zenika

Formation VUE 3

Généralités

VueJS a été créé en Février 2014 par Evan You. Son équipe et lui-même sont les seuls à maintenir le framework depuis (et non par un membre de la famille des GAFAM).

Depuis février 2022, Vue 3 est la version par défaut.

Généralités

VueJS fait partie des projets les plus populaires sur Github.

GitHub est un service web utilisé pour le contrôle de versions, il est fréquemment utilisé pour héberger des projets open source.

Généralités

VueJS est généralement utilisé pour créer des *Single-Page Applications* (SPA) et est également conçu pour pouvoir être adopté progressivement avec d'autres bibliothèques Javascript.

Plein de concepts utilisés par VueJS sont inspirés par d'autres librairies ou frameworks comme **AngularJS**, **React**, **Svelte**.

Documentations :

- Documentation officielle VueJS official doc
- API VueJS
- Tutoriel VueJS

Généralités

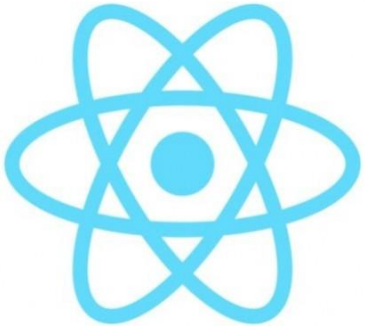
Concepts partagés avec Angular

- Syntaxe (v-if vs ng-if)
- Two-way binding
- Directives
- Watchers

Généralités

Concepts partagés avec React

- Virtual DOM
- Fonctions de rendu & JSX Support
- Hooks (Composition API)



Généralités

Concepts partagés avec Svelte

- Amélioration de la syntaxe pour les Single File Components

Create App

```
const vm = Vue.createApp({
   // options
})
```

Une application Vue a une instance principale qui peut être organisée comme un abres de composants réutilisables.
L'instance de l'application est déclarée dans la fonction Vue : createApp.

Chaque option est déclarée avec un objet `options` qui représente le composant principal de l'application.

`let vm = new Vue({})` deprecated:
- "Uncaught TypeError: Vue is not a constructor"

Create App

```
<div id="app">
   <h1>Hello World!</h1>
</div>

const app = Vue.createApp({})
app.mount('#app')
```

Pour lier l'instance de notre application à un élément du DOM, on utilise la méthode app.mount
Par convention, on utilise les noms de variables `vm` (ViewModel) ou `app`.

- `{ el: '#app' }` deprecated.
- Attention, `createApp()` retourne une instance de l'application, `mount(el)` retourne un Proxy.

Travaux pratiques

Lab 1

Vite

Vue Cli était l'outils webpack officiel de Vue, il est maintenant en maintenance.
Cette outils permettait facilement de créer, configurer et faire évoluer un projet Vue.

Vite est maintenant l'outils qui permet les déploiements plus rapide et léger d'application web modernes.

Il se compose de deux parties majeures :

- Un serveur de développement qui offre des améliorations par rapport aux modules ES natifs (basé sur `esbuild`).
- Une commande de "build" qui permet un regroupement du code.

Création d'un projet

create-vue est l'outils officiel qui permet la création d'un projet Vue.

La commande `npm init vue` permet de sélectionner les options à intégrer au projet lors de sa création

```
~/Apps $ npm init vue

Vue.js - The Progressive JavaScript Framework

✔ Project name: … vue-project
✔ Add TypeScript? … No / Yes
✔ Add JSX Support? … No / Yes
✔ Add Vue Router for Single Page Application development? … No / Yes
✔ Add Pinia for state management? … No / Yes
✔ Add Vitest for Unit Testing? … No / Yes
✔ Add Cypress for End-to-End testing? … No / Yes
✔ Add ESLint for code quality? … No / Yes
✔ Add Prettier for code formatting? … No / Yes

Scaffolding project in /home/jeremy/Apps/vue-project...

Done. Now run:

  cd vue-project
  npm install
  npm run lint
  npm run dev
```

Vite config

```
// vite.config.js
import { fileURLToPath, URL } from 'node:url'

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url))
    }
  }
})
```

La [configuration de Vite] peut être modifier dans le fichier `vite.config.js` (ci-dessus la configuration par défaut.

Vite, variables d'environnements

```
// .env
VITE_APP_NAME=My TV shows
APP_NAME=My TV shows

// src/main.js
console.log(import.meta.env.VITE_APP_NAME) // log "My TV shows" from VITE_APP_NAME
console.log(import.meta.env.APP_NAME) // log undefined because the env variable APP_NAME is not exposed in your
Vite app
```

Vite supporte les variables d'environnements. On peut y accéder dans le code via l'objet `import.meta.env`. En production ces variables sont remplacées statiquement.

Seules les variables préfixées **VITE_** sont accessible par le code traité par Vite.

VETUR / VOLAR

VS code a une vaste catalogue d'extensions pour améliorer l'expérience développeur.

Des extensions existent pour `Vue` et permettent à l'IDE de traiter les fichiers `*.vue` en proposant des fonctionnalités comme :

- Corrections de la syntaxe et sémantique
- Code snippet
- Lintage et vérification d'erreurs
- Formatage …

Vue 2 : Vetur
Vue 3 : Volar

`Volar` a été créé pour Vue 3 et présente de meilleures performance que `Vetur`.

Composants en vue 3

Options

Rappels

- Les composants sont des éléments HTML personnalisés.
- Ils permettent d’encapsuler du code réutilisable.
- Chaque composant doit avoir son propre comportement.

Composants en vue 3

Options

Déclaration globale.

`app.component(tagName, options)` enregistre un composant au niveau de l'application ou **globalement**.

```
import { createApp } from 'vue'
import BaseButton from './components/BaseButton.vue'

const app = createApp({})
app.component(BaseButton)
app.mount('#app')
```

On peut ainsi l'instancier dans un template comme ceci :

```
<div id="app">
    <BaseButton />
</div>
```

Composants en vue 3

Options

Déclaration locale.

Dans Vue, avec Vite ou Vue CLI, on utilise généralement l'import et l'enregistrement de composants localement.

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

export default {
  components: {
    ComponentA
  }
}
```

```
<ComponentA />
```

SFC Playground

Composants en vue 3

Options

Pour définir un composant, il est recommandé d'utiliser la fonction defineComponent

```
import { defineComponent } from 'vue'

export default defineComponent({
  data() {
    return {
      message: 'Hello Vue'
    }
  }
})
```

```
<template>
  <h1>{{ message }}</h1>
</template>
```

```
<!-- Rendering -->
<h1>Hello Vue</h1>
```

Comme pour l'instance de notre application, on peut intégrer de la donnée, grâce à la méthode data, dans notre composant. Chaque composant aura ainsi sa propre instance de ses propres données sans avoir à les partager avec d'autres composants.

La donnée peut donc être utilisé dans le template

SFC Playground

Composants en vue 3

Options, props

- De la donnée peut être passée à un composant enfant via les props
- Elles doivent être déclarées explicitements.
- Elles peuvent être utilisées dans les templates.

```
// src/components/StrongMessage.vue
export default {
  props: ['message']
}
```

```
<template>
  <h1>{{ message }}</h1>
</template>
```

```
import StrongMessage from
'@/components/StrongMessage.vue'

export default {
  components: { StrongMessage }
}
```

```
<template>
  <StrongMessage message="Message from my
parent" />
</template>
```

SFC Playground

Composants en vue 3

Options, props

Pour donner un type à une props, on peut lister celle-ci dans un objet

```
import { defineComponent } from 'vue'

export default defineComponent({
    props: {
        message: String
    },
})
```

Et plus encore ...

```
import { defineComponent } from 'vue'

export default defineComponent({
    props: {
        message: {
            type: String,
            required: false,
            default: 'Default message',
            validator: value => value.length > 0
        }
    },
})
```

Composants en vue 3

Options, props

On donnera, depuis un parent, l'expression à la props du composant avec `v-bind`.

```
import { defineComponent } from 'vue'
export default defineComponent({
  data() {
    return {
      description: 'Short show description',
      isFavorite: true,
    }
  }
})
```

```
<custom-component
  :is-favorite="isFavorite"
  :message="description"
></custom-component>
```

Composants en vue 3

Options, methods

- Methods, ce sont des fonctions déclarées dans l'instance de notre composant
- On utilise le mot clé methods.
- Elles ne peuvent être utilisées que dans le périmètre (scope) du composant.

```
// src/components/LikesButton.vue
export default {
  data() {
    return { likes: 0 }
  },
  methods: {
    addLike() { this.likes++ }
  }
}
```

```
<template>
  <button @click="addLike">{{ likes }}</button>
</template>
```

```
import LikesButton from
'@/components/LikesButton.vue'
export default {
  components: {
    LikesButton
  }
}
```

```
<template>
  <LikesButton />
</template>
```

SFC Playground

Composants en vue 3

Options, events

- Il est possible d'émettre et d'écouter des événements
- Le composant parent pourra intercepter ces événements grâce à la directive v-on.
- On listera les événements sous la clé `emits`. *Ceci reste facultatif mais permet une meilleure documentation du code.*
- Dans l'exemple ci-dessous, `myFunction` est exécuter chaque fois que la méthode `validate` du composant `child` est exécuté

```
  ...
  emits: ['my-event'],
  methods: {
    validate() {
      this.$emit('my-event')
    }
  }
}
```

```
<child @my-event="myFunction"></child>
```

SFC Playground

Composants en vue 3

Options, events

Depuis le composant enfant, il est possible de passer des arguments dans le deuxième paramètre de la fonction $emit

```
{
  ...
  emits: ['my-event']
  methods: {
    validate() {
      this.$emit('my-event', { data: 'some data' })
    }
  }
}
```

Dans le composant parent :

```
<child @my-event="myFunction"></child>
```

```
{
  ...
  methods: {
    myFunction(payload) {
      // payload === { data: 'some data' }
    }
  }
}
```

SFC Playground

Composants en vue 3

Options, events

Comme pour les props, il est possible de valider le type d'un événement en définissant un objet plutôt qu'un tableau.

```
export default {
    emits: {
        // No validation
        click: null,
        // Validate increment event
        increment: (value) => {
            if (!value || typeof value !== 'number') {
                console.warn('Invalid increment event payload !')
                return false
            }
            return true
}}}
```

```
<template>
      <button @click="$emit('increment', 2)">
            increment
      </button>
</template>
```

SFC Playground

Composants en vue 3

Options, computed

- Les propriétés `computed` sont utiles lorsque l'on veut éviter d'avoir des expressions complexes dans notre template.
- Une propriété `computed` sera réévaluée à chaque fois que l'une de ses dépendance sera mise à jour.

```
export default {
  data () {
    return {
      firstName: 'Evan',
      lastName: 'You'
    }
  },
  computed: {
    fullName () {
      return `${this.firstName} ${this.lastName}`
    }
  }
}
```

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

<!-- Renders -->
<div>Evan You</div>
```

SFC Playground

Composants en vue 3

Options, watcher

- Avec les `computed properties` on couvre la plupart des cas.
- Sinon, pour d'autres cas spécifiques, on peut utiliser les `watchers`.
- Utiles pour modifier la base de données lors d'opérations lourdes ou asynchrones.

```
export default {
  data () {
    return { count: 0 }
  },
  watch: {
    count(newValue, oldValue) {
      console.log(newValue, oldValue)
    }
  }
}
```

```
<template>
  <button @click="count++">{{count}}</button>
</template>
```

SFC Playground

Composants en vue 3

Options, watcher

N'utiliser un `watcher` que lorsque c'est pertinent. Utiliser `computed` sera, dans la plupart des cas, une meilleure approche.

```
export default {
  data() {
    return { firstName: '', reversedFirstName: '' }
  },
  watch: {
    firstName(value) {
      this.reversedFirstName = value.split('').reverse().join('')
    }
  }
}
```

```
<div>
  <input type="text" v-model="firstName"/>
  <span>{{ reversedFirstName }}</span>
</div>
```

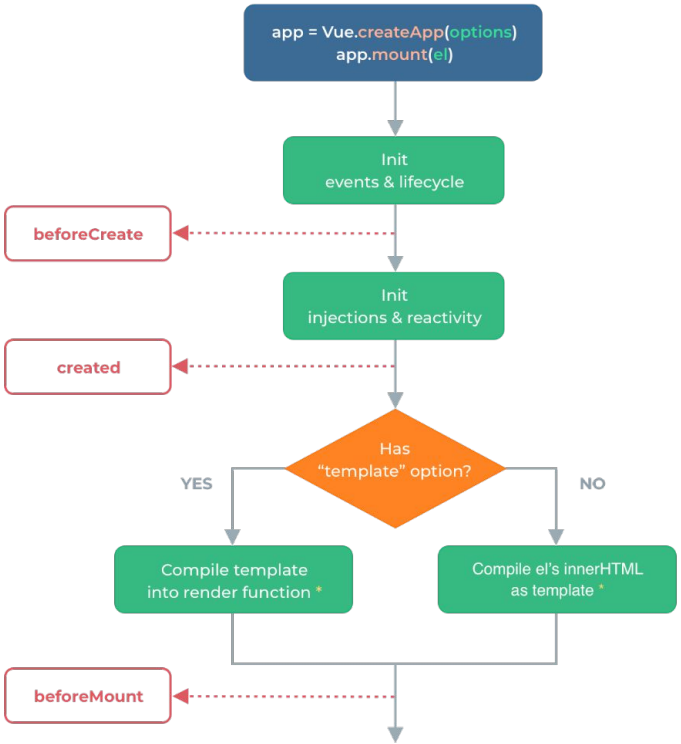
Lifecycle

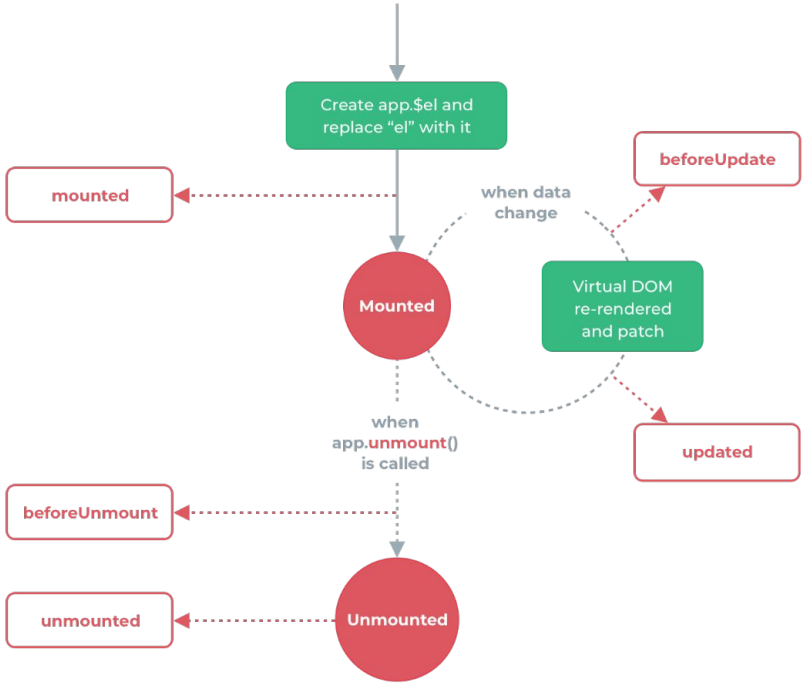
- **Lifecycle hooks** permettent de mettre votre propre code à certaines étapes spécifiques de la création du composant
- **Lifecycle hooks** ne bloquent pas le rendu

SFC Playground

```
export default {
    created() {
      // lifecycle hook appropriate for an API call
      console.log("I'm created");
    },
    mounted() {
      // interaction with the DOM possible
      console.log("I'm mounted");
    },
    beforeUnmount() {
      // can be used to clean up timers, listeners
      console.log("I'm beforeUnmount");
    },
}
```

Lifecycle





* Template compilation is performed ahead-of-time if using a build step, e.g., with single-file components.

Composants v-model

`v-model` peut aussi être utilisé sur des composants personnalisés. Celui-ci doit définir une prop: `modelValue` et emettre l'événement: `update:modelValue`.

```
// BaseInput.vue
import { defineComponent } from 'vue'

export default defineComponent({
  props: ['modelValue'],
  emits: ['update:modelValue']
})
```

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

[SFC Playground]

Maintenant la directive `v-model` peut être appelée

```
import { defineComponent } from 'vue'
import BaseInput from './BaseInput.vue'

export default defineComponent({
  components: { BaseInput },
  data () {
    return { message: '' }
  }
})
```

```
<template>
  <base-input v-model="message" />
</template>
```

Travaux pratiques

Lab 3

Composants en vue 3

Composition

Les composants Vue peuvent être écrit de deux façon différentes : `Options API` ou `Composition API`.

Les deux styles couvrent la totalité des besoins, cependant on remarque quelques bénéfices à utiliser `Composition API` :

- Meilleure logique de réutilisation
- Une organisation du code plus flexible
- Meilleure déduction des types

Composition API FAQ

Composants en vue 3

Composition

Composition API est une notion nouvelle à Vue 3. On la reconnaît à l'utilisation de la méthode setup qui est une nouvelle façon d'écrire nos composants.

```
export default {
  props: {
    user: { type: String }
  },
  setup(props) {
    console.log(props) // { user: '' }

    return {} // anything returned here will be available for the rest of the component
  },
  ...
}
```

Composants en vue 3

Composition, méthode setup

Setup est exécutée dans le cycle de vie du composant **avant** que celui-ci ne soit créé, une fois la résolution des `props` effectuée, mais avant la possibilité d'avoir accès au `this`.

C'est le seul moment dans un composant Vue que l'on pourra utiliser la réactivité.

Exception fait des `props`, il sera impossible d'accéder aux propriétés déclarées dans le composant (statut local, propriétés `computed`, méthodes ...)

Il faudra retourner de la méthode `setup` tout ce que l'on souhaite exposer au reste du composant ainsi qu'à son **template**.

Composants en vue 3

Composition, réactivité

JavaScript est exécuté **ligne par ligne** et les changements n'impactent pas les précédentes déclarations

```
let val1 = 2
const val2 = 3
const sum = val1 + val2

console.log(sum)
// 5

val1 = 3

console.log(sum)
// sum is still 5...
```

Avec l'**API de composition** vient l'**API Réactive**

Composants en vue 3

Compositions, réactivité

API Options et **Api de composition** utilisent la réactivité de manière intrinsèque.

Toutes deux peuvent être utilisées, même dans un même composant.
L'objet retourné par setup() peut être utilisé à l'intérieur de l'API Options, mais l'inverse ne sera pas possible.

```
import { ref } from 'vue'

export default {
  setup() { // composition API
    const counter = ref(0)
    return { counter }
  },
  mounted () {
    console.log("counter: ", this.counter) // logs 0
  }
}
```

SFC Playground

Composants en vue 3

Compositions, donnée

Pour créer un état réactif à partir d'une valeur primitive on utilise la méthode ref

```
import { ref } from 'vue'

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

    console.log(counter) // { value: 0 }
    console.log(counter.value) // 0
  },
}
```

```
<template><div>counter: {{ counter }}</div></template>
```

ref va envelopper notre objet, on devra donc utiliser `.value` pour accéder à sa valeur.

Les valeurs de `ref` sont automatiquement sorties de l'enveloppe dans le template ce qui signifie que l'on aura pas à utiliser `count.value`

Composants en vue 3

Compositions, donnée

Pour créer un état réactif à partir d'un objet, on utilise la méthode reactive

```
import { reactive } from 'vue'

export default {
  setup() {
    const state = reactive({
      display: false,
      userStatus: 'loggedOut'
    })

    state.display = true;
    console.log(state)
    // { display: true, userStatus: 'loggedOut' }

    return { state }
  }
}
```

```
<template>
  <div v-if="state.display">
    user status:
    {{ state.userStatus }}
  </div>
</template>
```

SFC Playground

Composants en vue 3

Compositions, donnée

Pour combiner des valeurs `reactive` on peut utiliser l'opérateur `...` avec la méthode toRefs:

```
import { reactive, toRefs } from 'vue'

export default {
  setup() {
    const data = reactive({
      display: true,
      userStatus: 'loggedOut'
    })

    return {...toRefs(data)} // ✅ { display, userStatus}
    // return { data } ✅ { data.display, data.userStatus}
    // return { ...data } ❌ breaks reactivity { display, userStatus}
  }
}
```

Sans cette méthode, l'opérateur casserait la réactivité et créerait un objet JS classique.

SFC Playground

Composants en vue 3

Compositions, méthodes

Les méthodes sont des fonctions déclarées puis retournées par la méthode setup(), `this` n'existe pas encore

```
import { ref } from 'vue'

export default {
  setup() {
    const display = ref(true)

    function toggleUserDisplay () {
      display.value = !display.value
    }

    return { display, toggleUserDisplay }
  }
}
```

```
<div v-if="display">hello</div>
<button @click="toggleUserDisplay">Toggle display</button>
```

SFC Playground

Composants en vue 3

Compositions, computed

Utiliser la méthode computed()
L'utilisation des `props` se fait dans la fonction setup(). Étant déjà exposées dans le template, elles n'ont pas besoin d'être retournées

```
import { computed } from 'vue'

export default {
  props: ['firstName', 'lastName'],
  setup(props) {
    const fullName = computed(() => `${props.firstName} ${props.lastName}`)

    return { fullName }
  }
}
```

```
<span>{{ fullName }}</span>
```

SFC Playground

Composants en vue 3

Compositions, watchEffect

La méthode watchEffect exécute une callback à chaque mise à jour d'une dépendance réactive.
On pourra utiliser la méthode watch pour avoir plus de contrôle sur le déclenchement de cette callback en la séparant du suivi de ces mises à jour.

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

export default {
  setup() {
    const counter = ref(0)
    watchEffect(() => {
      console.log('counter value:', counter.value)
    })
    return { counter }
  }
}
```

```
<button @click="counter++">counter: {{ counter }}</button>
```

SFC Playground

Composants en vue 3

Compositions, événements

Pour envoyer un événement, on utilise le deuxième paramètre de la méthode setup

```
export default {
  emits: ['my-event'],
  setup(props, { emit }) {
    const emitEvent = () => {
      emit('my-event', {data: 'some data'})
    }

    return { emitEvent }
  }
}
```

```
<button @click="emitEvent">Emit an event</button>
```

Le deuxième paramètre est un `context` qui contient d'autre objets non-réactifs. Ce qui signifie que l'on peut le destructurer sans avoir à se préoccuper de conserver la réactivité.

SFC Playground

Composants en vue 3

Compositions

Voici la signature d'un composant écrit avec l'API composition:

```
export default {
  name: 'ComponentName',
  components: { /* ... */},
  props: [/* ... */],
  emits: [/* ... */],
  setup() {
    // ...
  }
}
```

Composants en vue 3

Compositions, cycle de vie

Pour écrire du code déclenché dans le cycle de vie d'un composant avec l'API de composition, on utilisera le préfix on: i.e. mounted devient onMounted.

```
import { onMounted } from 'vue'

export default {
  setup () {
    onMounted(() => {
      console.log('on mounted')
    })
  }
}
```

SFC Playground

Travaux pratiques

Lab 4.1

Composants en vue 3

Compositions, script setup

La syntaxe script setup est recommandée dès lors que l'on utilise l'API de composition dans un Single File Components (SFCs)

Elle introduit de nombreux avantages vis à vis de la syntaxe `<script>` classique:

- Réduction du nombre de lignes de code
- Déclaration des props et événements en utilisant du pur TypeScript
- Meilleur performance à l'exécution (le template est compilé dans une fonction de rendu sans proxy intermédiaire
- Meilleure interprétation par les IDE (notamment pour la déduction des types)

Composants en vue 3

Compositions, script setup

Tout ce qui a été vu avec la méthode setup peut être utilisé dans la section script setup. Un autre avantage est que le binding de haut niveau est directement utilisable dans le template sans valeur de retour.

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

const counter = ref(0)
const increment = () => { counter.value++ }
const doubleCounter = computed(() => counter.value * 2)
watch(counter, () => {
  console.log('counter updated', counter.value)
})
</ script>

<template>
  <button @click="increment">{{ counter }}</button>
  <div>double counter: {{ doubleCounter }}</div>
</template>
```

SFC Playground

Composants en vue 3

Compositions, script setup

L'importation des composant suffit pour qu'ils soient utilisables sans avoir besoin de les enregistrer.

```
<script setup>
import HelloWorld from '@/components/HelloWorld.vue'
</ script>

<template>
  <HelloWorld />
</template>
```

SFC Playground

Composants en vue 3

Compositions, script setup

Pour définir les props et les événements, on utilise le macro-compilateur defineEmits et defineProps. Ils n'ont pas besoin d'être importés. Ils acceptent les même valeurs que les options `emits` et `props` de l'API Options.

```
<script setup>
const props = defineProps({
  message: {
    type: String,
    required: true
  },
})
const emits = defineEmits(['toggle-favorite'])

emits('toggle-favorite')
console.log(props.message)
</script>
```

SFC Playground

Travaux pratiques

Lab 4.2

Slots

Contenu par défaut

```
<!-- HTML of Parent component -->
<my-dialog>
  <div>This is the Dialog Content</div>
</my-dialog>

<!-- HTML of child component my-dialog-->
<section class="dialog">
  <h1>This is a dialog title</h1>
  <div class="dialogContent">
    <slot></slot>
  </div>
</section>
```

```
<!-- HTML produced -->
<section class="dialog">
  <h1>This is a dialog title</h1>
  <div class="dialogContent">
    <div>This is the Dialog Content</div>
  </div>
</section>
```

Avec les slots, il est possible de passer un fragment html à un composant enfant, celui-ci rendra le fragment dans son propre template.

SFC Playground

On appelle ça **Transclusion** dans d'autres frameworks.

Slots

Contenu par défaut

```
<section class="dialog">
  <h1>This is a dialog title</h1>
  <div class="dialogContent">
    <slot>Default content</slot>
  </div>
</section>
```

Il est possible d’ajouter un slot par défaut en ajoutant simplement du contenu dans la balise html

Slots

Slots nommés

```
<!-- HTML of Parent component -->
<my-dialog>
  <template v-slot:title>This is a dialog
title</template>
  <template #body>This is the Dialog
Content</template>
</my-dialog>

<!-- HTML of child component my-dialog-->
<section class="dialog">
  <h1><slot name="title"></slot></h1>
  <div class="dialogContent"><slot
name="body"></slot></div>
</section>
```

```
<!-- HTML produce-->
<section class="dialog">
  <h1>This is a dialog title</h1>
  <div class="dialogContent">
    <div>This is the Dialog Content</div>
  </div>
</section>
```

Il est également possible de nommer ses slots

Directives

En plus des directives données par défaut, Vue permet de créer des directives custom.
Exemple : `<input v-focus />`

On peut l'enregistrer globalement depuis `src/main.js`:

```
const app = createApp(App)
  .directive('focus', {
    mounted (el) {
      el.focus()
    }
  })
app.mount('#app')
```

Ou localement :

```
export default {
  directives: {
    focus: {
      mounted(el) {
        el.focus()
      }
    }
  }
}
```

Localement, le nom de la directive doit être déclaré en camelCase mais sera utilisé avec la syntaxe dash-case dans le template.

SFC Playground

Directives

L’objet de définition d’une directive peut être affecté à plusieurs endroits du cycle de vie :

- `beforeMount`: est appelé quand la directive est liée à son élément HTML et avant que le composant parent soit monté. (`bind` avec Vue@2)
- `mounted`: est appelé quand le lien de l’élément du composant parent est monté. (`inserted` avec Vue@2)
- `beforeUpdate`: est appelé avant que le VNode du composant contenant ne soit mis à jour
- `updated`: appelé après la mise à jour de VNode du composant contenant et des VNodes de ses enfants. (`componentUpdated` avec Vue@2, `update` a été supprimé)
- `beforeUnmount`: appelé avant que le composant parent de l'élément lié ne soit démonté
- `unmount`: appelé une seule fois, lorsque la directive est déliée de l'élément et que le composant parent est démonté. (`unbind` avec Vue@2)

Directives

Ils reprennent tous les mêmes arguments : `el`, `binding`, `vnode`, et `prevVnode`.

- `el`: L'élément auquel la directive est liée. Il peut être utilisée pour manipuler directement le DOM.
- `binding`: Un objet contenant certaines propriétés contextuelles, comme `value`, `arguments` ou `modifiers` fournis à la directive.

L'argument `binding` doit être traité comme un read-only et ne jamais modifier son contenu.

La plupart du temps, le même comportement est attendu pour `mounted` et `updated`, sans s'occuper des autres hooks.

Vue fournit une notation abrégée pour cela, en définissant la directive comme une fonction :

```
app.directive('focus', (el, binding) => {
  el.focus()
})
```

Plugins

Plugins

L’utilisation des plugins permet plusieurs choses :

- Enregistrer un ou plusieurs composants globaux ou des directives personnalisées avec app.component() et app.directive().
- Pouvoir faire des injections dans l’app en appelant app.provide().
- Ajouter des instances globales de propriétés ou méthodes avec app.config.globalProperties.
- Une librairie qui aurait besoin d’utiliser une combinaison des fonctionnalités ci-dessus (e.g. vue-router).

Plugins

Pour utiliser un plugin, il suffit d'appeler app.use()

```
import { createApp } from 'vue'
import App from './App.vue'
import myPlugin from './plugins/myPlugin'

const app = createApp(App)
app.use(myPlugin, { myPluginOption: true })
```

app.use() empêche automatiquement les initialisations multiples.

```
import myPlugin from './plugins/myPlugin'
// ...
app.use(myPlugin, { config: 1 })
app.use(myPlugin, { config: 2 })
app.use(myPlugin, { config: 3 })

// myPlugin is initialized with {config: 1}
```

Plugins

Il est aussi possible d’utiliser des plugins venant de [awesome vue]

Plugins

Écriture d'un plugin

```
// src/plugins/myPlugin.js
export default {
  install: (app, options) => {
  }
}
```

```
// src/plugins/myPlugin.js
export default function (app, options) {
}
```

Un plugin peut être défini des deux façons.

Depuis un plugin, tout est possible de faire avec [l'instance de l'application](#) :

```
export default {
  install: (app, options) => {
    app.component('my-component', { /* ... */ })
    app.directive('focus', (el) => el.focus())
    app.provide('message', 'hello vuejs')
  }
}
```

- Enregistrer un composant global avec app.component()
- Enregistre une directive personnalisée avec app.directive()
- Fournir de la donnée à chaque composants avec app.provide()

Plugins

Écriture d’un plugin

Il est aussi possible d’ajouter des propriétés globales en utilisant, app.config.globalProperties. On utilise généralement le préfixe $

```
export default {
  install: (app, options) => {
    app.config.globalProperties.$msg = 'hello vuejs'
  }
}
```

Il est maintenant possible d’accéder à $msg depuis tous les composants

```
export default {
  mounted() {
    console.log(this.$msg)
  }
}
```

Composables

Composables

Dans le contexte des applications Vue, un composable est une fonction qui tire parti de l' `API Composition` de Vue pour encapsuler et réutiliser la `logique d'état`.

Exemple avec un compteur :

```
// src/composables/useCounter.js
import { ref } from 'vue'

export function useCounter (initialCounter = 0) {
  const counter = ref(initialCounter)

  function increment() {
    counter.value++
  }
  return {
    counter,
    increment
  }
}
```

Composables

Utilisons maintenant le composable `useCounter()` :

```
import { useCounter } from '@/composables/useCounter'

const { counter, increment } = useCounter(4)
```

```
<template>
  <button @click="increment"> {{ counter }}</button>
</template>
```

SFC playground

Composables

VueUse est une riche collection de fonctions utilitaires basées sur l'API de composition.

Si vous avez un besoin spécifique pour votre application, il est très probable qu'une méthode existe déjà dans VueUse.

Travaux pratiques

Lab 5

Routing

vue-router

Installation

vue-router est la librairie officielle de routage pour Vue, elle permet aux développeurs de créer de SPA (Single Page Application) alternatives utilisant la navigation dans une application VUE en redirigeant les composants via l'URL.
Elle facilite les transitions entre différentes vues et gère l'état de navigation de l'application.

En utilisant create-vue il est possible de sélectionner vue-router ce qui permettra à la librairie de s'installer.
Sinon, il reste possible de l'installer via npm :

```
npm install vue-router
```

vue-router

Installation

On crée l'instance du router en définissant les routes et en activant l'historique.

```
// src/router.js

import Home from '@/views/Home.vue'
import About from '@/views/About.vue'

import { createRouter, createWebHistory } from 'vue-router'

const router = createRouter({
  // Provide the history implementation to use.
  history: createWebHistory(),
  // Define some routes. Each route should map to a component.
  routes:  [
    { path: '/', component: Home },
    { path: '/about', component: About },
  ],
})

export default router
```

vue-router

Installation

On enregistre le router dans l'app pour qu'elle puisse l'utiliser.

```
// src/main.ts

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

import router from '@/router'

const app = createApp(App)

app.use(router)

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

vue-router

Installation

Mise à jour du composant `App.vue` pour inclure `<router-view>` et `<router-link>`.

`<router-view>` est l'espace réservé pour le contenu rendu pour la route courante.

```
<template>
  <div>
    <!-- use the router-link component for navigation. -->
    <!-- specify the link by passing the `to` prop. -->
    <!-- `<router-link>` will render an `<a>` tag with the correct `href` attribute -->
    <router-link to="/">Go to Home</router-link>
    <router-link to="/about">Go to About</router-link>
  </div>

  <!-- component matched by the route will render here -->
  <router-view></router-view>
</template>
```

vue-router

Matching dynamique

On pourrait avoir le composant ShowDetails qui doit être rendu pour chaque Shows télévisés (ayant chacun un `id` différent).
Avec le router de Vue on utilisera pour ça les paramètres dynamique dans le `path`.

```
import { createRouter, createWebHistory } from
'vue-router'
import ShowDetails from
'@/views/ShowDetails.vue'

const router = createRouter({
  history: createWebHistory(),
  routes: [
    {
      path: '/shows/:id',
      component: ShowDetails
    }
  ]
})
```

```
<template>
  <div>Show details id: {{ $route.params.id
}}</div>
</template>
```

Ainsi, nous sommes capable de naviguer vers `shows/1` ou `shows/2` en utilisant le même composant ShowDetails.

vue-router

Matching dynamique

- Les props sont accessible depuis **$route.params**.
- Cela crée un couplage fort avec le chemin et limite la flexibilité du composant.
- On passera les props au chemin du composant.

```
import { createRouter, createWebHistory } from
'vue-router'
import ShowDetails from
'@/views/ShowDetails.vue'

const router = createRouter({
  history: createWebHistory(),
  routes: [
    {
      path: '/shows/:id',
      component: ShowDetails
      // define props to true to pass dynamic
params as props
      props: true
    }
  ]
})
```

```
<template>
      <div>Show details id: {{ id }}</div>
</template>

<script setup>
      defineProps(['id'])
</ script>
```

vue-router

Routes nommées

- Chaque route à une propriété `name` en plus de `path`.
- On peut le spécifier dans la déclaration de la route.

```
const routes = [
  {
    path: '/shows',
    name: 'showList',
    component: ShowList
  },
  {
    path: '/shows/:id',
    name: 'showDetails',
    component: ShowDetails
  }
]
```

```
<router-link :to="{ name: 'showList' }">
  TV Shows
</router-link>

<router-link :to="{
  name: 'showDetails',
  params: { id: show.id }
}">
      Game of thrones
</router-link>
```

vue-router

Redirection

Vue router permet la redirection, par exemple ici on redirige `/foo` vers `/bar`

```
const routes = [
  {
    path: '/foo',
    redirect: '/bar'
  }
]
```

On peut aussi gérer la redirection sous forme de fonction :

```
const routes = [
  {
    path: '/foo',
    redirect: (to) => {
      // the function receives the target route as the argument
      // we return a redirect path/location here.
    }
  }
]
```

vue-router

404 Not Found

Avec une regexp, il est possible de rediriger toutes les routes vers une 404

```
const routes = [
  {path: '/', name: 'homepage', component: Homepage},
  // will match everything and put it under `$route.params.pathMatch`
  {path: '/:pathMatch(.*)*', name: 'NotFound', component: NotFound},
]
```

vue-router

Lazy-loading

- Quand une application commence à grossir, il faut réfléchir à la performance.
- Quick win : ne charger une page seulement quand elle est appelée

Grâce à `() => import('...')` le fichier LazyLoadedPage sera chargé seulement quand il sera visité.

```
import Homepage from '@/views/HomePage.vue'

const routes = [
  {
    path: '/',
    component: Homepage
  },
  {
    path: '/other',
    component: () => import('./routes/LazyLoadedPage.vue')
  }
]
```

vue-router

Pour retrouver la route courante, la variable `$route` peut être lue dans un template

```
<template>
  <pre>
    {{ $route }}
  </pre>
</template>
```

Depuis `script`, on utilise la méthode `useRoute` si le composant est créé à partir de l'API Composition :

```
// Equivalent to using $route inside templates
const route = useRoute()

console.log(route)
```

Pour retrouver l'instance du router, la variable `$router` peut être lue dans un template

```
<template>
  <pre>
    {{ $router }}
  </pre>
</template>
```

Depuis `script`, on utilise la méthode `useRouter` si le composant est créé à partir de l'API Composition :

```
// Equivalent to using $router inside templates
const router = useRouter()

function goHome() {
  router.push('/')
}
```

vue-router

Guards

Les guards permettent de surveiller la navigation, si une ressources ne doit pas être accessible, une redirection ou une annulation sera mise en place par une guard.
On connaît 3 moyens de surveiller une route durant la navigation :

- Globalement
- Par routes
- Dans le composant

vue-router

Guards, global

En utilisant `router.beforeEach` ou `router.afterEach`:

```
const router = createRouter({
  // etc...
})

router.beforeEach(async (to, from) => {
  if (!canAccess()) {
    // redirect the user to the login page
    return '/login'
  }
})

router.afterEach((to, from) => {
  sendToAnalytics(to.fullPath)
})
```

vue-router

Guards, par routes

En utilisant la propriété `beforeEnter`

```
const routes = [
  {
    path: '/users/:id',
    component: UserDetails,
    beforeEnter: (to, from) => {
    },
  },
]
```

vue-router

Guards, dans le composant

```
import { onBeforeRouteLeave } from 'vue-router'

onBeforeRouteLeave(() => {
  const answer = window.confirm('Do you really want to leave? you have unsaved changes!')
  if (!answer) return false
})
```

Travaux pratiques

Lab 6

Stores

Stores

Gestion d’états

Dès lors qu’une application grossi, il est souvent nécessaire de pouvoir partager de la donnée entre plusieurs composants.

Plusieurs options sont disponibles pour gérer les états d’une application : L’API réactive et l’utilisation d’une librairie dédiée.

Vuex a été pendant longtemps la librairie officielle mais depuis la version 3 (version de Vue par défaut) Pinia est devenue la librairie recommandée pour la gestion d’états.

Stores

Pinia

Pinia est une librairie de stores. Elle permet de partager un état au travers de composants ou pages dans l'application.

La philosophie est similaire avec Vuex à l'exception qu'elle ne fait pas la différence entre mutation et actions.

Il n'existe que des actions et peuvent être synchrones ou asynchrones et changent l'état directement sans avoir recours à la mutation ce qui rend l'utilisation de la librairie plus simple.

La librairie Pinia s'interface directement avec Vuejs :

- Devtools
- Remplacement à chaud des modules
- Plugins pour étendre les fonctionnalités de Pinia et correspondrent au besoins de l'application.
- Supporte TypeScript
- Server Side Rendering

Aussi, Pinia a une sand-box.

Stores

Pinia

Installer la dépendance avec `npm install pinia`
Créer une instance Pinia avec la méthode createPinia en passant en paramètre l'instance de l'application

```
// src/main.js

import { createPinia } from 'pinia'

import { createApp } from 'vue'
import App from '@/App.vue'

const app = createApp(App)

app.use(createPinia())
```

Stores

Pinia

Un store est une entité qui stocke l'état et la logique métier de l'application.

On peut utiliser la méthode defineStore pour définir un store.

Il est possible d'utiliser deux syntaxes pour les définir :
- `Options`, similaire à l'API option
- `Composition` syntax, similaire à l'API Composition

Choisissez la syntaxe avec laquelle vous êtes le plus confortable.
Il est important de rester consistant dans un même projet.

Stores

Syntaxe Option

Le store a trois concepts : le **state**, les **getteurs** et les **actions**. Ce qui correspond au **data**, **computed** et **methods** déjà vu dans l'api option.

```
// src/stores/index.js

import { defineStore } from 'pinia'

// the first argument is a unique id of the store across your application
export const useStore = defineStore('main', {
  state () {
    return {
    }
  },
  getters: {
  },
  actions: {
  },
})
```

Stores

Syntaxe Option

Le `state` est défini comme une fonction qui retourne l'état initial.

```
// src/stores/index.js

import { defineStore } from 'pinia'

// the first argument is a unique id of the store across your application
export const useStore = defineStore('main', {
  state () {
    return {
      counter: 1
    }
  },
})
```

Stores

Syntaxe Option

```
// src/stores/index.js

import { defineStore } from 'pinia'

export const useStore = defineStore('main', {
  state () {
    return {
      counter: 0,
    }
  },
  getters: {
    // compute a value from the state
    doubleCount() {
      return this.counter * 2
    },
    // access other getter
    doubleCountPlusOne() {
      return this.doubleCount + 1
    },
  },
})
```

Les getters sont l'équivalent d'une valeur computed dans l'état du store.

Stores

Syntaxe Option

```
import { defineStore } from 'pinia'
import axios from 'axios'

export const useStore = defineStore('main', {
  state () {
    return {
      counter: 0
    }
  },
  actions: {
    increment() {
      this.counter++
    },
    async randomizeCounter() {
      const { data } = await
axios.get('http://www.randomnumberapi.com/api/v1.0/random')
      if (data.count && data.count.length > 0) {
        this.counter = data.count[0]
      }
    },
  },
})
```

Les actions sont l'équivalent des méthodes dans les composants. Elles peuvent être synchrones ou asynchrones et peuvent accéder à toute l'instance du store avec `this`.

Stores

Utilisation du store

Si le composant Vue est écrit avec l'API options, on peut utiliser les fonctions utilitaires de Pinia pour accéder au state, aux getters et aux actions. mapStores() permet l'accès à tout le store.

```
import { mapStores } from 'pinia'
import { useStore } from '@/stores'

export default {
  computed: {
    ...mapStores(useStore)
  },
  mounted() {
    console.log(this.mainStore.counter)
    this.mainStore.increment()
  },
}
```

En décomposant `...mapStores` dans `computed` on déclare une nouvelle variable dont le nom sera la concaténation du nom du store (`main` dans ce cas) et du suffixe `Store` => `this.mainStore`.

Stores

Syntaxe Composition

Avec la syntaxe de composition, on utilise la réactivité, on passe le store dans une fonction qui définit ses propriétés et méthodes réactives qu'il faudra retourner dans un objet.

```
import { defineStore } from 'pinia'
import { ref, computed } from 'vue'

export const useStore = defineStore('counter', () => {
  // ref()s become state properties
  const counter = ref(0)

  // computed()s become getters
  const doubleCounter = computed(() => counter.value * 2)

  // function()s become actions
  function increment () {
    counter.value++
  }

  return { counter, increment, doubleCounter }
})
```

Stores

Utilisation du store

Si le composant Vue est écrit avec l'`API Composition`, c'est plus évident.

On importe puis invoque le store.

Puis on a accès au state, getters et actions.

```
import { useStore } from '@/stores'

const store = useStore()
console.log(store.counter)
```

Migration Vue2 vers Vue3

@vue/compat

Qu'est-ce que `@vue/compat` ?

- Un outil de compatibilité pour aider à la migration de Vue 2 à Vue 3.
- Permet de faire fonctionner les composants et plugins Vue 2 dans un environnement Vue 3.
- Facilite la transition progressive en identifiant et en gérant les ruptures de compatibilité.

Migration Vue2 vers Vue3

@vue/compat

Pourquoi utiliser `@vue/compat` ?

- Transition progressive sans réécriture complète du code.
- Support continu des fonctionnalités Vue 2.
- Identification des zones du code nécessitant des mises à jour pour Vue 3.

Migration Vue2 vers Vue3

Installation @vue/compat

```
npm install @vue/compat
```

```
// main.js
import { createApp } from 'vue'
import App from './App.vue'

const app = createApp(App)
app.config.compatConfig = { MODE: 2 }  // Activer le mode compat
app.mount('#app')
```

Migration Vue2 vers Vue3

Configuration @vue/compat avec vue-cli

```
// vue.config.js
module.exports = {
  chainWebpack: (config) => {
    config.resolve.alias.set('vue', '@vue/compat')

    config.module
      .rule('vue')
      .use('vue-loader')
      .tap((options) => {
        return {
          ...options,
          compilerOptions: {
            compatConfig: {
              MODE: 2
            }
          }
        }
      })
  }
}
```

Migration Vue2 vers Vue3

Configuration @vue/compat avec vite

```
// vite.config.js
export default {
  resolve: {
    alias: {
      vue: '@vue/compat'
    }
  },
  plugins: [
    vue({
      template: {
        compilerOptions: {
          compatConfig: {
            MODE: 2
          }
        }
      }
    })
  ]
}
```

Migration Vue2 vers Vue3

Configuration @vue/compat avec webpack

```
// webpack.config.js
module.exports = {
  resolve: {
    alias: {
      vue: '@vue/compat'
    }
  },
  module: {
    rules: [
      {
        test: /\.vue$/,
        loader: 'vue-loader',
        options: {
          compilerOptions: {
            compatConfig: {
              MODE: 2
            }
          }
        }
      }
    ]
  }
}
```

Migration Vue2 vers Vue3

Installation @vue/compat

Configurer le mode compatibilité

- Permet de filtrer les features à utiliser en VUE 2
- Utilisation de `compatConfig`

```
app.config.compatConfig = {
  MODE: 3, // Utiliser les fonctionnalités Vue 3 par défaut
  // Ajuster les paramètres de compatibilité
  ATTR_FALSE_VALUE: false,
  COMPONENT_FUNCTIONAL: false,
  // Autres options de compatibilité...
}
```

Migration Vue2 vers Vue3

Détecter des incompatibilités

Outils de détection intégrés

- Messages de console pour les fonctionnalités non compatibles.
- Utilisation de `app.config.warnHandler` pour gérer les avertissements.

```
app.config.warnHandler = function (msg, vm, trace) {
  console.warn(`[Vue warn]: ${msg} - ${trace}`)
}
```

Migration Vue2 vers Vue3

Migration des plugins

Mise à jour des plugins tiers

- Vérifier la documentation des plugins pour les mises à jour de compatibilité Vue 3.
- Exemple avec Vue Router

```
import { createRouter, createWebHistory } from 'vue-router'

const router = createRouter({
  history: createWebHistory(),
  routes: [
    // Définir les routes ici...
  ]
})
```

Migration Vue2 vers Vue3

Bonne pratiques de migration

Stratégies de migration

- Migrer les composants et fonctionnalités critiques en premier.
- Tester régulièrement pour identifier les problèmes.
- Documenter les changements pour les autres membres de l'équipe.

Migration Vue2 vers Vue3

Ressources supplémentaires

- Migration Guide
- Breaking changes

Travaux pratiques

Migration Vue project

Tests

Tests

Pourquoi tester ?

L'écriture de tests sur une application vous permet de :

- d'éviter la régression
- d'augmenter notre confiance dans ce que nous livrons en production
- d'encourager les développeurs à mieux organiser le code en modules faciles à tester

Tests

Types de tests

- **Unitaires** : vérifient que les entrées d'une fonction, classe ou composant produisent le résultat attendu.
- **Composant** : vérifient qu'un composant se monte, s'affiche et se comporte comme prévu.
- **e2e** (**end to end** ou **bout en bout**) : vérifient un fonctionnalité de l'application dans son ensemble. Cela implique généralement la navigation entre les pages de l'application et/ou l'interaction avec une API.

Test unitaire

- Jest, si le projet a été créé avec Vue-CLI

- Vitest, si le projet a été créé avec Vite

Test unitaire

Exemple

```
import { toggleFavorite } from './toggle-favorite.js'

describe('toggleFavorite', () => {
  it('toggles favorite status', () => {
    const show = { user: { favorited: true } }

    toggleFavorite(show)
    expect(show.user.favorited).toBe(false)

    toggleFavorite(show)
    expect(show.user.favorited).toBe(true)
  })

  it('does not throw error if show is null', () => {
    expect(() => toggleFavorite(null)).not.toThrow()
    expect(() => toggleFavorite(undefined)).not.toThrow()
  })
})
```

Test composant

Comme pour les tests unitaires, utilisez Jest (pour Vue CLI) ou Vitest (pour Vite).

Ces tests impliquent souvent de monter le composant testé de manière isolée. Il faut donc une **librairie de montage** et il en existe **deux très populaires** :

- Vue Testing Library Des utilitaires de tests simples et complets encourageant les bonnes pratiques de test
- Vue Test Utils (bas niveau, utilisé par vue-testing-library)
  Comme cette bibliothèque expose certaines API de bas niveau, les tests peuvent être fragiles. **Attention à ne tester uniquement** ce que fait un composant, et non comment il le fait.

Test composant

Exemple

Voici comment mettre en œuvre un test avec ces deux librairies pour cet exemple simple.

```
import { ref } from 'vue'
const props = defineProps(['title'])
const counter = ref(0)

function increment () {
  counter.value++
}

function decrement () {
  counter.value--
}
```

```
<div class="container">
  <p>{{ title }} : {{ counter }}</p>
  <button id="decrement" @click="decrement" class="button">-</button>
  <button id="increment" @click="increment" class="button">+</button>
</div>
```

Test composant

vue-testing-library

```
import { render, fireEvent } from '@testing-library/vue'
import Counter from '@/components/Counter.vue'

describe('Counter', () => {
  it('should increment the counter', async () => {
    // The render method returns a collection of utilities to query your component.
    const { getByText } = render(Counter, {
      props: { title: 'My counter', }
    })

    // getByText returns the first matching node for the provided text, and
    // throws an error if no elements match or if more than one match is found.
    getByText('My counter : 0')

    const incrementBtn = getByText('+')
    await fireEvent.click(incrementBtn)

    getByText('My counter : 1')
  })
})
```

Check the docs here

Test E2E

cypress

Cypress est un framework de test tout-en-un, il contient des librairies pour les assertions, avec des fonctionnalités de mock etc...

Consultez la Documentation

Cypress s'exécute aussi vite que le navigateur peut rendre du contenu.
On peut observer l'exécution des tests en temps réel pendant le développement.

```
describe('ShowList', () => {
  it('renders the list of shows', () => {
    cy.visit('/')
    cy.get('h1').contains('My TV shows')
    cy.get('[data-cy=card-show]').should('have.length', 9)
  })
})
```

Travaux pratiques

Lab 8

TypeScript

TypeScript

Typescript est un sur-ensemble de JavaScript qui compile en JS pur.

- Grâce au typage, TypeScript lève plusieurs erreurs au travers de l'analyse statique du code à la compilation.
- Réduit considérablement le risque d'erreurs à l'exécution
- Permet de refactoriser son code avec plus d'aisance sur de grosse applications.
- TypeScript améliore grandement l'expérience développeur grâce (entre autres) à l'autocomplétion basé sur le typage dans un IDE.

TypeScript

Installation

Vue est écrit en TypeScript ce qui offre un support de première classe.
Lors de lancement de la commande `npm init vue`, sélectionner Typescript pour que le projet se configure correctement.
Enfin, ajouter `lang="ts"` dans la balise `script` de son composant le rend compatible avec TS.

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

const title = ref('Hello world') // Type will be Ref<string>

const array = ref(['Paul', 0]) // Type will be Ref<(string|number)[]>

onMounted(() => {
  title.value = 0 // This will throw a type error
})
</ script>
```

Voir la documentation officielle pour plus de détails Typescript Support.

TypeScript

Props

La macro defineProps() donne le type de son argument:

```
const props = defineProps({
  label: { type: String, required: true },
  counter: Number
})

props.label // string
props.counter // number | undefined
```

Si l'on a besoin de types complexes, on utilisera le mot clé as :

```
interface Show {
  title: string
  seasons: number
}
const props = defineProps({
  show: { type: Object as () => Show, required: true
},
  shows: { type: Array as () => Show[], default: [] },
})

props.show.title // string

props.shows.map((show) => {
  show.title // string
});
```

Props

Plus communément, on définit les props avec un typage pur via un argument de type générique.

Pour donner une valeur par défaut avec cette syntaxe, on utilisera la macro withDefaults.

```
interface Show {
  title: string
  seasons: number
}

const props = withDefaults(defineProps<{ show: Show; shows: Show[]}>(), {
  shows: () => []
})
```

TypeScript

Emit

La fonction `emit` peut être typée grâce à la déclaration `type`.

```ts
const emit = defineEmits<{
  (e: 'update-show', id: number): void
  (e: 'toggle-favorite'): void
}>()

emit('update-show', 6) // OK
emit('toggle-favorite') // OK

emit('toggle-favorite', 'test') // Error because a payload is not allowed
emit('toggle') // Error because the event does not exist
```

TypeScript

Ref

ref() déduit le type à partir de la valeur initiale.

```
import { ref } from 'vue'

// inferred type: Ref<number>
const counter = ref(0)
```

Pour des types plus complexes, on peut soit passer un générique à la méthode `ref()` ou utiliser le type `Ref`.

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

interface Show {
  title: string;
}

const gameOfThrones = ref<Show>({ title: 'game of thrones'})

const breakingBad: Ref<Show> = ref({ title: 'breaking bad'})
```

TypeScript

Reactive

reactive() déduit de manière implicite les types de son argument.

Pour des types plus complexes, on utilise une interface.

```
import { reactive } from 'vue'

const fruits = reactive(['apple', 'strawberry']) // string[]

interface Show {
  title: string;
}
const show: Show = reactive({ title: 'the walking dead' })
```

TypeScript

Computed

computed() déduit implicitement son type par la valeur de retour.

```
import { computed, ref } from 'vue'

const counter = ref(1)
const doubleCounter = computed(() => counter.value * 2) // infers number
```

Pour spécifier un type explicitement, on utilise un générique

```
import { computed } from 'vue'

interface User {
  firstName: string;
  lastName?: string;
}

const user = computed<User>(() => ({ firstName: 'James' }))
```

Travaux pratiques

Lab 9

QCM

QCM

Avec l'API option, comment définit-on la **donnée** d'un composant ?

- `data: { field: value }`
- `data: () => ({ field: value })`
- `data: function() { return { field: value } }`
- `data() { return { field: value } }`

QCM

Laquelle de ces phrases est fausse ?

- L’option **computed** ne peut pas utiliser de la donnée, des props ou d’autres computed
- L’option **computed** n’est disponible qu’à l'intérieur du **template**
- L’option **computed** est recalculée seulement quand une de ses dépendance est mise à jour
- L’option **computed** est une fonction dont le résultat est mis en cache jusqu’au prochain appel de la fonction

QCM

Quelle phase du cycle de vie d’un composant n’existe pas dans VUE ?

- `mounted`
- `beforeUnmount`
- `shouldComponentUpdated`
- `beforeCreated`
- `updated`

QCM

Vrai ou faux, un composant Vue peut être utilisé comme une balise HTML avec des attributs et des enfants ?

QCM

Laquelle de ces affirmations est fausse ?

- Un **plugin** peut enregistrer un ou plusieurs composants de manière globale.
- Un **plugin** peut créer une nouvelle étape dans le cycle de vie.
- Un **plugin** peut créer des nouvelles méthodes dans chaque instance.
- Un **plugin** ne peut être initialisé plusieurs fois, une fois suffit.

QCM

Laquelle des propositions suivantes est le gestionnaire de store officiel de Vue3 ?

- Pinia
- Vuex
- Redux
- NgRx
- Recoil
- XState

QCM

Comment mettre à jour **l'état d'un store** ?

- Seulement au travers d'actions dans ce même store.
- N'importe où, à condition d'avoir une référence de l'état du store.
- Seulement à l'intérieur des méthodes d'un composant.
- On ne peut pas mettre à jour l'état d'un store, il faut le remplacer entièrement.

QCM

Vrai ou Faux, pinia est supporté par vuedevtools ?

QCM

Vrai ou Faux, si `onMounted` est défini comme `async` cela va bloquer le rendu ?

QCM

Laquelle de ces affirmations est fausse ?

- Il est possible d'accéder à un **slots** dans l'API option avec `this.$slots`.
- Si l'on définit une **props** comme une `string` required mais qu'à l'exécution un `number est passé en paramètre, Vue va lever une erreur.
- Si l'on définit une **computed** avec le même nom qu'une **props**, alors Vue va crasher à la compilation.
- Si l'on définit une **donnée** initialisée avec une **props**, alors la **donnée** et la **props** seront égales seulement au moment de l'initialisation du composant.

QCM

Quand utiliser `<router-link />` dans un composant ?

- Jamais, la balise HTML `<a/>` est suffisante.
- Seulement pour les liens internes d'une SPA.
- Pour tous liens, la balise HTML `<a/>` n'est pas correctement interprétée dans Vue