Formation Vue.js

Chapitre 4

Vue.js 3 - Composition API

Ibrahim ALAME

2025

Objectifs

Ce chapitre couvre Vue.js 3 avec la Composition API :

- Comprendre les concepts fondamentaux de Vue.js
- Maîtriser la réactivité (ref, reactive, computed, watch)
- Créer et composer des composants
- Gérer la communication entre composants (props, events, provide/inject)
- Utiliser les directives structurelles (v-if, v-for, v-model)
- Comprendre le cycle de vie des composants
- Effectuer des requêtes HTTP
- Utiliser les composants natifs avancés

Prérequis : Chapitres 1, 2, 3 (HTML/CSS, JavaScript, TypeScript)
Durée estimée : 20 à 25 heures

Table des matières

1 Introduction à Vue.js

1.1 Qu'est-ce que Vue.js ?

Définition

Vue.js est un framework JavaScript progressif pour construire des interfaces utilisateur. Créé par Evan You en 2014, il se distingue par sa simplicité, sa flexibilité et ses excellentes performances.

1.1.1 Caractéristiques principales

- **Progressif** : adoptable progressivement, du simple widget à l'application complète
- **Réactif** : le DOM se met à jour automatiquement quand les données changent
- **Composants** : architecture basée sur des composants réutilisables
- **Performant** : DOM virtuel et compilation optimisée
- **TypeScript** : support natif de TypeScript
- **Écosystème riche** : Vue Router, Pinia, Vue DevTools...

1.1.2 Vue 3 vs Vue 2

Vue 3 apporte des améliorations majeures :

- **Composition API** : nouvelle façon d'organiser la logique des composants
- **Performances** : plus rapide et plus léger
- **TypeScript** : meilleur support natif
- **Teleport** : rendre des éléments ailleurs dans le DOM
- **Fragments** : composants multi-racines
- **Suspense** : gestion du chargement asynchrone

1.2 Les Single Page Applications (SPA)

Définition

Une **SPA** (Single Page Application) est une application web qui charge une seule page HTML et met à jour dynamiquement le contenu sans rechargement complet de la page.

Avantages :

- Expérience utilisateur fluide (pas de rechargement)
- Meilleure réactivité
- Séparation frontend/backend claire

Inconvénients :

- SEO plus complexe (résolu avec SSR)
- Temps de chargement initial plus long
- Nécessite JavaScript activé

1.3 Créer un projet avec create-vue

Listing 1 – Création d'un projet Vue.js

```
# Creer un nouveau projet
npm create vue@latest

# Repondre aux questions :
# - Project name: mon-projet
# - Add TypeScript? Yes
# - Add JSX Support? No
# - Add Vue Router? Yes (pour les SPA multi-pages)
# - Add Pinia? Yes (gestion d'etat)
# - Add Vitest? Yes (tests unitaires)
# - Add ESLint? Yes
# - Add Prettier? Yes

# Installer les dependances
cd mon-projet
npm install

# Lancer le serveur de developpement
npm run dev
```

1.4 Structure d'un projet Vue.js

```
mon-projet/
+-- node_modules/
+-- public/
|   +-- favicon.ico
+-- src/
|   +-- assets/           # Ressources statiques
|   +-- components/       # Composants reutilisables
|   +-- views/            # Pages/vues principales
|   +-- router/           # Configuration du routeur
|   +-- stores/           # Stores Pinia
|   +-- App.vue           # Composant racine
|   +-- main.ts           # Point d'entree
+-- index.html
+-- package.json
+-- tsconfig.json
+-- vite.config.ts
+-- env.d.ts
```

1.5 Anatomie d'un composant Vue (SFC)

Un **Single File Component** (SFC) est un fichier `.vue` qui contient trois sections :

Listing 2 – Structure d'un composant .vue

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

```
// Logique du composant (Composition API)
import { ref } from 'vue'

const message = ref('Bonjour Vue!')
</script>

<template>
  <!-- Structure HTML du composant -->
  <div class="container">
    <h1>{{ message }}</h1>
  </div>
</template>

<style scoped>
/* Styles CSS (scoped = locaux au composant) */
.container {
  padding: 20px;
}

h1 {
  color: #42b883;
}
</style>
```

Vue.js

L'attribut `setup` dans `<script setup>` active la Composition API avec une syntaxe simplifiée. C'est la syntaxe recommandée pour Vue 3.

1.6 Fonctionnement de Vite

Définition

Vite est un outil de build moderne créé par Evan You. Il offre un serveur de développement ultra-rapide grâce aux ES modules natifs et un build optimisé pour la production.

Avantages de Vite :

- Démarrage instantané (pas de bundling en dev)
- Hot Module Replacement (HMR) ultra-rapide
- Build de production optimisé avec Rollup
- Support natif de TypeScript, JSX, CSS

Listing 3 – Commandes Vite

```
# Developpement
npm run dev

# Build production
npm run build
```

```
# Preview du build
npm run preview
```

2 Interagir avec le DOM

2.1 L'interpolation de texte

La syntaxe {{ }} (moustaches) permet d'afficher des données dans le template :

```
<script setup lang="ts">
const nom = 'Alice'
const age = 25
const date = new Date()
</script>

<template>
  <!-- Affichage simple -->
  <p>Bonjour {{ nom }}!</p>

  <!-- Expressions JavaScript -->
  <p>Age: {{ age }} ans</p>
  <p>Majeur: {{ age >= 18 ? 'Oui' : 'Non' }}</p>
  <p>Nom en majuscules: {{ nom.toUpperCase() }}</p>
  <p>Date: {{ date.toLocaleDateString('fr-FR') }}</p>

  <!-- Calculs -->
  <p>Dans 10 ans: {{ age + 10 }} ans</p>
</template>
```

Important

Seules les **expressions** sont autorisées dans les interpolations, pas les instructions. Évitez la logique complexe dans le template.

```
// OK - expressions
{{ ok ? 'Oui' : 'Non' }}
{{ message.split('').reverse().join('') }}

// INTERDIT - instructions
{{ if (ok) { return 'Oui' } }}
{{ let x = 1 }}
```

2.2 Les directives v-text et v-html

```
<script setup lang="ts">
const texte = 'Bonjour <strong>Vue.js</strong>'
const html = '<span style="color: red">Texte rouge</span>'
</script>

<template>
  <!-- v-text : equivalent a {{ }} -->
  <p v-text="texte"></p>
  <!-- Resultat : Bonjour <strong>Vue.js</strong> (echappe) -->
```

```
10
11    <!-- v-html : interprete le HTML -->
12    <p v-html="html"></p>
13    <!-- Resultat : Texte rouge (en rouge) -->
14  </template>
```

Important

Attention aux failles XSS ! N'utilisez `v-html` que sur du contenu de confiance, jamais sur du contenu utilisateur.

2.3 La directive v-bind

`v-bind` lie dynamiquement un attribut HTML à une expression JavaScript :

```
1  <script setup lang="ts">
2  const imageUrl = '/images/logo.png'
3  const altText = 'Logo Vue.js'
4  const lienActif = true
5  const inputId = 'email-input'
6  </script>
7
8  <template>
9    <!-- Syntaxe complete -->
10   <img v-bind:src="imageUrl" v-bind:alt="altText">
11
12   <!-- Syntaxe raccourcie (recommandee) -->
13   <img :src="imageUrl" :alt="altText">
14
15   <!-- Attribut dynamique -->
16   <a :href="lienActif ? '/dashboard' : '#'">Dashboard</a>
17
18   <!-- Plusieurs attributs -->
19   <input :id="inputId" :name="inputId" type="email">
20 </template>
```

2.3.1 Attributs booléens

```
1  <script setup lang="ts">
2  const estDesactive = true
3  const estCoche = false
4  </script>
5
6  <template>
7    <!-- L'attribut est present si true, absent si false -->
8    <button :disabled="estDesactive">Envoyer</button>
9    <!-- Resultat : <button disabled>Envoyer</button> -->
10
11   <input type="checkbox" :checked="estCoche">
12   <!-- Resultat : <input type="checkbox"> (pas de checked) -->
```

```
</template>
```

2.3.2 Liaison d'un objet d'attributs

```
<script setup lang="ts">
const attributsInput = {
  id: 'username',
  type: 'text',
  placeholder: 'Entrez votre nom',
  required: true
}
</script>

<template>
  <!-- Lier tous les attributs d'un objet -->
  <input v-bind="attributsInput">
  <!-- Equivalent a:
       <input id="username" type="text"
              placeholder="Entrez votre nom" required> -->
</template>
```

2.4 Liaison de classes CSS

2.4.1 Syntaxe objet

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

const estActif = ref(true)
const aErreur = ref(false)
</script>

<template>
  <!-- Classe conditionnelle -->
  <div :class="{ active: estActif }">Element</div>
  <!-- Si estActif = true : <div class="active"> -->

  <!-- Plusieurs classes conditionnelles -->
  <div :class="{
    active: estActif,
    'text-danger': aErreur,
    disabled: !estActif
  }">
    Element
  </div>

  <!-- Combinaison avec classe statique -->
  <div class="base" :class="{ active: estActif }">
    Element
```

```
  </div>
  <!-- Resultat : <div class="base active"> -->
</template>
```

2.4.2 Syntaxe tableau

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

const classeActive = ref('active')
const classeErreur = ref('')
</script>

<template>
  <!-- Tableau de classes -->
  <div :class="[classeActive, classeErreur]">Element</div>

  <!-- Avec ternaire -->
  <div :class="[estActif ? 'active' : 'inactive', 'base']">
    Element
  </div>

  <!-- Objet dans tableau -->
  <div :class="['base', { active: estActif }]">Element</div>
</template>
```

2.5 Liaison de styles inline

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

const couleur = ref('red')
const taille = ref(16)

const stylesCalcules = computed(() => ({
  color: couleur.value,
  fontSize: `${taille.value}px`,
  fontWeight: 'bold'
}))
</script>

<template>
  <!-- Objet de styles (camelCase) -->
  <p :style="{ color: couleur, fontSize: taille + 'px' }">
    Texte style
  </p>

  <!-- Ou kebab-case avec quotes -->
  <p :style="{ 'font-size': taille + 'px' }">Texte</p>
```

```
22
23    <!-- Avec computed -->
24    <p :style="stylesCalcules">Texte calcule</p>
25
26    <!-- Tableau de styles (fusion) -->
27    <p :style="[stylesBase, stylesSpecifiques]">Texte</p>
28  </template>
```

2.6 Les événements (v-on)

v-on écoute les événements DOM :

```
1  <script setup lang="ts">
2  import { ref } from 'vue'
3
4  const compteur = ref(0)
5
6  function incrementer() {
7    compteur.value++
8  }
9
10 function gererClick(event: MouseEvent) {
11   console.log('Position:', event.clientX, event.clientY)
12 }
13
14 function saluer(nom: string) {
15   alert(`Bonjour ${nom}!`)
16 }
17 </script>
18
19 <template>
20   <!-- Syntaxe complete -->
21   <button v-on:click="incrementer">+1</button>
22
23   <!-- Syntaxe raccourcie (recommandee) -->
24   <button @click="incrementer">Compteur: {{ compteur }}</button>
25
26   <!-- Expression inline -->
27   <button @click="compteur++">+1 inline</button>
28
29   <!-- Avec argument -->
30   <button @click="saluer('Alice')">Saluer</button>
31
32   <!-- Acces a l'evenement -->
33   <button @click="gererClick">Click</button>
34
35   <!-- $event pour l'evenement natif avec arguments -->
36   <button @click="gererClick($event)">Click</button>
37 </template>
```

2.6.1 Modificateurs d'événements

```vue
<template>
  <!-- .prevent : preventDefault() -->
  <form @submit.prevent="soumettre">
    <button type="submit">Envoyer</button>
  </form>

  <!-- .stop : stopPropagation() -->
  <div @click="parent">
    <button @click.stop="enfant">Ne propage pas</button>
  </div>

  <!-- .once : s'execute une seule fois -->
  <button @click.once="executerUneFois">Une fois</button>

  <!-- .self : seulement si l'evenement vient de l'element lui-
     meme -->
  <div @click.self="divClick">
    <button>Le click ici ne declenche pas divClick</button>
  </div>

  <!-- Combinaison de modificateurs -->
  <form @submit.prevent.stop="soumettre"></form>

  <!-- .capture : mode capture -->
  <div @click.capture="handler">...</div>

  <!-- .passive : pour le scroll (performance) -->
  <div @scroll.passive="onScroll">...</div>
</template>
```

2.6.2 Modificateurs de touches

```vue
<template>
  <!-- Touches specifiques -->
  <input @keyup.enter="soumettre">
  <input @keyup.esc="annuler">
  <input @keyup.tab="suivant">
  <input @keyup.delete="supprimer">
  <input @keyup.space="espace">

  <!-- Fleches -->
  <div @keyup.up="haut" @keyup.down="bas"
       @keyup.left="gauche" @keyup.right="droite">
  </div>

  <!-- Modificateurs systeme -->
  <button @click.ctrl="ctrlClick">Ctrl+Click</button>
  <button @click.alt="altClick">Alt+Click</button>
  <button @click.shift="shiftClick">Shift+Click</button>
```

```
18    <button @click.meta="metaClick">Cmd/Win+Click</button>
19
20    <!-- Combinaisons -->
21    <input @keyup.ctrl.enter="ctrlEnter">
22    <button @click.ctrl.shift="ctrlShiftClick">Ctrl+Shift+Click</
         button>
23
24    <!-- .exact : exactement ces touches -->
25    <button @click.ctrl.exact="ctrlSeul">Ctrl seul</button>
26  </template>
```

2.6.3 Modificateurs de souris

```
1  <template>
2    <button @click.left="clickGauche">Clic gauche</button>
3    <button @click.right="clickDroit">Clic droit</button>
4    <button @click.middle="clickMilieu">Clic molette</button>
5  </template>
```

3 La réactivité

Définition

La **réactivité** est le mécanisme qui permet à Vue de détecter les changements de données et de mettre à jour automatiquement le DOM en conséquence.

3.1 La fonction ref()

`ref()` crée une référence réactive pour les valeurs primitives :

Listing 4 – Utilisation de ref()

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

// Creer des refs reactives
const compteur = ref(0)
const message = ref('Bonjour')
const estVisible = ref(true)

// Acceder/modifier avec .value dans le script
function incrementer() {
  compteur.value++
  console.log(compteur.value)
}

function changerMessage() {
  message.value = 'Au revoir'
}
</script>

<template>
  <!-- Dans le template, .value est automatique -->
  <p>Compteur: {{ compteur }}</p>
  <p>Message: {{ message }}</p>

  <button @click="incrementer">+1</button>
  <button @click="changerMessage">Changer</button>
</template>
```

3.1.1 Typage avec TypeScript

```
import { ref, type Ref } from 'vue'

// Type infere automatiquement
const nombre = ref(0)            // Ref<number>
const texte = ref('hello')       // Ref<string>

// Type explicite
const id = ref<string | null>(null)
```

```
const liste = ref<string[]>([])

// Type complexe
interface Utilisateur {
  id: number
  nom: string
  email: string
}

const utilisateur = ref<Utilisateur | null>(null)

// Acces type-safe
utilisateur.value = {
  id: 1,
  nom: 'Alice',
  email: 'alice@example.com'
}
```

3.2 La fonction reactive()

`reactive()` crée un objet réactif. Idéal pour les objets et tableaux :

Listing 5 – Utilisation de reactive()

```
<script setup lang="ts">
import { reactive } from 'vue'

// Objet reactif
const etat = reactive({
  compteur: 0,
  message: 'Bonjour',
  utilisateur: {
    nom: 'Alice',
    age: 25
  }
})

// Tableau reactif
const taches = reactive([
  { id: 1, texte: 'Tache 1', fait: false },
  { id: 2, texte: 'Tache 2', fait: true }
])

// Pas besoin de .value !
function incrementer() {
  etat.compteur++
}

function ajouterTache() {
  taches.push({
    id: taches.length + 1,
    texte: 'Nouvelle tache',
```

```
      fait: false
    })
  }
  </script>

  <template>
    <p>Compteur: {{ etat.compteur }}</p>
    <p>Utilisateur: {{ etat.utilisateur.nom }}</p>

    <ul>
      <li v-for="tache in taches" :key="tache.id">
        {{ tache.texte }}
      </li>
    </ul>
  </template>
```

Important

Limitations de reactive() :

- Ne fonctionne qu'avec les objets (pas les primitifs)
- Ne peut pas être réassigné (perdrait la réactivité)
- La déstructuration perd la réactivité

```
const etat = reactive({ count: 0 })

// MAUVAIS : perd la reactivite
let { count } = etat
count++  // Ne met pas a jour le template

// BON : utiliser toRefs
const { count } = toRefs(etat)
count.value++  // Reactif!
```

3.3 ref() vs reactive()

Caractéristique	ref()	reactive()
Types supportés	Tous	Objets seulement
Accès dans script	`.value`	Direct
Accès dans template	Automatique	Direct
Réassignation	Oui	Non
Déstructuration	Garde réactivité	Perd réactivité

TABLE 1 – Comparaison ref() vs reactive()

À retenir

Recommandation : Utilisez `ref()` par défaut. C'est plus cohérent et évite les pièges de `reactive()`.

3.4 La fonction computed()

`computed()` crée une valeur calculée qui se met à jour automatiquement :

Listing 6 – Propriétés calculées

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

const prenom = ref('Alice')
const nom = ref('Dupont')
const articles = ref([
  { nom: 'Article 1', prix: 10, quantite: 2 },
  { nom: 'Article 2', prix: 25, quantite: 1 },
  { nom: 'Article 3', prix: 15, quantite: 3 }
])

// Computed en lecture seule
const nomComplet = computed(() => {
  return `${prenom.value} ${nom.value}`
})

const total = computed(() => {
  return articles.value.reduce((sum, article) => {
    return sum + article.prix * article.quantite
  }, 0)
})

const nombreArticles = computed(() => articles.value.length)

// Computed avec getter et setter
const nomCompletEditable = computed({
  get() {
    return `${prenom.value} ${nom.value}`
  },
  set(nouveauNom: string) {
    const [p, n] = nouveauNom.split(' ')
    prenom.value = p
    nom.value = n || ''
  }
})
</script>

<template>
  <p>Nom complet: {{ nomComplet }}</p>
  <p>Total: {{ total }} EUR</p>
  <p>{{ nombreArticles }} articles</p>
```

```
42
43    <!-- Computed avec setter -->
44    <input v-model="nomCompletEditable">
45  </template>
```

Vue.js

computed() vs méthode :

— **computed** : mis en cache, recalculé seulement si les dépendances changent

— **méthode** : exécutée à chaque rendu

Utilisez `computed()` pour les valeurs dérivées, les méthodes pour les actions.

3.5 La directive v-model

`v-model` crée une liaison bidirectionnelle entre un champ de formulaire et une donnée :

Listing 7 – v-model sur différents champs

```
1  <script setup lang="ts">
2  import { ref } from 'vue'
3
4  const texte = ref('')
5  const nombre = ref(0)
6  const coche = ref(false)
7  const selection = ref('option1')
8  const selections = ref<string[]>([])
9  const choix = ref('')
10 </script>
11
12 <template>
13   <!-- Input texte -->
14   <input v-model="texte" type="text">
15   <p>Texte: {{ texte }}</p>
16
17   <!-- Input nombre -->
18   <input v-model.number="nombre" type="number">
19
20   <!-- Checkbox simple -->
21   <input v-model="coche" type="checkbox" id="cb">
22   <label for="cb">Accepter</label>
23
24   <!-- Checkboxes multiples -->
25   <input v-model="selections" type="checkbox" value="a"> A
26   <input v-model="selections" type="checkbox" value="b"> B
27   <input v-model="selections" type="checkbox" value="c"> C
28   <!-- selections = ['a', 'c'] si A et C coches -->
29
30   <!-- Radio buttons -->
31   <input v-model="choix" type="radio" value="opt1"> Option 1
32   <input v-model="choix" type="radio" value="opt2"> Option 2
```

```
33
34    <!-- Select -->
35    <select v-model="selection">
36      <option value="option1">Option 1</option>
37      <option value="option2">Option 2</option>
38      <option value="option3">Option 3</option>
39    </select>
40
41    <!-- Select multiple -->
42    <select v-model="selections" multiple>
43      <option value="a">A</option>
44      <option value="b">B</option>
45      <option value="c">C</option>
46    </select>
47
48    <!-- Textarea -->
49    <textarea v-model="texte"></textarea>
50  </template>
```

3.5.1 Modificateurs de v-model

```
1  <template>
2    <!-- .lazy : synchronise sur 'change' au lieu de 'input' -->
3    <input v-model.lazy="texte">
4
5    <!-- .number : convertit en nombre -->
6    <input v-model.number="age" type="number">
7
8    <!-- .trim : supprime les espaces -->
9    <input v-model.trim="nom">
10
11   <!-- Combinaison -->
12   <input v-model.lazy.trim="email">
13 </template>
```

3.6 Les watchers : watch() et watchEffect()

3.6.1 watch()

`watch()` observe des sources réactives spécifiques :

Listing 8 – Utilisation de watch()

```
1  <script setup lang="ts">
2  import { ref, watch } from 'vue'
3
4  const question = ref('')
5  const reponse = ref('')
6  const compteur = ref(0)
7
8  // Observer une ref
```

```
watch(question, (nouvelleValeur, ancienneValeur) => {
  console.log(`Question changee de "${ancienneValeur}" a "${
     nouvelleValeur}"`)

  if (nouvelleValeur.includes('?')) {
    reponse.value = 'Je reflechis...'
    // Appel API par exemple
  }
})

// Observer plusieurs sources
watch([question, compteur], ([newQ, newC], [oldQ, oldC]) => {
  console.log('Question ou compteur a change')
})

// Observer une propriete d'un objet reactif (getter)
const utilisateur = ref({ nom: 'Alice', age: 25 })

watch(
  () => utilisateur.value.age,
  (nouvelAge) => {
    console.log(`Nouvel age: ${nouvelAge}`)
  }
)

// Options
watch(
  question,
  (newVal) => { /* ... */ },
  {
    immediate: true,  // Execute immediatement
    deep: true,       // Observe en profondeur
    once: true        // Une seule fois
  }
)
</script>
```

3.6.2 watchEffect()

`watchEffect()` observe automatiquement toutes les dépendances utilisées :

Listing 9 – Utilisation de watchEffect()

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

const compteur = ref(0)
const message = ref('Bonjour')

// S'execute immediatement et re-execute quand les
// dependances changent
watchEffect(() => {
```

```
  console.log(`Compteur: ${compteur.value}`)
  console.log(`Message: ${message.value}`)
})

// Avec cleanup (nettoyage)
watchEffect((onCleanup) => {
  const timer = setInterval(() => {
    compteur.value++
  }, 1000)

  // Execute avant chaque re-execution et a la destruction
  onCleanup(() => {
    clearInterval(timer)
  })
})

// Arreter un watcher
const stop = watchEffect(() => { /* ... */ })
// Plus tard...
stop()  // Arrete l'observation
</script>
```

À retenir

watch() vs watchEffect() :

- **watch()** : sources explicites, accès à l'ancienne valeur, lazy par défaut
- **watchEffect()** : dépendances automatiques, immédiat, plus simple

Utilisez `watch()` quand vous avez besoin de l'ancienne valeur ou d'un contrôle précis.

4 Le style et les classes

4.1 Portée des styles (scoped)

Listing 10 – Styles scopés

```
<template>
  <div class="container">
    <h1>Titre</h1>
    <p>Paragraphe</p>
  </div>
</template>

<style scoped>
/* Ces styles s'appliquent UNIQUEMENT a ce composant */
.container {
  padding: 20px;
  background: #f5f5f5;
}

h1 {
  color: #42b883;
}

p {
  font-size: 16px;
}
</style>

<!--
Vue genere un attribut unique, ex: data-v-7ba5bd90
Le CSS devient:
.container[data-v-7ba5bd90] { ... }
h1[data-v-7ba5bd90] { ... }
-->
```

4.1.1 Cibler les composants enfants

```
<style scoped>
/* :deep() pour cibler les enfants */
.container :deep(.child-class) {
  color: red;
}

/* :slotted() pour cibler le contenu des slots */
:slotted(p) {
  font-weight: bold;
}

/* :global() pour des regles globales */
```

```
:global(.classe-globale) {
  margin: 0;
}
</style>
```

4.2 Utilisation de Sass/SCSS

```
npm install -D sass
```

Listing 11 – SCSS dans Vue

```
<style lang="scss" scoped>
$primary-color: #42b883;
$secondary-color: #35495e;

.container {
  padding: 20px;

  h1 {
    color: $primary-color;

    &:hover {
      color: darken($primary-color, 10%);
    }
  }

  .card {
    background: white;
    border-radius: 8px;

    &--active {
      border: 2px solid $primary-color;
    }

    &__title {
      font-size: 1.2rem;
    }

    &__content {
      padding: 1rem;
    }
  }
}

// Mixins
@mixin flex-center {
  display: flex;
  justify-content: center;
  align-items: center;
}

```

```
.centered {
  @include flex-center;
  height: 100vh;
}
</style>
```

4.3 CSS Modules

Listing 12 – CSS Modules

```
<template>
  <div :class="$style.container">
    <p :class="[$style.text, $style.bold]">Texte</p>
  </div>
</template>

<style module>
.container {
  padding: 20px;
}

.text {
  color: #333;
}

.bold {
  font-weight: bold;
}
</style>

<!-- Les classes sont renommees automatiquement
     pour eviter les conflits -->
```

5 Les directives structurelles

5.1 Affichage conditionnel : v-if, v-else-if, v-else

Listing 13 – Directives conditionnelles

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

const estConnecte = ref(false)
const role = ref('utilisateur')
const score = ref(75)
</script>

<template>
  <!-- v-if simple -->
  <p v-if="estConnecte">Bienvenue!</p>

  <!-- v-if / v-else -->
  <div v-if="estConnecte">
    <p>Contenu pour utilisateur connecte</p>
  </div>
  <div v-else>
    <p>Veuillez vous connecter</p>
  </div>

  <!-- v-if / v-else-if / v-else -->
  <p v-if="score >= 90">Excellent!</p>
  <p v-else-if="score >= 70">Bien</p>
  <p v-else-if="score >= 50">Passable</p>
  <p v-else>Insuffisant</p>

  <!-- Avec role -->
  <nav v-if="role === 'admin'">
    <a href="/admin">Administration</a>
  </nav>
  <nav v-else-if="role === 'moderateur'">
    <a href="/moderation">Moderation</a>
  </nav>
  <nav v-else>
    <a href="/profil">Mon profil</a>
  </nav>
</template>
```

5.1.1 v-if sur template

```
<template>
  <!-- Grouper plusieurs elements sans wrapper -->
  <template v-if="estConnecte">
    <h1>Bienvenue</h1>
```

```
    <p>Vous etes connecte</p>
    <button @click="deconnexion">Deconnexion</button>
  </template>
</template>
```

5.2 v-show vs v-if

```
<script setup lang="ts">
import { ref } from 'vue'
const visible = ref(true)
</script>

<template>
  <!-- v-if : ajoute/supprime l'element du DOM -->
  <p v-if="visible">Visible avec v-if</p>

  <!-- v-show : toggle display: none -->
  <p v-show="visible">Visible avec v-show</p>
</template>
```

Caractéristique	v-if	v-show
Manipulation DOM	Ajout/suppression	CSS display
Coût initial	Faible si false	Toujours rendu
Coût de toggle	Élevé	Faible
Supporte template	Oui	Non
Supporte v-else	Oui	Non
Utilisation	Toggle rare	Toggle fréquent

TABLE 2 – Comparaison v-if vs v-show

5.3 Boucles avec v-for

Listing 14 – Directive v-for

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

const fruits = ref(['Pomme', 'Banane', 'Orange'])

const utilisateurs = ref([
  { id: 1, nom: 'Alice', age: 25 },
  { id: 2, nom: 'Bob', age: 30 },
  { id: 3, nom: 'Charlie', age: 35 }
])

const objet = ref({
  titre: 'Mon objet',
  auteur: 'Alice',
```

```
    date: '2025-01-15'
})
</script>

<template>
  <!-- Iteration sur un tableau -->
  <ul>
    <li v-for="fruit in fruits" :key="fruit">
      {{ fruit }}
    </li>
  </ul>

  <!-- Avec index -->
  <ul>
    <li v-for="(fruit, index) in fruits" :key="index">
      {{ index + 1 }}. {{ fruit }}
    </li>
  </ul>

  <!-- Tableau d'objets (TOUJOURS utiliser :key avec id unique)
    -->
  <div v-for="user in utilisateurs" :key="user.id" class="card">
    <h3>{{ user.nom }}</h3>
    <p>Age: {{ user.age }} ans</p>
  </div>

  <!-- Iteration sur un objet -->
  <ul>
    <li v-for="(valeur, cle) in objet" :key="cle">
      {{ cle }}: {{ valeur }}
    </li>
  </ul>

  <!-- Avec index aussi -->
  <li v-for="(valeur, cle, index) in objet" :key="cle">
    {{ index }}. {{ cle }}: {{ valeur }}
  </li>

  <!-- Iteration sur un nombre -->
  <span v-for="n in 5" :key="n">{{ n }} </span>
  <!-- 1 2 3 4 5 -->
</template>
```

Important

L'attribut :key est obligatoire ! Il permet à Vue d'identifier chaque élément de manière unique pour optimiser les mises à jour du DOM.

— Utilisez un ID unique (`user.id`) plutôt que l'index

— L'index comme clé peut causer des bugs lors de réordonnancement

5.3.1 v-for sur template

```
<template>
  <!-- Rendre plusieurs elements sans wrapper -->
  <template v-for="item in items" :key="item.id">
    <h3>{{ item.titre }}</h3>
    <p>{{ item.description }}</p>
    <hr>
  </template>
</template>
```

5.3.2 v-for avec v-if

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

const taches = ref([
  { id: 1, texte: 'Tache 1', fait: false },
  { id: 2, texte: 'Tache 2', fait: true },
  { id: 3, texte: 'Tache 3', fait: false }
])

// BONNE PRATIQUE : filtrer avec computed
const tachesActives = computed(() =>
  taches.value.filter(t => !t.fait)
)
</script>

<template>
  <!-- MAUVAIS : v-if et v-for sur le meme element -->
  <!-- v-if a priorite, tache n'existe pas encore! -->
  <!--
  <li v-for="tache in taches" v-if="!tache.fait" :key="tache.id">
    {{ tache.texte }}
  </li>
  -->

  <!-- BON : utiliser computed -->
  <li v-for="tache in tachesActives" :key="tache.id">
    {{ tache.texte }}
  </li>

  <!-- OU : v-if sur un parent/template -->
  <template v-for="tache in taches" :key="tache.id">
    <li v-if="!tache.fait">
      {{ tache.texte }}
    </li>
  </template>
</template>
```

5.4 Les directives v-once et v-memo

Listing 15 – v-once et v-memo

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

const compteur = ref(0)
const items = ref([1, 2, 3, 4, 5])
</script>

<template>
  <!-- v-once : rendu une seule fois, jamais re-rendu -->
  <h1 v-once>Titre statique: {{ compteur }}</h1>
  <!-- Meme si compteur change, le titre reste le meme -->

  <p>Compteur dynamique: {{ compteur }}</p>
  <button @click="compteur++">+1</button>

  <!-- v-memo : re-render seulement si les dependances changent
     -->
  <div v-for="item in items" :key="item" v-memo="[item]">
    <p>Item: {{ item }}</p>
    <p>Compteur: {{ compteur }}</p>
    <!-- Cette div n'est re-rendue que si 'item' change,
         pas quand 'compteur' change -->
  </div>
</template>
```

5.5 La directive v-pre

```
<template>
  <!-- v-pre : affiche le contenu sans compilation -->
  <p v-pre>{{ ceci ne sera pas interprete }}</p>
  <!-- Affiche litteralement: {{ ceci ne sera pas interprete }}
     -->

  <!-- Utile pour afficher de la syntaxe Vue dans la
     documentation -->
</template>
```

6 Les composants

6.1 Introduction aux composants

Définition

Un **composant** Vue est une instance réutilisable avec son propre template, sa logique et son style. Les composants permettent de découper l'interface en morceaux indépendants.

Listing 16 – Composant simple : BoutonCompteur.vue

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

const compteur = ref(0)
</script>

<template>
  <button @click="compteur++">
    Clique: {{ compteur }}
  </button>
</template>

<style scoped>
button {
  padding: 10px 20px;
  font-size: 16px;
  cursor: pointer;
}
</style>
```

6.1.1 Utiliser un composant

Listing 17 – App.vue

```vue
<script setup lang="ts">
// Import du composant
import BoutonCompteur from './components/BoutonCompteur.vue'
</script>

<template>
  <h1>Mon application</h1>

  <!-- Utilisation du composant -->
  <BoutonCompteur />

  <!-- Plusieurs instances independantes -->
  <BoutonCompteur />
  <BoutonCompteur />
</template>
```

6.2 Les props

Les **props** permettent de passer des données du parent vers l'enfant :

Listing 18 – Composant avec props : CarteUtilisateur.vue

```vue
<script setup lang="ts">
// Definition des props avec defineProps
interface Props {
  nom: string
  age: number
  email?: string  // Optionnel
  actif?: boolean
}

const props = defineProps<Props>()

// Ou avec valeurs par defaut
const props2 = withDefaults(defineProps<Props>(), {
  actif: true,
  email: 'non renseigne'
})
</script>

<template>
  <div class="carte" :class="{ inactive: !actif }">
    <h3>{{ nom }}</h3>
    <p>Age: {{ age }} ans</p>
    <p v-if="email">Email: {{ email }}</p>
    <span v-if="actif" class="badge">Actif</span>
  </div>
</template>
```

Listing 19 – Utilisation des props

```vue
<script setup lang="ts">
import CarteUtilisateur from './CarteUtilisateur.vue'
import { ref } from 'vue'

const utilisateurs = ref([
  { id: 1, nom: 'Alice', age: 25, email: 'alice@example.com' },
  { id: 2, nom: 'Bob', age: 30 }
])
</script>

<template>
  <!-- Props statiques -->
  <CarteUtilisateur nom="Charlie" :age="35" />

  <!-- Props dynamiques avec v-bind -->
  <CarteUtilisateur
    :nom="utilisateurs[0].nom"
    :age="utilisateurs[0].age"
```

```
      :email="utilisateurs[0].email"
    />

    <!-- Passer un objet de props -->
    <CarteUtilisateur v-bind="utilisateurs[1]" />

    <!-- Dans une boucle -->
    <CarteUtilisateur
      v-for="user in utilisateurs"
      :key="user.id"
      v-bind="user"
    />
  </template>
```

6.2.1 Validation des props (runtime)

Listing 20 – Validation avec PropType

```
  <script setup lang="ts">
  import { type PropType } from 'vue'

  interface Utilisateur {
    id: number
    nom: string
  }

  // Validation runtime (utile pour composants de librairie)
  const props = defineProps({
    // Type simple
    titre: String,

    // Type requis
    id: {
      type: Number,
      required: true
    },

    // Avec valeur par defaut
    taille: {
      type: String,
      default: 'medium'
    },

    // Validateur personnalise
    status: {
      type: String,
      validator: (value: string) => {
        return ['pending', 'active', 'closed'].includes(value)
      }
    },

```

```
  // Type complexe avec PropType
  utilisateur: {
    type: Object as PropType<Utilisateur>,
    required: true
  },

  // Tableau
  tags: {
    type: Array as PropType<string[]>,
    default: () => []
  }
})
</script>
```

6.3 Les événements (emits)

Les **emits** permettent à l'enfant de communiquer avec le parent :

Listing 21 – Composant émetteur : BoutonAction.vue

```
<script setup lang="ts">
// Definition des evenements avec TypeScript
const emit = defineEmits<{
  (e: 'click'): void
  (e: 'submit', data: { nom: string; email: string }): void
  (e: 'update:modelValue', value: string): void
}>()

function gererClick() {
  emit('click')
}

function soumettre() {
  emit('submit', { nom: 'Alice', email: 'alice@example.com' })
}
</script>

<template>
  <button @click="gererClick">Action</button>
  <button @click="soumettre">Soumettre</button>
</template>
```

Listing 22 – Écouter les événements

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

function onBoutonClick() {
  console.log('Bouton clique!')
}

function onSubmit(data: { nom: string; email: string }) {
```

```
  console.log('Donnees soumises:', data)
}
</script>

<template>
  <BoutonAction
    @click="onBoutonClick"
    @submit="onSubmit"
  />

  <!-- Avec modificateurs -->
  <BoutonAction @click.once="onBoutonClick" />
</template>
```

6.4 Le cycle de vie d'un composant

Listing 23 – Hooks du cycle de vie

```
<script setup lang="ts">
import {
  onBeforeMount,
  onMounted,
  onBeforeUpdate,
  onUpdated,
  onBeforeUnmount,
  onUnmounted
} from 'vue'

// Avant le montage dans le DOM
onBeforeMount(() => {
  console.log('Composant va etre monte')
})

// Apres le montage (DOM disponible)
onMounted(() => {
  console.log('Composant monte')
  // Acces au DOM, appels API, timers...
})

// Avant une mise a jour
onBeforeUpdate(() => {
  console.log('Composant va etre mis a jour')
})

// Apres une mise a jour
onUpdated(() => {
  console.log('Composant mis a jour')
})

// Avant la destruction
onBeforeUnmount(() => {
```

```
  console.log('Composant va etre demonte')
})

// Apres la destruction (nettoyage)
onUnmounted(() => {
  console.log('Composant demonte')
  // Nettoyage: timers, listeners, subscriptions...
})
</script>
```

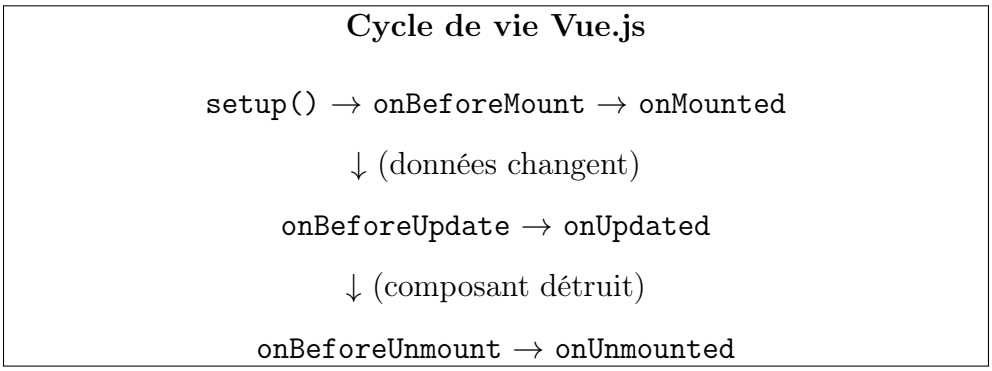

FIGURE 1 – Ordre des hooks du cycle de vie

Exemple

Cas d'utilisation courants :

```
onMounted(() => {
  // Charger des donnees initiales
  fetchData()

  // Ajouter un event listener global
  window.addEventListener('resize', onResize)
})

onUnmounted(() => {
  // Nettoyer les listeners
  window.removeEventListener('resize', onResize)

  // Annuler les timers
  clearInterval(timer)
})
```

6.5 Les références de template (refs)

Listing 24 – Références DOM

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

```

```
// Ref pour element DOM
const inputRef = ref<HTMLInputElement | null>(null)
const divRef = ref<HTMLDivElement | null>(null)

onMounted(() => {
  // Acces a l'element DOM apres le montage
  inputRef.value?.focus()
  console.log(divRef.value?.offsetHeight)
})

function focusInput() {
  inputRef.value?.focus()
}
</script>

<template>
  <input ref="inputRef" type="text">
  <div ref="divRef">Contenu</div>

  <button @click="focusInput">Focus</button>
</template>
```

6.5.1 Refs sur composants enfants

Listing 25 – Ref sur composant enfant

```
<!-- ComposantEnfant.vue -->
<script setup lang="ts">
import { ref } from 'vue'

const compteur = ref(0)

function incrementer() {
  compteur.value++
}

// Exposer des methodes/proprietes au parent
defineExpose({
  compteur,
  incrementer
})
</script>
```

Listing 26 – Accès depuis le parent

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

const enfantRef = ref<InstanceType<typeof ComposantEnfant> | null
  >(null)
```

```
6
7  onMounted(() => {
8    console.log(enfantRef.value?.compteur)
9    enfantRef.value?.incrementer()
10 })
11 </script>
12
13 <template>
14   <ComposantEnfant ref="enfantRef" />
15 </template>
```

7 Fonctionnalités avancées des composants

7.1 v-model sur les composants

Listing 27 – Composant avec v-model : InputPerso.vue

```vue
<script setup lang="ts">
const props = defineProps<{
  modelValue: string
}>()

const emit = defineEmits<{
  (e: 'update:modelValue', value: string): void
}>()

function onInput(event: Event) {
  const target = event.target as HTMLInputElement
  emit('update:modelValue', target.value)
}
</script>

<template>
  <input
    :value="modelValue"
    @input="onInput"
    class="input-perso"
  >
</template>
```

Listing 28 – Utilisation de v-model sur composant

```vue
<script setup lang="ts">
import { ref } from 'vue'
import InputPerso from './InputPerso.vue'

const texte = ref('')
</script>

<template>
  <!-- Ces deux syntaxes sont equivalentes -->
  <InputPerso v-model="texte" />

  <InputPerso
    :modelValue="texte"
    @update:modelValue="texte = $event"
  />
</template>
```

7.1.1 v-model multiples

Listing 29 – v-model multiples

```vue
<!-- FormulaireNom.vue -->
<script setup lang="ts">
defineProps<{
  prenom: string
  nom: string
}>()

const emit = defineEmits<{
  (e: 'update:prenom', value: string): void
  (e: 'update:nom', value: string): void
}>()
</script>

<template>
  <input
    :value="prenom"
    @input="emit('update:prenom', ($event.target as
       HTMLInputElement).value)"
    placeholder="Prenom"
  >
  <input
    :value="nom"
    @input="emit('update:nom', ($event.target as HTMLInputElement
       ).value)"
    placeholder="Nom"
  >
</template>
```

```vue
<script setup lang="ts">
import { ref } from 'vue'
import FormulaireNom from './FormulaireNom.vue'

const prenom = ref('')
const nom = ref('')
</script>

<template>
  <FormulaireNom v-model:prenom="prenom" v-model:nom="nom" />
  <p>{{ prenom }} {{ nom }}</p>
</template>
```

7.2 Les slots

Les **slots** permettent de passer du contenu template à un composant :

Listing 30 – Slot par défaut : Carte.vue

```vue
<template>
  <div class="carte">
    <div class="carte-body">
```

```
      <!-- Slot par defaut -->
      <slot>Contenu par defaut si rien n'est passe</slot>
    </div>
  </div>
</template>

<style scoped>
.carte {
  border: 1px solid #ddd;
  border-radius: 8px;
  padding: 16px;
}
</style>
```

```
<template>
  <!-- Le contenu remplace le slot -->
  <Carte>
    <h2>Mon titre</h2>
    <p>Mon contenu personnalise</p>
  </Carte>

  <!-- Sans contenu : affiche "Contenu par defaut..." -->
  <Carte />
</template>
```

7.2.1 Slots nommés

Listing 31 – Slots nommés : Layout.vue

```
<template>
  <div class="layout">
    <header>
      <slot name="header">Header par defaut</slot>
    </header>

    <main>
      <slot>Contenu principal</slot>
    </main>

    <footer>
      <slot name="footer">Footer par defaut</slot>
    </footer>
  </div>
</template>
```

```
<template>
  <Layout>
    <template #header>
      <h1>Mon Application</h1>
      <nav>...</nav>
    </template>
```

```
7
8       <!-- Contenu par defaut (slot sans nom) -->
9       <p>Contenu de la page</p>
10
11      <template #footer>
12        <p>&copy; 2025 Mon App</p>
13      </template>
14    </Layout>
15  </template>
```

7.2.2 Slots scopés (scoped slots)

Listing 32 – Slot scopé : ListeItems.vue

```
1  <script setup lang="ts">
2  interface Item {
3    id: number
4    nom: string
5    prix: number
6  }
7
8  defineProps<{
9    items: Item[]
10 }>()
11 </script>
12
13 <template>
14   <ul>
15     <li v-for="item in items" :key="item.id">
16       <!-- Passer des donnees au slot -->
17       <slot :item="item" :index="items.indexOf(item)">
18         {{ item.nom }}
19       </slot>
20     </li>
21   </ul>
22 </template>
```

```
1  <script setup lang="ts">
2  const produits = [
3    { id: 1, nom: 'Produit A', prix: 29.99 },
4    { id: 2, nom: 'Produit B', prix: 49.99 }
5  ]
6  </script>
7
8  <template>
9    <ListeItems :items="produits">
10     <!-- Recevoir les donnees du slot -->
11     <template #default="{ item, index }">
12       <span>{{ index + 1 }}. {{ item.nom }} - {{ item.prix }} EUR
           </span>
13     </template>
```

```
    </ListeItems>

    <!-- Syntaxe destructuree -->
    <ListeItems :items="produits" v-slot="{ item }">
      <strong>{{ item.nom }}</strong>
    </ListeItems>
  </template>
```

7.3 Provide et Inject

`provide` et `inject` permettent de partager des données entre ancêtres et descendants sans passer par les props :

Listing 33 – Provide dans un ancêtre

```
<!-- App.vue ou composant parent -->
<script setup lang="ts">
import { provide, ref, readonly } from 'vue'

// Donnees a partager
const theme = ref('clair')
const utilisateur = ref({ nom: 'Alice', role: 'admin' })

// Provide avec cle string
provide('theme', theme)

// Provide en lecture seule (recommande)
provide('utilisateur', readonly(utilisateur))

// Provide avec fonction pour modifier
provide('changerTheme', (nouveauTheme: string) => {
  theme.value = nouveauTheme
})
</script>
```

Listing 34 – Inject dans un descendant

```
<!-- Composant enfant/petit-enfant -->
<script setup lang="ts">
import { inject, type Ref } from 'vue'

// Inject avec valeur par defaut
const theme = inject<Ref<string>>('theme', ref('clair'))

// Inject requis (erreur si non fourni)
const utilisateur = inject('utilisateur')

// Inject de fonction
const changerTheme = inject<(t: string) => void>('changerTheme')

function toggleTheme() {
  changerTheme?.(theme.value === 'clair' ? 'sombre' : 'clair')
```

```
}
</script>

<template>
  <p>Theme: {{ theme }}</p>
  <p>Utilisateur: {{ utilisateur?.nom }}</p>
  <button @click="toggleTheme">Changer theme</button>
</template>
```

7.3.1 Utiliser des symboles comme clés

Listing 35 – Clés symboliques typées

```
// keys.ts
import type { InjectionKey, Ref } from 'vue'

export interface Utilisateur {
  nom: string
  role: string
}

export const utilisateurKey: InjectionKey<Ref<Utilisateur>> =
   Symbol('utilisateur')
export const themeKey: InjectionKey<Ref<string>> = Symbol('theme'
   )
```

```
// Parent
import { provide, ref } from 'vue'
import { utilisateurKey, themeKey } from './keys'

provide(utilisateurKey, ref({ nom: 'Alice', role: 'admin' }))
provide(themeKey, ref('clair'))

// Enfant
import { inject } from 'vue'
import { utilisateurKey, themeKey } from './keys'

const utilisateur = inject(utilisateurKey)  // Type infere!
const theme = inject(themeKey)
```

7.4 Composants asynchrones

Listing 36 – Composant asynchrone

```
<script setup lang="ts">
import { defineAsyncComponent } from 'vue'

// Chargement paresseux (lazy loading)
const ComposantLourd = defineAsyncComponent(() =>
  import('./ComposantLourd.vue')
```

```
)

// Avec options
const ComposantAvecOptions = defineAsyncComponent({
  loader: () => import('./ComposantLourd.vue'),
  loadingComponent: LoadingSpinner,
  errorComponent: ErrorDisplay,
  delay: 200,       // Delai avant d'afficher le loading
  timeout: 3000     // Timeout avant erreur
})
</script>

<template>
  <ComposantLourd />
  <ComposantAvecOptions />
</template>
```

8 Requêtes HTTP

8.1 Utiliser fetch() dans Vue

Listing 37 – Requêtes HTTP avec fetch

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

interface Utilisateur {
  id: number
  name: string
  email: string
}

const utilisateurs = ref<Utilisateur[]>([])
const loading = ref(false)
const erreur = ref<string | null>(null)

// GET
async function chargerUtilisateurs() {
  loading.value = true
  erreur.value = null

  try {
    const response = await fetch('https://jsonplaceholder.
       typicode.com/users')

    if (!response.ok) {
      throw new Error(`Erreur HTTP: ${response.status}`)
    }

    utilisateurs.value = await response.json()
  } catch (e) {
    erreur.value = e instanceof Error ? e.message : 'Erreur
       inconnue'
  } finally {
    loading.value = false
  }
}

// POST
async function creerUtilisateur(data: Partial<Utilisateur>) {
  const response = await fetch('https://jsonplaceholder.typicode.
     com/users', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(data)
  })
```

```
43
44    return await response.json()
45  }
46
47  // DELETE
48  async function supprimerUtilisateur(id: number) {
49    await fetch(`https://jsonplaceholder.typicode.com/users/${id}`,
          {
50      method: 'DELETE'
51    })
52  }
53
54  // PATCH
55  async function modifierUtilisateur(id: number, data: Partial<
       Utilisateur>) {
56    const response = await fetch(`https://jsonplaceholder.typicode.
        com/users/${id}`, {
57      method: 'PATCH',
58      headers: {
59        'Content-Type': 'application/json'
60      },
61      body: JSON.stringify(data)
62    })
63
64    return await response.json()
65  }
66
67  onMounted(() => {
68    chargerUtilisateurs()
69  })
70  </script>
71
72  <template>
73    <div v-if="loading">Chargement...</div>
74    <div v-else-if="erreur" class="erreur">{{ erreur }}</div>
75    <ul v-else>
76      <li v-for="user in utilisateurs" :key="user.id">
77        {{ user.name }} - {{ user.email }}
78      </li>
79    </ul>
80  </template>
```

8.2 Créer un composable pour les requêtes

Listing 38 – Composable useFetch

```
1  // composables/useFetch.ts
2  import { ref, type Ref } from 'vue'
3
4  interface UseFetchReturn<T> {
5    data: Ref<T | null>
```

```
  loading: Ref<boolean>
  error: Ref<string | null>
  execute: () => Promise<void>
}

export function useFetch<T>(url: string): UseFetchReturn<T> {
  const data = ref<T | null>(null) as Ref<T | null>
  const loading = ref(false)
  const error = ref<string | null>(null)

  async function execute() {
    loading.value = true
    error.value = null

    try {
      const response = await fetch(url)
      if (!response.ok) throw new Error(`HTTP ${response.status}`
          )
      data.value = await response.json()
    } catch (e) {
      error.value = e instanceof Error ? e.message : 'Erreur'
    } finally {
      loading.value = false
    }
  }

  return { data, loading, error, execute }
}
```

Listing 39 – Utilisation du composable

```
<script setup lang="ts">
import { onMounted } from 'vue'
import { useFetch } from '@/composables/useFetch'

interface Post {
  id: number
  title: string
  body: string
}

const { data: posts, loading, error, execute } = useFetch<Post
    []>(
  'https://jsonplaceholder.typicode.com/posts'
)

onMounted(execute)
</script>

<template>
  <button @click="execute" :disabled="loading">Recharger</button>

```

```
21    <div v-if="loading">Chargement...</div>
22    <div v-else-if="error">{{ error }}</div>
23    <article v-else v-for="post in posts" :key="post.id">
24      <h2>{{ post.title }}</h2>
25      <p>{{ post.body }}</p>
26    </article>
27  </template>
```

9 Composants natifs avancés

9.1 KeepAlive

<KeepAlive> met en cache les composants dynamiques :

Listing 40 – KeepAlive

```vue
<script setup lang="ts">
import { ref, shallowRef } from 'vue'
import TabA from './TabA.vue'
import TabB from './TabB.vue'

const ongletActif = shallowRef(TabA)
</script>

<template>
  <button @click="ongletActif = TabA">Tab A</button>
  <button @click="ongletActif = TabB">Tab B</button>

  <!-- Sans KeepAlive : le composant est detruit/recree -->
  <component :is="ongletActif" />

  <!-- Avec KeepAlive : le composant est mis en cache -->
  <KeepAlive>
    <component :is="ongletActif" />
  </KeepAlive>

  <!-- Options -->
  <KeepAlive :include="['TabA']" :exclude="['TabB']" :max="10">
    <component :is="ongletActif" />
  </KeepAlive>
</template>
```

Listing 41 – Hooks onActivated/onDeactivated

```vue
<script setup lang="ts">
import { onActivated, onDeactivated } from 'vue'

// Appele quand le composant est reactive (depuis le cache)
onActivated(() => {
  console.log('Composant active')
})

// Appele quand le composant est desactive (mis en cache)
onDeactivated(() => {
  console.log('Composant desactive')
})
</script>
```

9.2 Teleport

<Teleport> rend du contenu ailleurs dans le DOM :

Listing 42 – Teleport

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

const modalOuverte = ref(false)
</script>

<template>
  <button @click="modalOuverte = true">Ouvrir modale</button>

  <!-- Le contenu est rendu dans <body>, pas ici -->
  <Teleport to="body">
    <div v-if="modalOuverte" class="modal-overlay">
      <div class="modal">
        <h2>Ma Modale</h2>
        <p>Contenu de la modale</p>
        <button @click="modalOuverte = false">Fermer</button>
      </div>
    </div>
  </Teleport>

  <!-- Teleport vers un element specifique -->
  <Teleport to="#notifications">
    <div class="notification">Message!</div>
  </Teleport>

  <!-- Teleport conditionnel -->
  <Teleport to="body" :disabled="!isMobile">
    <nav>Navigation</nav>
  </Teleport>
</template>
```

9.3 Suspense

<Suspense> gère les composants asynchrones avec états de chargement :

Listing 43 – Composant async (setup async)

```vue
<!-- ComposantAsync.vue -->
<script setup lang="ts">
// Le setup peut etre async!
const response = await fetch('https://api.example.com/data')
const data = await response.json()
</script>

<template>
  <div>{{ data }}</div>
</template>
```

Listing 44 – Utilisation de Suspense

```vue
<script setup lang="ts">
import { onErrorCaptured, ref } from 'vue'
import ComposantAsync from './ComposantAsync.vue'

const erreur = ref<Error | null>(null)

onErrorCaptured((e) => {
  erreur.value = e
  return false  // Empeche la propagation
})
</script>

<template>
  <div v-if="erreur">Erreur: {{ erreur.message }}</div>

  <Suspense v-else>
    <!-- Contenu principal (async) -->
    <template #default>
      <ComposantAsync />
    </template>

    <!-- Affiche pendant le chargement -->
    <template #fallback>
      <div class="loading">
        <span>Chargement en cours...</span>
      </div>
    </template>
  </Suspense>
</template>
```

10 Résumé et points clés

À retenir

Réactivité :

- `ref()` : pour toutes les valeurs, accès avec `.value` dans le script
- `reactive()` : pour les objets, accès direct (éviter la déstructuration)
- `computed()` : valeurs calculées et mises en cache
- `watch()` / `watchEffect()` : réagir aux changements

À retenir

Directives :

- `v-bind` (`:`) : liaison d'attributs
- `v-on` (`@`) : écoute d'événements
- `v-model` : liaison bidirectionnelle
- `v-if` / `v-show` : affichage conditionnel
- `v-for` : boucles (toujours avec `:key`)

À retenir

Composants :

- `defineProps` : recevoir des données du parent
- `defineEmits` : émettre des événements vers le parent
- `defineExpose` : exposer des méthodes/propriétés
- Slots : passer du contenu template
- `provide` / `inject` : partager des données en profondeur

À retenir

Cycle de vie :

- `onMounted` : après le montage (DOM disponible)
- `onUnmounted` : nettoyage (timers, listeners)
- `onUpdated` : après une mise à jour

À retenir

Composants natifs :

- `<KeepAlive>` : mettre en cache les composants
- `<Teleport>` : rendre ailleurs dans le DOM
- `<Suspense>` : gérer le chargement asynchrone

Félicitations !

Vous maîtrisez maintenant les fondamentaux de Vue.js 3 avec la Composition API.

Pour aller plus loin : Vue Router, Pinia, Tests unitaires, SSR avec Nuxt...