',
    computed: {
        adresse() {
            return this.$route.query
        }
    },
}
</script>
```

Vue.js

Reprenons le `script` du composant `AdresseView`

```
<script>
export default {
    name: 'AdresseView',
    computed: {
        adresse() {
            return this.$route.query
        }
    },
}
</script>
```

Objectif

Transformer ce composant en **Composition API**.

Vue.js

Commençons par définir la structure `Composition API`

```
<script setup>

const adresse = computed(() => {
    return this.$route.query
})

</script>
```

Vue.js

Commençons par définir la structure `Composition API`

```
<script setup>

const adresse = computed(() => {
    return this.$route.query
})

</script>
```


Remarque

Notre code ne récupère plus les paramètres parce que `this` n'est pas utilisable.

Vue.js

Pour récupérer les paramètres de la requête, commençons par importer `useRoute`

```
<script setup>

import { useRoute } from 'vue-router';
const route = useRoute();

const adresse = computed(() => {
})

</script>
```

Vue.js

Utilisons `route` pour récupérer les paramètres

```
<script setup>

import { useRoute } from 'vue-router';
const route = useRoute();

const adresse = computed(() => {
    return route.query
})

</script>
```

Vue.js

Utilisons `route` pour récupérer les paramètres

```
<script setup>

import { useRoute } from 'vue-router';
const route = useRoute();

const adresse = computed(() => {
    return route.query
})

</script>
```

Vérifiez que les paramètres sont correctement récupérés.

Vue.js

Hypothèse

- Nous voudrions ajouter un bouton `Retour à la page d'accueil` dans `AdresseView`.
- En cliquant sur ce bouton, une redirection s'effectue vers `HomeView`.

Vue.js

Hypothèse

- Nous voudrions ajouter un bouton `Retour à la page d'accueil` dans `AdresseView`.
- En cliquant sur ce bouton, une redirection s'effectue vers `HomeView`.

Même problématique

Plus d'accès à l'objet `this`.

Vue.js

Commençons par ajouter le bouton dans `template` d'`AdresseView`

```
<button @click="backHome">
    Retour à la page d'accueil
</button>
```

Vue.js

Commençons par ajouter le bouton dans `template` d'`AdresseView`

```
<button @click="backHome">
    Retour à la page d'accueil
</button>
```

Ensuite, dans `script`, importons `useRouter`

```
import { useRouter } from 'vue-router';
const router = useRouter();
```

Vue.js

Commençons par ajouter le bouton dans `template` **d'**`AdresseView`

```
<button @click="backHome">
    Retour à la page d'accueil
</button>
```

Ensuite, dans `script`**, importons** `useRouter`

```
import { useRouter } from 'vue-router';
const router = useRouter();
```


Enfin, utilisons `router` **dans** `backHome` **pour rediriger vers** `HomeView`

```
const backHome = () => {
    router.push('home')
}
```

Vue.js

Questions

- Comment récupérer les données envoyées du parent à l'enfant ?
- Comment l'enfant peut envoyer des données à son parent ?

Reprenons le `script` du composant `HelloWorld`

```
<script>
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world",
      nom: null
    }
  },
  props: {
    ville: {
      type: String,
    }
  },
  mounted() {
    this.$refs.name.placeholder = 'Votre nom';
    this.$refs.name.focus();
  },
  methods: {
    sendData() {
      this.$emit('sendData', this.nom)
    }
  }
}
</script>
```

Reprenons le `script` du composant `HelloWorld`

```
<script>
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world",
      nom: null
    }
  },
  props: {
    ville: {
      type: String,
    }
  },
  mounted() {
    this.$refs.name.placeholder = 'Votre nom';
    this.$refs.name.focus();
  },
  methods: {
    sendData() {
      this.$emit('sendData', this.nom)
    }
  }
}
</script>
```

Objectif

Transformer ce composant en **Composition API**.

Vue.js

Commençons par définir la structure de Composition API

```
<script setup>

</script>
```

Vue.js

Transformons les `data` en `ref`

```
<script setup>

import { ref } from 'vue';

const msg = ref("Hello world")
const nom = ref(null)

</script>
```

Vue.js

Mettons à jour le hook `onMounted`

```
<script setup>

import { ref, onMounted } from 'vue';

const msg = ref("Hello world")
const nom = ref(null)

onMounted(() => {
  nom.value.placeholder = 'Votre nom';
  nom.value.focus();
})

</script>
```

Vue.js

Mettons à jour le `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>
  <div>
    <label for="nom">Nom</label>
    <input type="text" id="nom" ref="nom">
    <button @click="envoyer">Envoyer</button>
  </div>
</template>
```

Vue.js

Utilisons `defineProps` pour déclarer les `props`

```
<script setup>

import { ref, onMounted, defineProps } from 'vue';

defineProps({
  ville: String
})

const msg = ref("Hello world")
const nom = ref(null)

onMounted(() => {
  nom.value.placeholder = 'Votre nom';
  nom.value.focus();
})

</script>
```

Vue.js

Impossible d'utiliser une `props` dans `script` sans l'affecter à une variable (ou constante)

```
<script setup>

import { ref, onMounted } from 'vue';

const props = defineProps({
  ville: String
})

const msg = ref("Hello world")
const nom = ref(null)

onMounted(() => {
  nom.value.placeholder = 'Votre nom';
  nom.value.focus();
  console.log(props.ville)
})

</script>
```

Vue.js

Depuis Vue.js 3.2, il est possible de déstructurer les `props`

```
<script setup>

import { ref, onMounted } from 'vue';

const { ville } = defineProps({
  ville: String
})

const msg = ref("Hello world")
const nom = ref(null)

onMounted(() => {
  nom.value.placeholder = 'Votre nom';
  nom.value.focus();
  console.log(ville)
})

</script>
```

Préparons `emit`

```
<script setup>

import { ref, onMounted } from 'vue';

defineProps({
  ville: String
})
const emit = defineEmits(['sendData'])

const msg = ref("Hello world")
const nom = ref(null)

onMounted(() => {
  nom.value.placeholder = 'Votre nom';
  nom.value.focus();
})

</script>
```

Préparons `emit`

```
<script setup>

import { ref, onMounted } from 'vue';

defineProps({
  ville: String
})
const emit = defineEmits(['sendData'])

const msg = ref("Hello world")
const nom = ref(null)

onMounted(() => {
  nom.value.placeholder = 'Votre nom';
  nom.value.focus();
})

</script>
```

Remarque

`defineEmits` ne doit pas être placé dans une fonction.

Utilisons `emit` dans la fonction `envoyer` pour émettre un évènement au parent

```
<script setup>

import { ref, onMounted } from 'vue';

defineProps({
  ville: String
})

const emit = defineEmits(['sendData'])

const msg = ref("Hello world")
const nom = ref(null)

onMounted(() => {
  nom.value.placeholder = 'Votre nom';
  nom.value.focus();
})

const envoyer = () => {
  emit('sendData', nom.value)
}

</script>
```

Vue.js

Considérons le code suivant de `PaysComponent`

```
<template>
    <h1>Pays</h1>
    <div>
        <template v-for="item in villes">
            {{ item }}  
        </template>
    </div>
    <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

Et le code suivant 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

Le code précédent peut être simplifié en utilisant `defineModel` [Vue.js 3.4]

```
<template>
    <li>
        <input type="text" v-model="ville">
    </li>
</template>

<script setup>
const ville = defineModel()
</script>
```

Vue.js

Et si `PaysComponent` **envoyait plusieurs données à** `VilleComponent`

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

La récupération de données dans `VilleComponent` s'effectue ainsi

```
<template>
    <h2>Ville</h2>
    <div>
        Ville : <input v-model="ville" />
        Code postal : <input v-model="codePostal" />
    </div>
</template>

<script setup>
const ville = defineModel('ville')
const codePostal = defineModel('codePostal')
</script>
```

Vue.js

Cliquons sur le bouton `Traduire en français` de `AboutView`, qui permettait de traduire un message défini dans le composant enfant `HelloWorld` en français, et vérifions que le message ne se traduit plus

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

Vue.js

Cliquons sur le bouton `Traduire en français` de `AboutView`, qui permettait de traduire un message défini dans le composant enfant `HelloWorld` en français, et vérifions que le message ne se traduit plus

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


Explication

- `AboutView` n'a plus accès aux `data` de son composant enfant `HelloWorld` depuis sa transformation en **API Composition**.
- Les composants **API Composition** doivent exposer les `data` pour que le parent y ait accès.

Vue.js

Dans le composant enfant `HelloWorld`, exposons la propriété `msg`

```
defineExpose({ msg })
```

Vue.js

Dans le composant enfant `HelloWorld`, exposons la propriété `msg`

```
defineExpose({ msg })
```

Cliquons sur le bouton `Traduire en français` de `AboutView` et vérifions que le message s'affiche de nouveau en français

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

Vue.js

`getCurrentInstance()` de **Vue 3**

- fonction interne exposée par **API composition**
- donne accès à l'instance du composant courant
- équivalent à `this` dans **API Option**
- ne fonctionne que dans `setup()`

Reprenons le `script` du composant `PersonneDetails.vue`

```
<script>
import axios from 'axios';

export default {
    data() {
        return {
            personne: {}
        }
    },
    props: ['id'],
    created() {
        axios
            .get(`${this.BACKEND_URL}/personnes/${this.id}`)
            .then(res => this.personne = res.data)
    }
}
</script>
```

Reprenons le `script` du composant `PersonneDetails.vue`

```
<script>
import axios from 'axios';

export default {
    data() {
        return {
            personne: {}
        }
    },
    props: ['id'],
    created() {
        axios
            .get(`${this.BACKEND_URL}/personnes/${this.id}`)
            .then(res => this.personne = res.data)
    }
}
</script>
```

Objectif

Transformer ce composant en **Composition API**.

Vue.js

L'équivalent de `PersonneDetails.vue` en API Composition

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

const personne = ref({})
const { id } = defineProps(['id'])

const { proxy } = getCurrentInstance()

const BACKEND_URL = proxy.BACKEND_URL

onMounted(() => {
    axios
        .get(`${BACKEND_URL}/personnes/${id}`)
        .then(res => personne.value = res.data)
})

</script>
```

Vue.js

Remarque

Le hook `created` de **API Option** est souvent remplacée par `onBeforeMount` ou `onMounted` dans **API Composition**.

Vue.js

Deux solutions possibles pour la validation de formulaires

- Première solution avec `Field`, `Form` et `ErrorMessage` (comme pour les composants **API Option**
- Deuxième solution avec `useField` et `useForm`

Vue.js

Dans la suite

Nous allons

1. créer le composant `PersonShowView` dans `views`,
2. créer le composant `PersonAdd` dans `components`,
3. associer une route `person` pour `PersonShowView`
4. définir `PersonAdd` comme composant enfant de `PersonShowView`,

Vue.js

Commençons par définir `PersonAdd` comme enfant de `PersonneShowView`

```
<template>
    <h1>Gestion de personnes</h1>
    <PersonneAdd />
    <ul>
        <li v-for="(elt, index) in personnes" :key="index">
            <router-link :to="{ name: 'personne-details', params: { id: elt.id } }">
                {{ elt.nom }} {{ elt.prenom }}
            </router-link>
        </li>
    </ul>
</template>

<script setup>
import PersonneAdd from '@/components/PersonAdd.vue';
import { reactive } from 'vue';

var personnes = reactive([
    { id: 1, nom: 'Wick', prenom: 'John', age: 45 },
    { id: 2, nom: 'Dalton', prenom: 'Jack', age: 40 },
    { id: 3, nom: 'Dupont', prenom: 'Sophie', age: 30 }
]);

</script>
```

Vue.js

Dans `PersonneAdd.vue`, commençons par préparer le template

```
<template >
    <h2>Ajouter une nouvelle personne</h2>
    <Form @submit="ajouterPersonne" :validation-schema="simpleSchema">
        <div>
            <label for="nom">Nom</label>
            <Field id="nom" type="text" name="nom" />
            <ErrorMessage name="nom" />
        </div>
        <div>
            <label for="prenom">Prénom</label>
            <Field id="prenom" type="text" name="prenom"/>
            <ErrorMessage name="prenom" />
        </div>
        <div>
            <label for="age">Age</label>
            <Field id="age" type="number" name="age" />
            <ErrorMessage name="age" />
        </div>
        <button>Ajouter</button>
    </Form>
</template>
```

Vue.js

Utilisons ensuite API Composition pour la validation du formulaire et la récupération de données

```
<script setup>

import { Field, Form, ErrorMessage } from 'vee-validate'
import * as yup from 'yup'

const simpleSchema = {
    age:
        yup
            .number()
            .required("L'age est obligatoire")
            .min(18, "L'age min est 18")
            .max(120, "L'age max est 120"),
    nom:
        yup
            .string()
            .required("Ce champ est obligatoire")
            .matches(/^[A-Z]{1}.*/, "Première lettre en majuscule"),
    prenom:
        yup
            .string()
            .required("Ce champ est obligatoire")
            .matches(/^[A-Z]{1}.*/, "Première lettre en majuscule"),
}
const ajouterPersonne = (values) => {
    console.log(values)
}
</script>
```

Vue.js

Une deuxième solution consiste à utiliser `useField` pour associer le validateur au champ et afficher le message d'erreur

```
<template>
    <div>
        <input v-model="value" />
        <span>{{ errorMessage }}</span>
    </div>
</template>

<script setup>
import { useField } from 'vee-validate';

const { value, errorMessage } = useField('age', 'minMax:18,150');
</script>
```

Vue.js

Pour changer le nom du champ (`age` par exemple)

```
<template>
    <div>
        <input v-model="age" />
        <span>{{ errorMessage }}</span>
    </div>
</template>

<script setup>
import { useField } from 'vee-validate';

const { value: age, errorMessage } = useField('age', 'minMax:18,150');
</script>
```

Vue.js

On peut aussi faire

```
<template>
    <div>
        <input v-model="age" />
        <span>{{ ageErr }}</span>
    </div>
</template>

<script setup>
import { useField } from 'vee-validate';

const { value: age, errorMessage: ageErr } = useField('age', 'minMax:18,150');
</script>
```

Vue.js

Commençons par définir le schéma de validation dans `PersonAdd`

```
<script setup>

import { useForm, useField } from 'vee-validate';
import { number, object, string } from 'yup';

const schema = object({
    age: number().required().min(18).max(120),
    nom: string().required().matches(/^[A-Z]{1}.*/),
    prenom: string().required().matches(/^[A-Z]{1}.*/)
});

</script>
```

Vue.js

Utilisons `useForm` pour obtenir un objet `handleSubmit` permettant la gestion du formulaire associé au schéma de validation `schema`

```
<script setup>

import { useForm, useField } from 'vee-validate';
import { number, object, string } from 'yup';

const schema = object({
    age: number().required().min(18).max(120),
    nom: string().required().matches(/^[A-Z]{1}.*/),
    prenom: string().required().matches(/^[A-Z]{1}.*/)
});

const { handleSubmit } = useForm({
    validationSchema: schema,
});

</script>
```

Vue.js

Pour exécuter une fonction dans le cas où la validation du formulaire a échoué

```
<script setup>

import { useForm, useField } from 'vee-validate';
import { number, object, string } from 'yup';

const schema = object({
    age: number().required().min(18).max(120),
    nom: string().required().matches(/^[A-Z]{1}.*/),
    prenom: string().required().matches(/^[A-Z]{1}.*/)
});

const { handleSubmit } = useForm({
    validationSchema: schema,
});

function onInvalidSubmit({ values, errors, results }) {
    console.log(values);
    console.log(errors);
    console.log(results);
}

</script>
```

Vue.js

Pour récupérer le message d'erreur associé à chaque champ

```
<script setup>

import { useForm, useField } from 'vee-validate';
import { number, object, string } from 'yup';

const schema = object({
    age: number().required().min(18).max(120),
    nom: string().required().matches(/^[A-Z]{1}.*/),
    prenom: string().required().matches(/^[A-Z]{1}.*/)
});

const { handleSubmit } = useForm({
    validationSchema: schema,
});

function onInvalidSubmit({ values, errors, results }) {
    console.log(values);
    console.log(errors);
    console.log(results);
}

const { value: age, errorMessage: ageErr } = useField('age');
const { value: nom, errorMessage: nomErr } = useField('nom');
const { value: prenom, errorMessage: prenomErr } = useField('prenom');

</script>
```

Enfin, définissons la méthode qui sera exécuté dans le cas de la soumission d'un formulaire valide

```
<script setup>

import { useForm, useField } from 'vee-validate';
import { number, object, string } from 'yup';

const schema = object({
    age: number().required().min(18).max(120),
    nom: string().required().matches(/^[A-Z]{1}.*/),
    prenom: string().required().matches(/^[A-Z]{1}.*/)
});

const { handleSubmit } = useForm({
    validationSchema: schema,
});

function onInvalidSubmit({ values, errors, results }) {
    console.log(values);
    console.log(errors);
    console.log(results);
}

const { value: age, errorMessage: ageErr } = useField('age');
const { value: nom, errorMessage: nomErr } = useField('nom');
const { value: prenom, errorMessage: prenomErr } = useField('prenom');

const ajouterPersonne = handleSubmit((values) => {
    console.log(values)
}, onInvalidSubmit);

</script>
```

Vue.js

Nous pourrons remplacer les `useField` précédents par un seul `udeFieldModel`

```
<script setup>

import { useForm } from 'vee-validate';
import { number, string } from 'yup';

const schema = {
    age: number().required().min(18).max(120),
    nom: string().required().matches(/^[A-Z]{1}.*/),
    prenom: string().required().matches(/^[A-Z]{1}.*/)
};
const { useFieldModel, errors, handleSubmit } = useForm({
    validationSchema: schema,
});

const [nom, prenom, age] = useFieldModel(['nom', 'prenom', 'age']);


function onInvalidSubmit({ values, errors, results }) {
    console.log(values);
    console.log(errors);
    console.log(results);
}

const ajouterPersonne = handleSubmit((values) => {
    console.log(values)
}, onInvalidSubmit);
</script>
```

Vue.js

Le template aussi change

```
<template>
    <form @submit="ajouterPersonne" >
        <div>
            Nom : <input type=text v-model="nom"><span>{{ errors.nom }}</span>
        </div>
        <div>
            Prénom : <input type=text v-model="prenom"><span>{{ errors.prenom }}</span>
        </div>
        <div>
            Age : <input type=number v-model="age"><span>{{ errors.age }}</span>
        </div>
        <div>
            <button>
                Ajouter
            </button>
        </div>
    </form>
</template>
```

Vue.js

Question

Comment faire si un composant 'grand parent' a besoin de transmettre une donnée à tous ses descendants (composants enfants, 'petits enfants'...) ?

Vue.js

Question

Comment faire si un composant 'grand parent' a besoin de transmettre une donnée à tous ses descendants (composants enfants, 'petits enfants'...) ?

Première solution

Chaque parent transmet les données à ses enfants, les enfants utiliseront `props` pour récupérer les données (qu'ils devront, à leur tour, les transmettre à leurs enfants...)

Vue.js

Question

Comment faire si un composant 'grand parent' a besoin de transmettre une donnée à tous ses descendants (composants enfants, 'petits enfants'...) ?

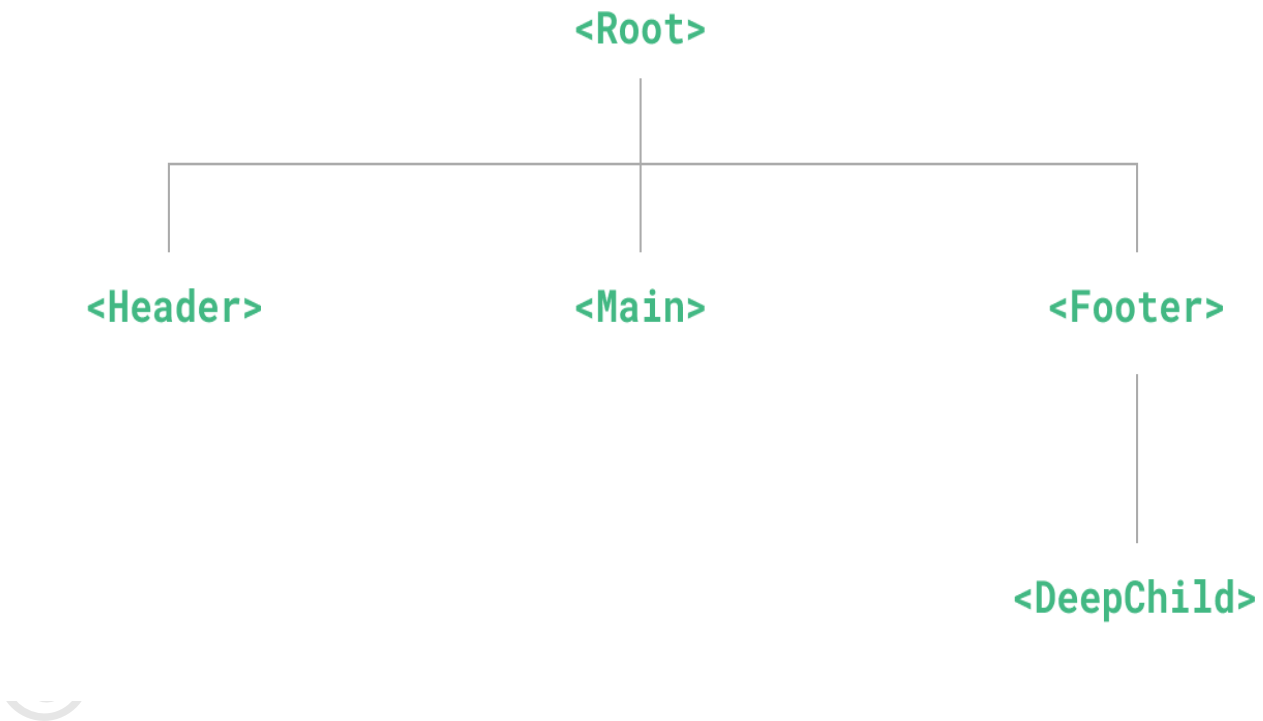
Première solution

Chaque parent transmet les données à ses enfants, les enfants utiliseront `props` pour récupérer les données (qu'ils devront, à leur tour, les transmettre à leurs enfants...)

Deuxième solution

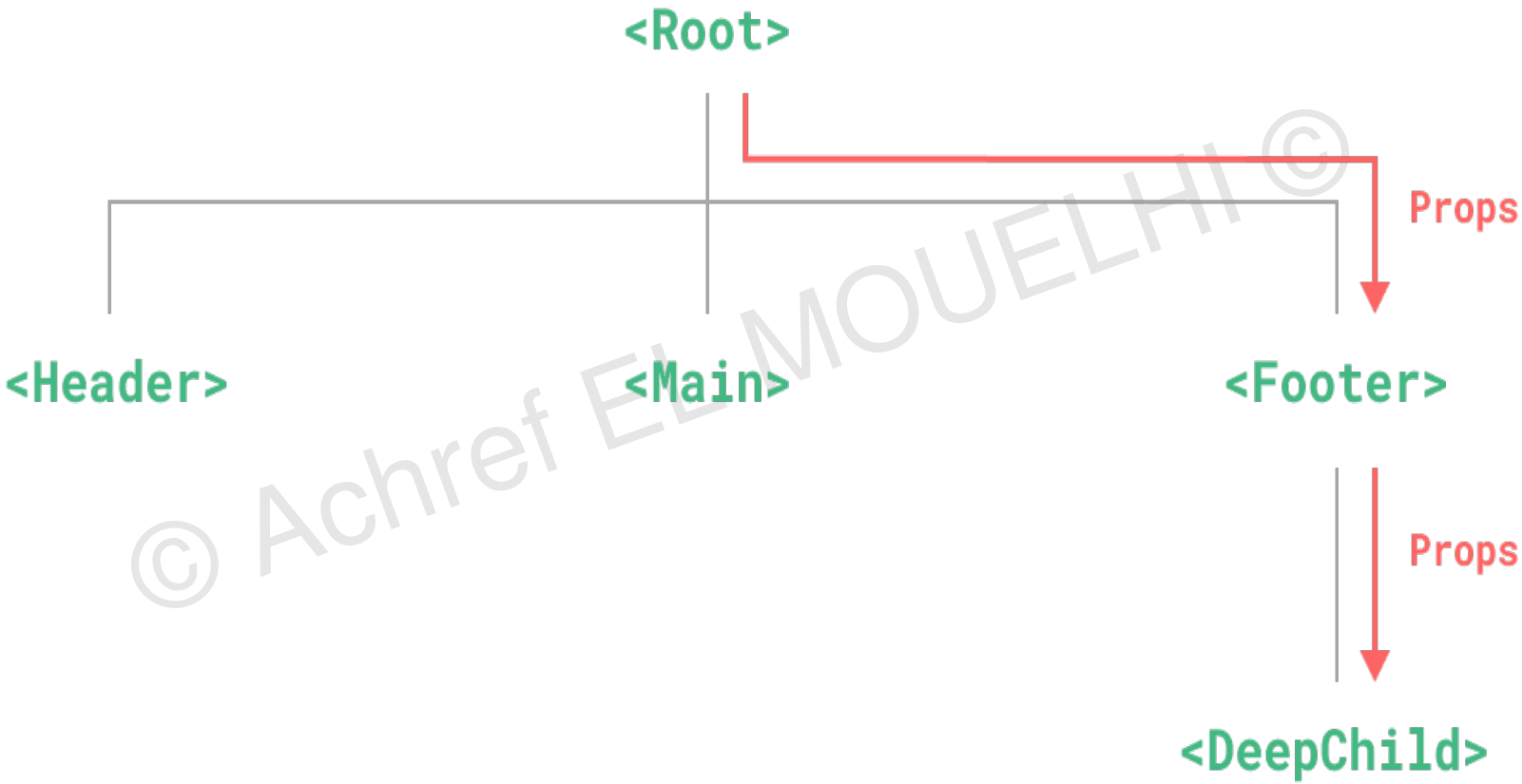
Le parent utilise `provide` pour envoyer les données et les enfants utilisent `inject` pour la récupération.

Vue.js



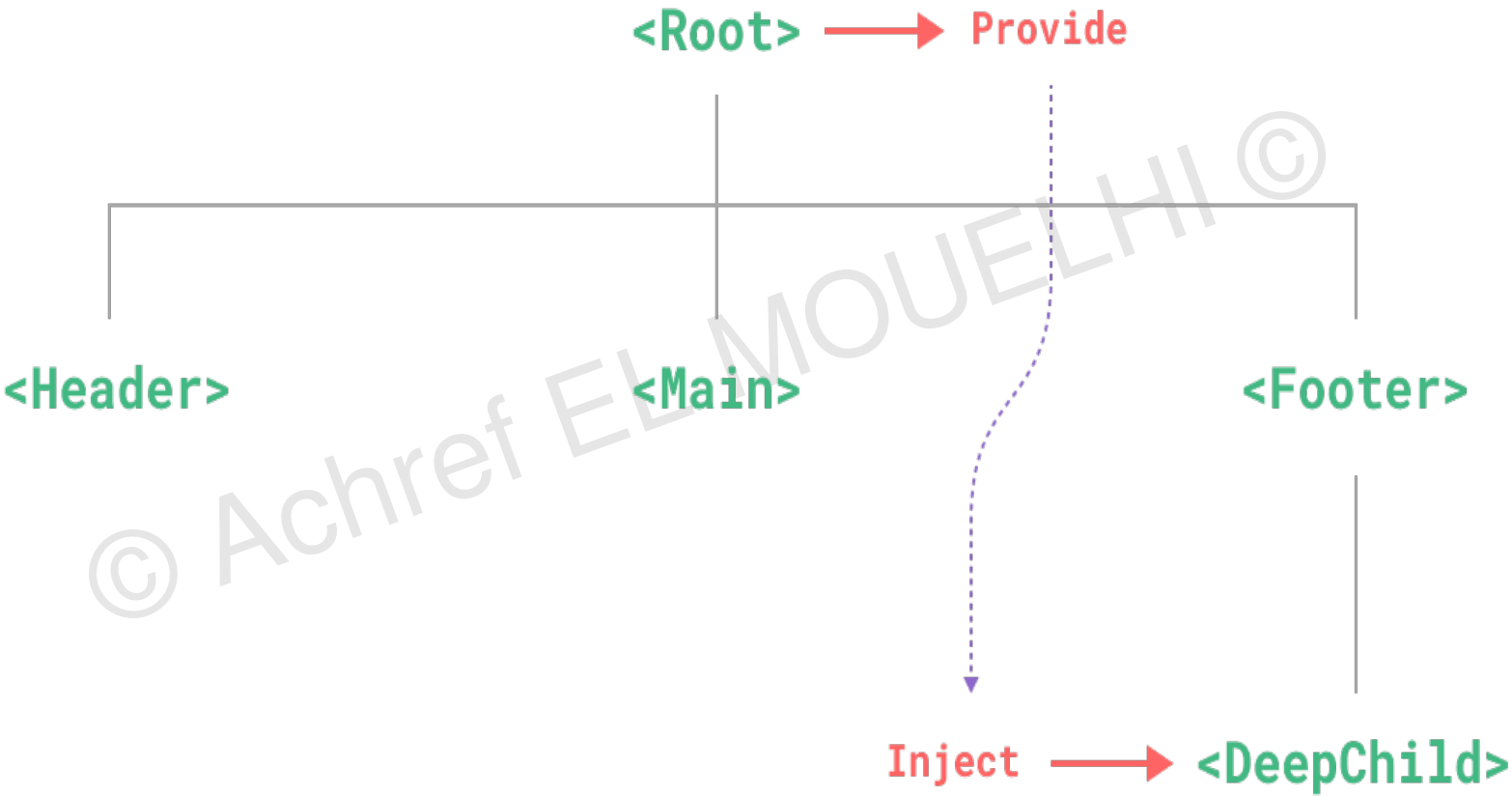
Source : *documentation officielle*

Vue.js



Source : *documentation officielle*

Vue.js



Source : *documentation officielle*

Vue.js

Exemple

- Renommons le composant `Primeur.vue` en `PrimeurView.vue`
- Déplaçons `PrimeurView.vue` dans `views`
- Associons une route `/primeur` à ce composant
- Transformons `PrimeurView.vue` en **API Composition**

Vue.js

Nouveau contenu de `PrimeurView.vue`

```
<template>
    <ProduitComponent
        v-for="(elt, index) in produits"
        :key="index"
        :produit="elt"/>
</template>

<script setup>

import { reactive } from 'vue';
import ProduitComponent from '../components/ProduitComponent.vue'

const produits = reactive([
    { nom: "banane", prix: 3, quantite: 10 },
    { nom: "fraise", prix: 10, quantite: 20 },
    { nom: "poivron", prix: 5, quantite: 10 }
])

</script>
```

Vue.js

Nouveau contenu de `ProduitComponent.vue`

```
<script>

export default {
    props: ['produit']
}

</script>

<template>
    <ul>
        <li> {{ produit.nom }} </li>
        <li>Quantité en stock : {{ produit.quantite }} </li>
        <li>
            Prix : {{ produit.prix }}
        </li>
    </ul>
</template>
```

Vue.js

Pour la suite

- Nous voudrons que le composant `PrimeurView` fournisse la valeur de la TVA à tous ses composants enfants.
- Nous allons créer un composant `PrixComponent` dans `components` qui affichera le prix HT et TTC de chaque produit.
- `PrixComponent` est l'enfant de `ProduitComponent` qui est lui même l'enfant de `PrimeurView`.

Vue.js

Dans `PrimeurView.vue`, fournissons la valeur de la TVA

```
<template>
    <ProduitComponent
        v-for="(elt, index) in produits"
        :key="index"
        :produit="elt"/>
</template>

<script setup>

import { reactive, provide } from 'vue';
import ProduitComponent from '../components/ProduitComponent.vue'

const produits = reactive([
    { nom: "banane", prix: 3, quantite: 10 },
    { nom: "fraise", prix: 10, quantite: 20 },
    { nom: "poivron", prix: 5, quantite: 10 }
])

provide('tva', 0.2)
</script>
```

Vue.js

Dans `PrixComponent.vue`, utilisons `inject` pour récupérer la TVA fournie par `PrimeurView`

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

defineProps({
    prix: Number
})

const tva = inject('tva')
</script>

<template>
    <ul>
        <li>Prix HT : {{ prix }}</li>
        <li>Prix TTC : {{ prix + prix * tva }}</li>
    </ul>
</template>
```

Vue.js

Utilisons le composant `PrixComponent` dans `ProduitComponent`

```
<script>
import PrixComponent from './PrixComponent.vue'
export default {
    props: ['produit'],
    components: {
        PrixComponent
    }
}
</script>

<template>
    <ul>
        <li> {{ produit.nom }} </li>
        <li>Quantité en stock : {{ produit.quantite }} </li>
        <li>
            <PrixComponent :prix="produit.prix" />
        </li>
    </ul>
</template>
```

Vue.js

Dans `PrixComponent.vue`, nous pouvons définir une valeur par défaut dans `inject` qui sera utilisée si le parent ne fournit pas la variable

```
<script setup>

import { defineProps } from 'vue';
import { inject } from 'vue'

defineProps({
    prix: Number
})

const tva = inject('tva', 0.2)
</script>

<template>
    <ul>
        <li>Prix HT : {{ prix }}</li>
        <li>Prix TTC : {{ prix + prix * tva }}</li>
    </ul>
</template>
```

Vue.js

Deuxième application : dans `main.js`, on utilise `provide` pour exporter `axios` et `baseUrl` à tous les composants enfants

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import axios from 'axios'
import VueAxios from 'vue-axios'

const app = createApp(App);
app.config.globalProperties.baseUrl = 'http://localhost:5555';

app
    .use(router)
    .use(VueAxios, axios)

app.provide('axios', app.config.globalProperties.axios)
app.provide('baseUrl', app.config.globalProperties.baseUrl)

app.mount('#app')

import "bootstrap/dist/css/bootstrap.min.css"
import "bootstrap/dist/js/bootstrap.bundle.min.js"
import "@fortawesome/fontawesome-free/css/all.css"
import "bootstrap-icons/font/bootstrap-icons.css"
import "./assets/css/style.css"
import '@/validators/min-max';
```

Vue.js

Dans `script` **de** `PersonneShowView.vue`**, on utilise** `inject` **pour utiliser** `axios` **et** `baseUrl`

```
<script setup>
import PersonneAdd from '@/components/PersonneAdd.vue';
import { ref, onMounted, inject } from 'vue';

const erreur = ref(null)
let personnes = ref([
])

const axios = inject('axios')
const baseUrl = inject('baseUrl')

onMounted(() => {
    axios
        .get(`${baseUrl}/personnes`)
        .then(response => personnes.value = response.data)
        .catch((error) => erreur.value = error)
})

const ajouterDansListe = (values) => {
    personnes.value.push(values)
}

const supprimerPersonne = (id) => {
    axios
        .delete(`${baseUrl}/personnes/${id}`)
        .then(() => personnes.value = personnes.value.filter(elt => elt.id != id))
}
</script>
```

Vue.js

Composants dynamiques : objectif

Basculer entre plusieurs composants sans

- utiliser le routage
- changer de route

Vue.js

Exemple

- Définir trois composants dans `components`
  - `Actors.vue`
  - `Players.vue`
  - `Singers.vue`
- Définir un composant `DynamicView.vue` dans `views` et lui associer une route

Vue.js

Contenu de `Actors.vue`

```
<template>
    <h3>Acteurs</h3>
    <ul>
        <li>Denzel Washington</li>
        <li>Robert De Niro</li>
        <li>Morgan Freeman</li>
    </ul>
</template>

<script>
export default {
    name: 'ActorsComponent'
}
</script>
```

Vue.js

Contenu de `Players.vue`

```
<template>
    <h3>Joueurs</h3>
    <ul>
        <li>Lionel Messi</li>
        <li>Cristiano Ronaldo</li>
        <li>Andrés Iniesta</li>
    </ul>
</template>

<script>
export default {
    name: 'PlayersComponent'
}
</script>
```

Vue.js

Contenu de `Singers.vue`

```
<template>
    <h3>Chanteurs</h3>
    <ul>
        <li>Madonna</li>
        <li>Michael Jackson</li>
        <li>Johnny Hallyday</li>
    </ul>
</template>

<script>
export default {
    name: 'SingersComponent'
}
</script>
```

Vue.js

Considérons le contenu initial de `DynamicView.vue`

```
<template>
    <h1> Composants dynamiques </h1>
    <button>Acteurs</button>
    <button>Joueurs</button>
    <button>Chanteurs</button>
</template>
```

Vue.js

Considérons le contenu initial de `DynamicView.vue`

```
<template>
    <h1> Composants dynamiques </h1>
    <button>Acteurs</button>
    <button>Joueurs</button>
    <button>Chanteurs</button>
</template>
```


Objectif

En cliquant sur un de ces trois boutons, afficher le composant correspondant.

Vue.js

Utilisons `component` et l'attribut `is` pour visualiser le composant demandé

```
<script setup>
import { ref } from "vue";
import ActorsComponent from "../components/Actors.vue";
import PlayersComponent from "../components/Players.vue";
import SingersComponent from "../components/Singers.vue";

const composantCourant = ref(ActorsComponent)

const toggle = (component) => {
    composantCourant.value = component
}
</script>

<template>
    <h1> Composants dynamiques </h1>
    <button @click="toggle(ActorsComponent)">Acteurs</button>
    <button @click="toggle(PlayersComponent)">Joueurs</button>
    <button @click="toggle(SingersComponent)">Chanteurs</button>
    <component :is="composantCourant" />
</template>
```

Exercice

Modifiez le composant `Players.vue` pour permettre à l'utilisateur d'ajouter un nouveau joueur.

Vue.js

Une solution possible

```
<template>
    <h3>Joueurs</h3>
    <ul>
        <li v-for="(elt, indice) in players" :key="indice"> {{ elt }}</li>
    </ul>
    <input type="text" placeholder="Nom d'une légende" v-model="player">
    <button @click="ajouter">Ajouter</button>
</template>

<script>
export default {
    name: 'PlayersComponent',
    data() {
        return {
            players: ['Lionel Messi', 'Cristiano Ronaldo', 'Andrés Iniesta'],
            player: null,
        }
    },
    methods: {
        ajouter() {
            this.players.push(this.player)
            this.player = ''
        }
    }
}
</script>
```

Remarques

- Saisissez une valeur pour ajouter un nouveau joueur sans cliquer sur le bouton puis changez de composant et revenez sur `Players.vue` et vérifiez que la valeur a disparu de la zone de saisie.
- Ajoutez un nouveau joueur et changez de composant et revenez sur `Players.vue` et vérifiez que le joueur n'est plus affiché.

Remarques

- Saisissez une valeur pour ajouter un nouveau joueur sans cliquer sur le bouton puis changez de composant et revenez sur `Players.vue` et vérifiez que la valeur a disparu de la zone de saisie.
- Ajoutez un nouveau joueur et changez de composant et revenez sur `Players.vue` et vérifiez que le joueur n'est plus affiché.

Explication

Le composant sera recréé chaque fois.

Remarques

- Saisissez une valeur pour ajouter un nouveau joueur sans cliquer sur le bouton puis changez de composant et revenez sur `Players.vue` et vérifiez que la valeur a disparu de la zone de saisie.
- Ajoutez un nouveau joueur et changez de composant et revenez sur `Players.vue` et vérifiez que le joueur n'est plus affiché.

Explication

Le composant sera recréé chaque fois.

Solution

Éviter de recréer le composant à chaque visite (le garder en vie).

Pour garder un composant en vie, on utilise la balise `<keep-alive>` (dans `DynamicView.vue`)

```
<script setup>
import { ref } from "vue";
import ActorsComponent from "../components/Actors.vue";
import PlayersComponent from "../components/Players.vue";
import SingersComponent from "../components/Singers.vue";

const composantCourant = ref(ActorsComponent)

const toggle = (component) => {
    composantCourant.value = component
}
</script>
<template>
    <h1> Composants dynamiques </h1>
    <button @click="toggle(ActorsComponent)">Acteurs</button>
    <button @click="toggle(PlayersComponent)">Joueurs</button>
    <button @click="toggle(SingersComponent)">Chanteurs</button>
    <keep-alive>
        <component :is="composantCourant" />
    </keep-alive>
</template>
```