VUEJS

Ce cours se base en une grande partie sur la documentation officielle de vuejs : https://vuejs.org/

Introduction

- Framework javascript pour créer des interfaces utilisateur
  - Il s'appuie sur les standards HTM, CSS et le langage Javascript
  - Il fournit un modèle de programmation déclaratif et basé sur les composants => il simplifie le développement des interfaces utilisateur
- Deux fonctionnalités principales de Vue :
  - **Rendu déclaratif** : Vue étend le code HTML standard avec une syntaxe de template (modèle) qui nous permet de décrire de manière déclarative la sortie HTML en fonction de l'état JavaScript
  - **Réactivité** : Vue suit automatiquement les changements d'état de JavaScript et met efficacement à jour le DOM lorsque des changements se produisent.

Introduction (suite)

Exemple :

```
<script>
  export default {
    data() {
      return {
        count: 0
      }
    }
  }
</script>

<template>
  <button @click="count++">Count is: {{ count }}</button>
</template>

<style scoped>
  button {
    font-weight: bold;
  }
</style>
```

Styles d'API

- Vue propose deux styles d'API :
  - API d'options : la logique d'un composant est définie à l'aide d'un objet d'options telles que des données (data), des méthodes et monté (mounted). Les propriétés définies par les options sont exposées via this à l'intérieur de ces fonctions, qui pointe vers l'instance du composant
  - API de composition : la logique d'un composant est définie à l'aide de fonctions d'API importées. Dans les SFC, l'API de composition est généralement utilisée avec <script setup>. L'attribut setup est un indice qui permet à Vue d'effectuer des transformations au moment de la compilation qui nous permettent d'utiliser l'API Composition avec moins de temps. Par exemple, es importations et les variables/fonctions de niveau supérieur déclarées dans <script setup> sont directement utilisables dans le template (modèle).

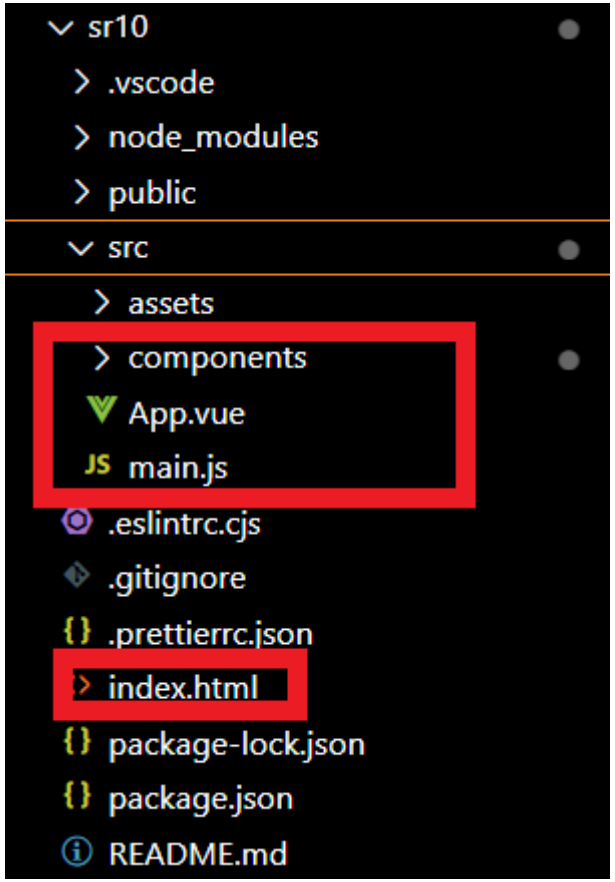
- Deux modes :
  - Single-File Components
  - HTML sans étape de construction

Un projet VUEJS mode SFC

- une configuration de construction basée sur Vite
  - > npm init vue@latest
    - Configuration via le choix de certaines options (nom de projet, typescript, etc.)
  - > cd <your-project-name>
  - > npm install
  - > npm run dev

- Construire votre application (production) :
  - > npm run build : un dossier ./dist sera crée pour contenir le résultat de cette étape

Structure de projet



- **./components/** : vos composants
- **App.vue** : la page entière de votre application
- **Main.js**

```
import './assets/main.css'
import   createApp   from 'vue'
import App from './App.vue'
createApp App  mount '#app'
```

- **Index.html**

```
<!DOCTYPE html>
<html lang "en">
  <head>
    <meta charset "UTF-8">
    <title>    </title>
  </head>
  <body>
    <div id "app"></div>
    <script type "module"src "/src/main.js"></script>
  </body>
</html>
```

Mode HTML

- CDN : <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>

```
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>

<div id="app">{{ message }}</div>

<script>
  const { createApp } = Vue

  createApp({
    data() {
      return {
        message: 'Hello Vue!'
      }
    }
  }).mount('#app')
</script>
```

Comosant vue.js

- un composant de fichier unique Vue (SFC) :
  - contient de HTML, CSS et Javascript
  - est sauvegardé dans un fichier d'extension *.vue
  - est un bloc indépendant et réutilisable

```
<script>
export default {
    data() {
        return {
            message: 'Hello World!',
        }
    }
}
</script>

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

Rendu déclaratif

- La fonctionnalité principale de Vue est le rendu déclaratif : en utilisant une syntaxe de modèle (templète) qui étend le HTML, nous pouvons décrire à quoi le HTML devrait ressembler en fonction de l'état de JavaScript. Lorsque l'état change, le HTML se met à jour automatiquement

L'état

- Peut déclencher des mises à jour lorsqu'il est modifié est considéré comme réactif.
- Dans Vue, l'état réactif est maintenu dans les composants.
- Nous pouvons déclarer l'état réactif en utilisant l'option de composant de données (data), qui devrait être une fonction qui renvoie un objet.

```
<script>
export default {
  data() {
    return {
      message: 'Hello World!',
      counter: {
        count: 0
      }
    }
  }
}
</script>

<template>
  <h1>{{ message }}</h1>
  <p>Count is: {{ counter.count }}</p>
</template>
```

Insérer de texte

- La valeur d'une propriété peut être insérée en utilisant {{la syntaxe mustaches}} dans la partie template :
  - <h1>{{ message }}</h1>
- On peut afficher n'importe expression javascript
  - <h1>{{ message.split('').reverse().join('') }}</h1>

Insérer un attribut

La syntaxe {{mustache}} ne permet pas d'insérer des attributs

- Pour le faire on utilise la directive v-bind
  -
  - syntaxe abrégée :
    - <div :id="dynamicId">

```
<script>
export default {
    data() {
        return {
            titleClass: 'title'
        }
    }
}
</script>

<template>
    <h1 :class="titleClass">Make me red</h1>
</template>

<style>
.title {
    color: red;
}
</style>
```

Intercepter des évènements

- La directive v-on : <button v-on:click="increment">{{ count }}</button>
- Syntaxe abrégée : <button @click="increment">{{ count }}</button>
- Accès aux propriétés de composants via this : this.count

```
<script>
export default {
  data() {
    return {
      count: 0
    }
  },
  methods: {
    increment() {
      this.count++
    }
  }
}
</script>

<template>
  <button @click="increment">count is: {{ count
}}</button>
</template>
```

Formulaire

- Utilisation de v-bind et v-on ensemble pour associer la valeur de input à une propiété (ici « text ») :
  - <input :value="text" @input="onInput">

```
<script>
export default {
    data() {
        return {
            text: ''
        }
    },
    methods: {
        onInput(e) {
            // a v-on handler receives the
native DOM event
            // as the argument.
            this.text = e.target.value
        }
    }
}
</script>
```

Formulaire

- Utilisation de v-model

```
<script>
export default {
    data() {
        return {
            text: ''
        }
    }
}
</script>

<template>
    <input v-model="text" placeholder="Type
here">
    <p>{{ text }}</p>
</template>
```

Rendu conditionnel

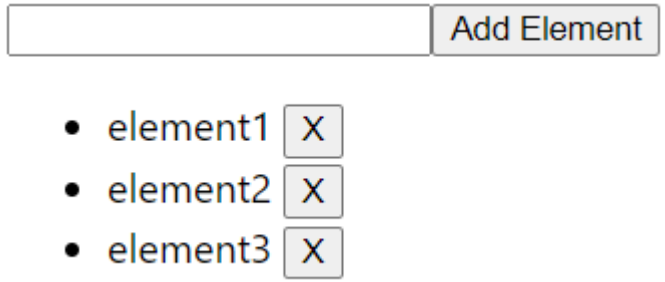
- <h1 v-if="awesome">Vue is awesome!</h1>
  - Si la valeur de la propriété « awesome » est vraie alors cette balise h1 sera intégrée au DOM. Si elle devient fausse elle sera enlevé de DOM.
- On peut ajouter le « else » comme l'exemple

```
<script>
export default {
    data() {
        return {
            awesome: true
        }
    },
    methods: {
        toggle() {
            this.awesome = !this.awesome
        }
    }
}
</script>

<template>
    <button @click="toggle">toggle</button>
    <h1 v-if="awesome">Vue is awesome!</h1>
    <h1 v-else>Oh no 😢</h1>
</template>
```

Afficher l'éléments d'un tableau
la directive v-for


```
<script>
// give each element a unique id
let id = 0

export default {
  data() {
    return {
      newElement: '',
      elements: [
        { id: id++, text: 'element1' },
        { id: id++, text: 'element2' },
        { id: id++, text: 'element3' }
      ]
    }
  },
  methods: {
    addElement() {
      this.elements.push({ id: id++, text: this.newElement })
      this.newElement = ''
    },
    removeElement(element) {
      this.elements = this.elements.filter((t) => t !== element)
    }
  }
}
</script>
```

```
<template>
  <form @submit.prevent="addElement">
    <input v-model="newElement">
    <button>Add Element</button>
  </form>
  <ul>
    <li v-for="element in elements" :key="element.id">
      {{ element.text }}
      <button @click="removeElement(element)">X</button>
    </li>
  </ul>
</template>
```



Propriété calculée

- déclarer une propriété qui est calculée de manière réactive à partir d'autres propriétés en utilisant l’option « computed »

Cycle de vie et Refs de template

- Ref template : une référence à un élément du template en utilisant l'attribut spécial ref
  - hello
  - L'élément sera exposé via this.$refs en tant que this.$refs.p
- Vous ne pouvez y accéder qu'après le montage du composant
  - Pour exécuter le code après le montage, nous pouvons utiliser l'option de cycle de vie : mount.
  - Autres options : created et updated

```
<script>
export default {
    mounted() {
        this.$refs.p.textContent = 'mounted!'
    }
}
</script>

<template>
    <p ref="p">hello</p>
</template>
```

Watchers (observateur)

- Observateur : faire des traitements quand une propriété change de valeur

```
export default {
    data() {
      return {
        count: 0
      }
    },
    watch: {
      count(newCount) {
        // yes, console.log() is a side effect
        console.log(`new count is: ${newCount}`)
      }
    }
  }
```

Composants

> App.vue

```
<script>
import ChildComp from './ChildComp.vue'

export default {
    components: {
        ChildComp
    }
}
</script>

<template>
    <ChildComp />
</template>
```

> ChildComp.vue

```
<template>
    <h2>A Child Component!</h2>
</template>
```

Les props

- Entrées acceptées par un composant :

```
export default {
  props: {
    msg: String
  }
}
```

- <ChildComp :msg="greeting" />

Emits

- En plus de recevoir des entrées, un composant enfant peut également émettre des événements vers le parent :

Parent

```
<script>
import ChildComp from './ChildComp.vue'

export default {
    components: {
        ChildComp
    },
    data() {
        return {
            childMsg: 'No child msg yet'
        }
    }
}
</script>

<template>
    <ChildComp @response="(msg) => childMsg = msg" />
    <p>{{ childMsg }}</p>
</template>
```

Enfant

```
<script>
export default {
    emits: ['response'],
    created() {
        this.$emit('response', 'hello from
child')
    }
}
</script>

<template>
    <h2>Child component</h2>
</template>
```

Fragment (Slot)

- Le composant parent peut transmettre des fragments de modèle à l'enfant via des slots :
  - <ChildComp> This is some slot content! </ChildComp>
- Dans le template enfant on utilise <slot /> pour récupérer le fragment transmis:
  - <!-- in child template -->

    <slot/>
  - Pour afficher un message dans le cas où le fragment n'est pas passé par le parent : <slot>Fallback content</slot>