



Formation Vue.js

Maîtriser le framework
JavaScript Open Source

Fabien Grignoux

fabien.grignoux@outlook.com
<https://developpeurfullstack.fr>

Sommaire

- Présentation de Vue.js - p.5
- Ecosystème du développeur JS moderne - p.11
- Typescript - p.21
- Premiers pas avec Vue.js - p.25
- Création d'une SPA - p.29
- Les composants - Option Api - p.33
- Les composants - Composition Api - p.51
- Le templating - p.61
- Les APIs - p.65
- Le routing - p.73
- Les stores - p.83
- Les animations - p.102
- Les tests unitaires - p.107
- Les test end to end - p.116
- Aller plus loin - p.121

Objectifs

- Comprendre les concepts clés de Vue.js
- Être capable de développer une application Single Page App complète avec Vue.js
- Connaître les outils indispensables au développement d'applications Vue.js

Prérequis

Connaissances :

- Les bases HTML & CSS
- Les bases de Javascript
- Des bases de Node.js (ça peut aider mais c'est pas essentiel)

Environnement de développement :

- IDE : Visual Studio Code
- VSCode plugins : Volar, Typescript Vue Plugin (Volar), Prettier, Vue VS Code Snippets, Live Share, Vue 3 snippets, REST Client
- Node.js version LTS



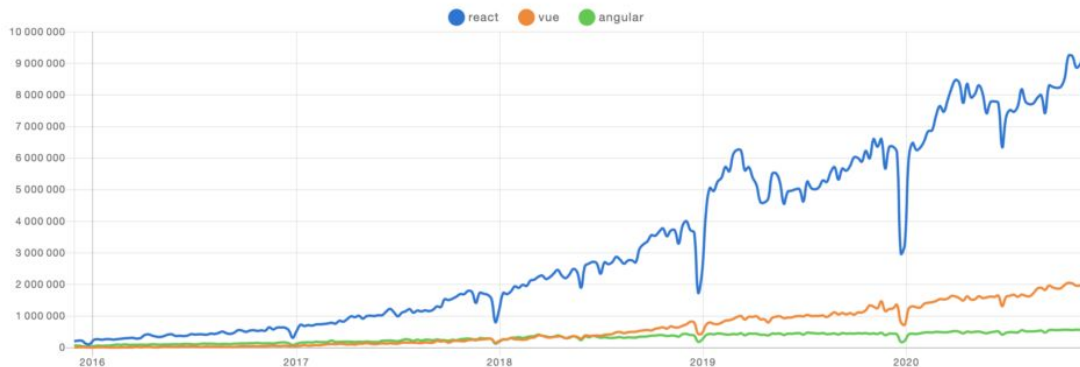
Présentation de Vue.js

Vue.js Présentation

Vue.js a été conçu par Evan You un ancien employé de Google, puis de Meteor. De projet secondaire, Il se consacre maintenant entièrement à Vue. Je vous recommande le reportage sur Vue.js : <https://www.youtube.com/watch?v=OrxmtDw4pVI&t=1s>

Vue est un framework Javascript. C'est le dernier arrivé (2014) dans le trio Angular / React / Vue.

Evolution des téléchargement via npm



A quoi sert Vue.js ?

Vue.js est un framework front-end, c'est à dire qu'il s'exécute sur le poste du client. Il permet de créer des application complètes en Javascript. Les applications sont beaucoup plus réactives qu'avec une architecture "LAMP" classique ou tout le calcul et le rendu est réalisé côté serveur.

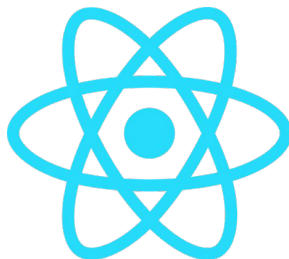
La logique de l'application est maintenant double : Il va falloir un serveur backend pour gérer nos bases de données et distribuer des contenus au travers d'APIs et une application front-end Comme Vue pour exploiter ces données.

La logique applicative se déporte aujourd'hui vers le navigateur. Le métier de développeur front-end, considéré comme plus facile que celui de développeur back-end se complexifie. Désormais il se dit "Front-end is the new Back-end".

Comparaison des frameworks



Angular



React



Vue.js

- Développé par Google
- OpenSource
- Succède à Angular.js
- Angular.js date de 2010
- Angular date de 2016
- En septembre 2023, 85 000 Github stars
- Utilise Typescript
- Utilise un moteur de template
- 2 ème en terme d'utilisation

- Développé par Facebook
- OpenSource
- date de sortie 2013
- En septembre 2023, 213 000 Github stars
- Utilise JSX
- Mélange code, style et logique
- Typescript aisé
- Domine le marché

- Développé par Evan You
- OpenSource
- Date de 2014
- En septembre 2023, 245 000 Github stars
- Utilise un moteur de template
- Typescript difficile à mettre en place en V2 mais possible
- Actuel numéro 3 en utilisation

et au niveau des performances ? 🖱️ <https://blog.vuejs.org/posts/vue-3-2>

React JSX vs Vue template

Résultat à obtenir :

```
<ul>
  <li> John </li>
  <li> Sarah </li>
  <li> Kevin </li>
  <li> Alice </li>
</ul>
```

React JSX :

```
const names = ['John', 'Sarah', 'Kevin', 'Alice'];

const displayNewHires = (
  <ul>
    {names.map(name => <li>{name}</li> )}
  </ul>
);

ReactDOM.render(
  displayNewHires,
  document.getElementById('root')
);
```

Vue template

Résultat à obtenir :

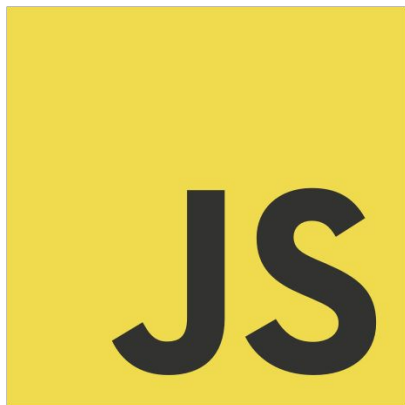
```
<ul>
  <li> John </li>
  <li> Sarah </li>
  <li> Kevin </li>
  <li> Alice </li>
</ul>
```

Vue Javascript

```
const listOfNames = ['Kevin', 'John', 'Sarah', 'Alice']
```

Vue HTML

```
<div>
  <ul>
    <li v-for='name in listOfNames'>
      {{name}}
    </li>
  </ul>
</div>
```



L'écosystème du
développeur JS
moderne

ECMAScript

- ECMAScript est un ensemble de normes qui sont mises oeuvre dans le langage Javascript
- La version d'ES6 (2015) a été définie comme majeure. Ce cours utilise cette syntaxe
- La version actuelle est ES10(2019) et la prochaine version sera ES11 (2020). Ce sont des versions mineures
- ES11 apportera des fonctions d'aides sur les entiers de grande taille, les imports dynamiques, les nullish coalecents (??) et les optional chaining opération (?.)

Fonctions principales d'ECMAScript 6 et plus

- Déclaration des variables avec “let” et “const” pour éviter d'utiliser le “var”. Let permet de limiter sa portée dans son bloc { }. Const ajoute les constantes.

Avant ES6 - Avec Var

```
function varTest() {  
  var x = 1;  
  if (true) {  
    var x = 2; // même variable!  
    console.log(x); // 2  
  }  
  console.log(x); // 2  
}
```

Après ES6 - Avec let

```
function letTest() {  
  let x = 1;  
  if (true) {  
    let x = 2; // variable différente  
    console.log(x); // 2  
  }  
  console.log(x); // 1  
}
```

- Les templates literals pour créer des concaténation plus proprement en utilisant `\${ maVariable}`

```
//Version ES5  
var prenom = "Fabien", nom= "Grignoux";  
console.log("je suis "+prenom+" "+nom);
```

```
//Version ES6  
let prenom = "Fabien", nom= "Grignoux";  
console.log(`je suis ${prenom} ${nom}`);
```

- Les fonctions fléchées

```
//Version ES5  
function calcul(a,b){  
    return a+b  
}  
console.log(calcul(5,3))
```

```
//Version ES6  
const calcul=(a,b)=>a+b  
console.log(calcul(5,3))
```

- Tableaux

```
const tableau=["a", "b", "c", "d"]

//ajout à la fin
tableau.push("e")

//ajout au début
tableau.unshift("z")

//suppression de la chaine "d"
tableau = tableau.filter(element=>element!="d")

ou
const index = tableau.indexOf("d")
tableau.splice(index,1)
```


- Le forEach et le map

```
let tableau = ["a", "b", "c", "d"];

for (let i = 0; i < tableau.length; i++) {
  var element = tableau[i];
  console.log(element);
}

// forEach -> opération sur le tableau (peut modifier le tableau d'origine)
tableau.forEach(function (element) {
  console.log(element);
});

// Equivalent à
tableau.forEach(element=>console.log(element));

// map -> opération sur le tableau (ne modifie pas le tableau original)
const nouveauTableau=tableau.map(element=>`lettre ${element}`)
console.log(nouveauTableau);
```

Mais aussi :

- Les fonctions asynchrones simplifiées avec `async/data`
- L'object destructuring
- Les classes et objets (sucre syntaxique)
- ...

Les nouveautés ES9/ES10

- Les spreads operators "...": sélectionne chaque élément d'un tableau ou objet

```
const monTableau1=[1,2,3];  
const monTableau2=[...monTableau1,4,5,6];  
console.log(monTableau2) //1 2 3 4 5 6
```

- Promise.finally : Dans l'utilisation des promesses, exécute à la fin un code qui

```
fetch(url)  
  .then()  
  .catch()  
  .finally(() => console.log(`Je serai toujours appelé :)`));
```

- trimStart et trimEnd : suppriment les espace blancs

```
const maChaine="  Bonjour Tout le monde !  ";  
console.log(maChaine.trimStart().trimEnd());  
//Bonjour tout le monde
```

- Array.flat : aplatit un tableau

```
let monTableau=[1,2,3,4,[6,7,8]];  
console.log(monTableau.flat());  
// 1 2 3 4 5 6 7 8
```



Typescript

Typescript

Typescript est un superset de Javascript développé par Microsoft et open source. Il permet de profiter de fonctionnalités supplémentaires en Javascript comme le typage, des fonctionnalités objet avancé comme les interfaces.

Le code Typescript est compilé en Javascript par webpack. Ce langage est de plus en plus populaire et atteint la troisième place de langages les plus appréciés derrière Rust et Clojure dans la dernière étude Stack Overflow de 2021.

<https://insights.stackoverflow.com/survey/2021>

Il est possible d'utiliser Typescript avec Vue.js même s'il n'est vraiment facile d'utilisation qu'à partir de Vue 3 grâce à sa composition API.

Avantages / inconvénients de TS

Avantages

- + Code plus clair
- + Autocomplétion dans l'IDE
- + Moins de bugs
- + Plus facile pour les projets à plusieurs développeurs

Inconvénients

- Nécessite un travail de configuration
- Plus complexe à apprendre que le Javascript
- Plus verbeux que JS

Typing son code JS

```
const maVar="Hello" //Type string (inférence de type)
```

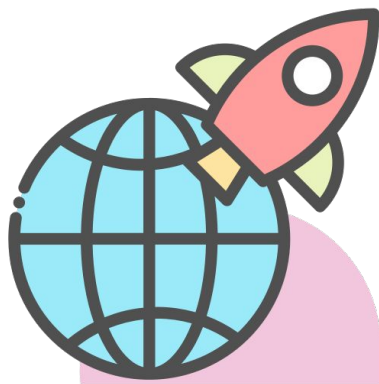
```
type lang="fr"|"en"|"de" //Création d'un type personnalisé de 3 valeurs possibles (custom type)  
const maLangue:lang = "ru" //erreur car pas dans le type personnalisé
```

```
interface IPersonne{  
  id?:string  
  nom:string;  
  age:number;  
  isDriver:boolean  
}  
const michel:IPersonne={nom:"Michel",age:50,isDriver:false}
```

```
const getPersonneFromApi(id:string):Promise<IPersonne>{  
  ...  
}
```

Utilisation d'une interface pour décrire l'objet attendu.
Le ? après id rend l'id non obligatoire.

Création d'une fonction qui retourne une promesse qui contiendra un objet de type personne



Premiers pas avec Vue.js

Vue.js version 3

Elle est sortie fin 2020. Version 3.2 majeure en aout 2021. Février 2022 la version 3 de Vue est maintenant la version officielle.

Elle apporte:

- Le support natif de Typescript (comme Angular) mais sera optionnel
- Des performances améliorées
- Une nouvelle façon de faire des composants plus lisible (mais qui ne supprime pas l'actuelle) : La Composition API
- En version 3.2 elle apporte aussi la Setup API : une amélioration plus simple de composition API
- Abandon de IE 11

Principes de Vue.js

- Le coeur de Vue.js est volontairement très simple. Il peut être utilisé en tant que librairie dans un projet existant.
- Vue est extensible grâce à des bibliothèque complémentaires qui lui permettent de gérer par exemple le routing, les stores d'application, les requêtes HTTP ...
- Vue est basé sur un système de composant. Chaque composant peut comporter un nombre infini de composants qui eux même intègrent d'autres composants. Une page d'application est aussi un composant !
- Chaque composant est constitué d'un seul fichier .vue
- Vue n'est qu'un framework JS et n'intègre pas de librairie d'interface utilisateur. Nous sommes libre d'utiliser celle que nous souhaitons.

Vue fonctionne de 2 manières

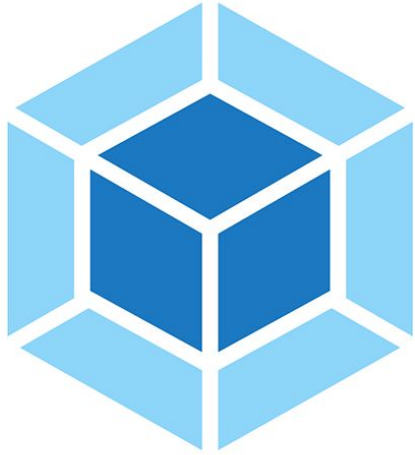
- Soit en tant que librairie externe (comme JQuery par exemple), en l'ajoutant dans un document html avec le code :

```
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
```

- Soit en utilisant Webpack pour réaliser une application complète en Vue

Même si ça n'est pas la meilleure façon de profiter de la puissance de Vue.js, nous verrons dans ce chapitre comme l'utiliser en tant que librairie pour pouvoir l'intégrer dans un projet existant.

En effet il est possible d'intégrer Vue dans une application Web existante à la manière de JQuery. Cependant pour tout nouveau projet, il est conseillé d'utiliser Webpack qui permettra une compilation des sources, l'import de composants, de meilleures performances, des mises à jour des dépendances ...



Création d'une application complète SPA Single Page Application

Commandes vue-cli & npm utiles

`npm run dev` : permet de lancer son application en mode développement

`npm install` : installe toutes les dépendances de l'application

`npm install xxxx` : Installe la dépendance xxxx

`npm update` : met à jour les dépendances de l'application

`vue-ui` : Lance l'interface graphique d'administration de vue

`npm run build` : Compile l'application pour la production dans le répertoire /dist

`npm run generate` : Compile l'application en version statique dans /dist

A noter que Yarn peut aussi être utilisé à la place de NPM si vous le souhaitez

Initialisation d'un projet Vue

- Installer grâce à Vite: `npm init vue@latest`
- Vite va déployer un projet vide Vue. Il permettra aussi de bundler nos fichiers.

Structure du répertoire :

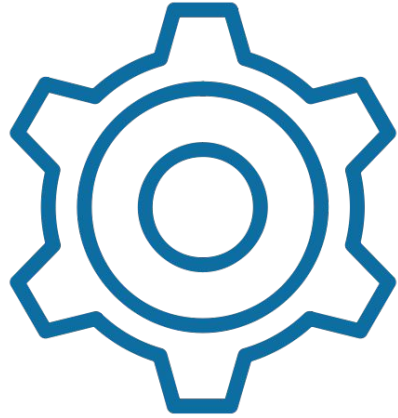
- mon-projet
 - node_modules → Paquets js de dépendance
 - public → Fichiers qui seront présents sur dans la build (ex favicons, .htaccess)
 - src → Notre répertoire de travail
 - packages.json → Gère les dépendances et la configuration du projet
 - package-lock.json
 - .gitignore → Permet d'ignorer des fichiers lors du Git

Structure du répertoire SRC

Le répertoire “src” contient la source de notre code. Nous y passerons 99% du temps à développer notre application.

- src

- assets —————> Fichiers externes : feuilles de style, images ...
- components —————> Contient tous les composants de notre application
- App.vue —————> Point d'entrée de notre application. “composant racine”
- main.js —————> Point central de notre application. Il gère :
 - l'import et la configuration des modules
 - Les variables globales
 - Définit le point de montage de l'application



Les composants : Option API

Exemple de structure d'un composant (App.vue)

```
<template>
  <div id="app">
    
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>
```

Les composants vue comportent 3 parties. La première est la partie template qui affiche les données

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

export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
</script>
<style></style>
```

La partie script de notre application contient toute la logique applicative.

- Le bloc import importe un autre composant HelloWorld
- export default contient le code de notre application. Ici il enregistre le nom de l'application, puis intègre le composant HelloWorld

La partie style contient le css de notre application.

Créer un composant “single file component”

En vue.js tout est composant ou presque ! Ils doivent se trouver dans le répertoire components, mais vous êtes libre de la structure dans ce répertoire. Voici comment créer un composant minimal. Grâce au plugin VSCode vue2-snippets il vous suffit de taper “vbase” et entrée pour générer :

Dans components/MonComposant.vue :

```
<template>
  <div>
  </div>
</template>

<script>
  export default {
  }
</script>

<style lang="scss" scoped>
</style>
```

Notes :

- Les composants portent l'extension .vue
- La partie template doit toujours contenir un “wrapper” (ici un div) qui va entourer le code du template
- Il est considéré comme une bonne pratique de nommer ses composants avec un majuscule (mais ça n'est pas obligatoire)

Ajouter un composant à un autre

Pour utiliser un composant il faut :

1. L'importer avec "import xxx from xxxx"
2. Le déclarer dans l'objet components
3. L'utiliser dans le template
`<MonComposant />` ou
`<MonComposant></MonComposant>`

Dans l'exemple présenté le "@" situé dans le chemin fait référence à la racine du répertoire "src". On peut aussi utiliser le symbole "~".

Dans App.vue :

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

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

export default {
  components: {
    HelloWorld
  },
};
</script>
```

Propriétés dynamiques

Les propriétés dynamiques font le lien entre le template et le code.

Elles peuvent être de type string, object, int, array.

Exemple :

Dans la partie script :

```
data() {  
  return {  
    personne: { prenom: "Fabien", nom: "Grignoux" }  
  }  
}
```

Dans la partie template :

```
<h1>{{ personne.prenom }}</h1>
```

```
<template>  
  <div id="app">  
    ...  
    {{ maProprieteDynamique }}  
    ...  
  </div>  
</template>  
  
<script>  
export default {  
  ...  
  data() {  
    return {  
      maProprieteDynamique : "xxx"  
    }  
  },  
}
```

Les propriétés calculées (computed)

Elles permettent de simplifier le code et d'effectuer des calculs. Elles permettent ainsi de décharger la partie template de codes complexes qui n'ont pas leur place ici.

Ces fonctions sont initialisées dans l'objet `computed` et appelées directement dans les templates.

```
export default {  
  ...  
  computed: {  
    nomComplet() {  
      return this.personne.prenom + " " + this.personne.nom;  
    }  
  },  
  data() {  
    return {  
      personne: { prenom: "Fabien", nom: "Grignoux" }  
    };  
  }  
};
```

```
<h1>{{ nomComplet }}</h1>
```

Cycle de vie d'un composant

Des opérations et calculs peuvent être réalisés à différents moments de la vie d'un composant :

1. `beforeCreate` : Initialisation des événements de cycle de vie
2. `created` : Initialisation des injections & de la réactivité. Le DOM n'est pas disponible mais les données si.
3. `beforeMount` : Les templates sont compilés
4. `mounted` : Le composant est monté et le DOM disponible. Le composant est complètement disponible.
5. `beforeUpdate` : Lancé au changement de données
6. `updated` : Le rendu du DOM a été effectué après changement des données
7. `beforeUnmount` : appelé juste avant qu'un composant soit démonté
8. `unMounted` : appelé une fois le composant démonté
9. `errorTracked` : appelé quand un composant enfant reçoit une erreur
10. `renderTracked` : appelé une fois et indique quels sont les opérations trackées par le composant
11. `renderTriggered` : s'exécute à chaque fois qu'une opération trackée par le composant est réalisée
12. `activated` : utilisé quand un composant utilisant `<keep-alive>` est utilisé et activé (exemple : tabulation)
13. `deactivated` : utilisé quand un composant utilisant `<keep-alive>` est désactivé

Cycle de vie d'un composant utilisation

Des opérations et calculs peuvent être réalisés à différents moments de la vie d'un composant :

Les plus utilisés sont au montage avec “mounted”
et à la création, avant le rendu du DOM avec “created”


```
export default {  
  ...  
  created() {  
    //code à effectuer exemple : récupération de données d'une API  
  },  
};
```


Les méthodes

Les méthodes sont des fonctions qui permettent de réaliser des opérations dans notre application. Elles sont le coeur de la logique de notre application. Les méthodes sont accessibles directement dans les templates ou dans le script. Si elles sont utilisées dans le script elles doivent toujours comporter “this.” devant.

Le code suivant exécute la méthode “afficherMessage” au montage du composant :

```
export default {  
  mounted() {  
    this.afficherMessage("Bonjour!")  
  },  
  methods: {  
    afficherMessage(info) {  
      alert("Message : " + info);  
    }  
  }  
};
```



Les événements

Ils permettent de faire interagir le template avec le script de l'application. Lors de leur application, ils utilisent les fonctions décrites dans l'objet "methods". On les utilise avec le symbole @ puis le type d'événement. Le @ est un raccourci pour "v-on".

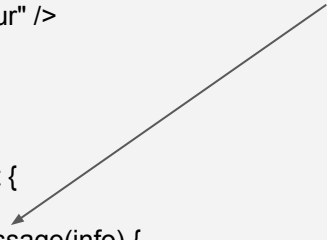
Le code de droite affiche "Bonjour" au click sur le bouton.

Voici quelques types d'événements utiles :

- click
- click.once (assure un seul click)
- mouseover
- mouseout
- change
- keyup.enter

```
<template>
  <div>
    <input type="button" @click="afficherMessage('bonjour')"
value="bonjour" />
  </div>
</template>

<script>
export default {
  methods: {
    afficherMessage(info) {
      alert("Message : " + info);
    }
  }
};
</script>
```

A diagonal arrow points from the `@click="afficherMessage('bonjour')"` attribute in the `<input>` tag of the template to the `afficherMessage` method within the `methods` object of the script.

Communication entre composants : Parent à enfant

Pour faire passer des datas d'un composant parent à un composant enfant on va utiliser les props :

Composant parent :

```
<template>
  <div>
    <HelloWorld
      :personne="personne" />
  </div>
</template>
```

Composant enfant :

```
<template>
  <div>
    Hello {{ personne.nom }}!
  </div>
</template>

<script>
  export default {
    props: {
      personne: {
        type: Object,
        default: null
      },
    },
  },
}
</script>
```

(...suite) communication parent -> enfant

Le composant parent envoie un paramètre avec l'instruction ":". C'est un raccourci à l'instruction "v-bind".

Le ":" devant le paramètre indique qu'il s'agit d'une propriété réactive.

```
<MonComposant :maData="monContenu" />
```

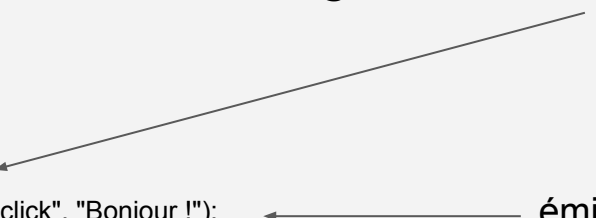
Le composant enfant récupère le paramètre envoyé avec l'objet props. Celui-ci prend en propriété le nom de la donnée envoyée et il est une bonne pratique de la spécifier en indiquant son type (String, Number, Boolean, Array, Object, Function ...) et sa valeur par défaut (0, "test", [], null ...)

Communication du composant enfant au parent

C'est un petit peu plus compliqué. Il va falloir émettre un événement au niveau du composant enfant qui sera récupéré au niveau du composant parent.

1. Emettre un événement dans le composant enfant "HelloWorld"

```
<template>
  <div>
    <input type="button" value="test événement" @click="emettreEvtAction" />
  </div>
</template>
<script>
export default {
  methods: {
    emettreEvtAction() {
      this.$emit("bouton-click", "Bonjour !");
    }
  },
  ...
};
</script>
```



émission de l'événement

The diagram illustrates the event flow. An arrow points from the `@click="emettreEvtAction"` attribute in the template to the `emettreEvtAction()` method in the script. Another arrow points from the `this.$emit("bouton-click", "Bonjour !");` line in the script to the text "émission de l'événement".

(...suite) Réception de l'événement par le parent

```
<template>
  <div>
    <HelloWorld @bouton-click="afficherMessageAction($event)" />
  </div>
</template>
```

```
<script>
...
export default {
  components: {
    HelloWorld
  },
...
  methods: {
    afficherMessageAction(info) {
      alert("Message : " + info);
    }
  }
};
</script>
```

Méthode à exécuter

Événement émis par le
composant enfant

Les mixins

Les mixins sont des bouts de code que l'on peut réutiliser dans plusieurs composants. En général quand on utilise plusieurs fois les mêmes computed properties ou les mêmes méthodes, il est probablement intéressant d'utiliser les mixins.

Il est recommandé de stocker ses mixins dans un répertoire `"/src/mixins"`.

Pour les utiliser il faut créer un mixin avec un `"export default"`.

Il faut ensuite l'importer dans les composants et ajouter un tableau mixins:[nomDuMixin]

Le code du mixin est alors importé comme une méthode ou computed property classique et est utilisable comme on le fait habituellement.

Exemple de Mixin

L'exemple suivant permet d'utiliser un mixin contenant des fonctions utilitaire et d'utiliser un méthode pour mettre un texte en majuscules.

```
//code du mixin
// ./mixins/MesFonctions.js

export default {
  methods: {
    texteEnMajuscules(texte) {
      return texte.toUpperCase()
    }
  },
}
```

```
//Code du template
<template>
  <v-container>
    <h1>{{ title }}</h1>
  </v-container>
</template>

<script>
  import MesFonctions from "@mixins/MesFonctions"

  export default {
    data() {
      return {
        title: "hello world"
      }
    },
    mounted () {
      this.title=this.texteEnMajuscules(this.title);
    },
    mixins:[MesFonctions]
  }
</script>
```


Les slots

Les slots sont une autre méthode pour communiquer du composant parent au composant enfant. C'est une manière élégante de partager des informations et rend la réutilisation des composants plus lisible.

// Exemple d'un composant Post enfant

```
<template>
  <div>
    <slot></slot>
  </div>
</template>

<script>
  export default {
  }
</script>
```

Remplit le slot

// Exemple d'un composant parent utilisant Post

```
<template>
  <Post>
    Mon contenu
  </Post>
</template>

<script>
  import Post from './components/Post.vue'

  export default {
    components: {
      Post
    }
  };
</script>
```

Les slots nommés

Pour plus de personnalisation il est possible de nommer les slots. Exemple

// Exemple d'un composant Post enfant

```
<template>
  <div>
    <header>
      <h1>
        <slot name="titre"></slot>
      </h1>
    </header>
    <p><slot></slot></p>
  </div>
</template>

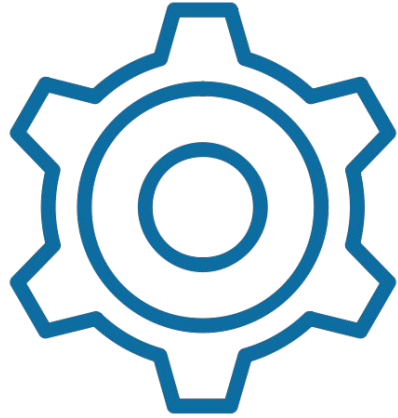
<script>
  export default {
  }
</script>
```

// Exemple d'un composant parent utilisant Post

```
<template>
  <Post>
    <template #titre>Titre de mon article</template>
  </Post>
  Mon contenu
</template>

<script>
  import Post from './components/Post.vue'

  export default {
    components: {
      Post
    }
  };
</script>
```



Les composants : Composition API

Pourquoi une autre façon de faire des composants

C'est une nouvelle manière de créer des composants

Avantages :

- Organisation indépendante de blocs “data”, “methods” ... plus lisible
- Réutilisation de code sans avoir à utiliser les mixins
- Meilleur support de Typescript

Propriétés dynamiques

```
<template>
  <div>
    <p>J'utilise {{framework}}</p>
  </div>
</template>
```

Partie de template pour l'affichage des données.
Identique à l'Option API

```
<script setup>
```

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

Tout le code est dans un unique bloc script setup. Exit les blocs datas, methods, computed, mounted, created

```
const framework=ref("vue 2")
framework.value="vue 3"
```

Création d'une propriété réactive avec la fonction ref() et une valeur par défaut. Il faut importer ref avant de l'utiliser

```
</script>
```

Pour modifier la valeur d'une propriété réactive il faut utiliser ".value"

Les méthodes

```
<template>
  <div>
    <button @click="afficheAction('Hello')">Afficher
    message</button>
  </div>
</template>
```

Partie de template pour l'affichage des données.
Identique à l'Option API

```
<script setup>
```

```
  const afficheAction=(message)=>{
    alert(`message : ${message}`)
  }
```

```
</script>
```

Création d'une fonction en JS. La fonction
est alors disponible dans le template

Les computed properties

```
<template>
  <div>
    <p>{{ presentation }}</p>
  </div>
</template>
```

Partie de template pour l'affichage des données.
Identique à l'Option API

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

import de computed pour l'utiliser par la suite

```
const person = { name: "Grignoux", firstName: "Fabien" };
```

```
const presentation = computed(
  () => `Je m'appelle ${person.firstName} ${person.name}`
);
</script>
```

Utilise computed qui prend en paramètre la fonction que l'on souhaite être "computable"

Communication parent->enfant (props)

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

```
<script setup>
```

```
const props = defineProps({myText:string,myTitle:string});
console.log(props.myText)
```

```
const { myText } =defineProps({myText:{required:true, default:'Hello'},myTitle:string});
```

```
//Nouveauté vue 3.3 en Typescript
```

```
const {maProp='Hello', monBool=true} = defineProps<maProp?:string, monBool:boolean>();
</script>
```

A partir de vue 3.3
l'object destructuring ne
casse pas la réactivité !

Fonctionnement en mode
Typescript et avec des
valeurs par défaut.

Le “?” indique un
paramètre facultatif

Emettre un événement personnalisé (communication enfant parent)

Composant enfant "HelloWorld.vue"

```
<template>
  <h1>
    <button @click="sendMessageAction('hello')>Envoi d'un paramètre</button>
  </h1>
</template>
```

Déclaration des événements émettables par notre composant

```
<script setup>
const emit = defineEmits(["send-message", "send-warning"]);
const sendMessageAction = (message) => emit("send-message", message);
```

Emission de notre événement avec un paramètre à envoyer

```
// Définition d'emit avec Typescript (vue >= 3.3)
const emit = defineEmits<{
  send-message: [message: string];
  send-warning: [warning: Warning];
}>();
</script>
```

(...suite) Réception de l'événement par le parent

```
<template>
  <div>
    <HelloWorld @send-message="afficherMessageAction($event)" />
  </div>
</template>

<script setup>
import HelloWorld from "@src/components/HelloWorld"
const afficherMessageAction=(message)=>console.log(message)
</script>
```

Méthode à exécuter

Événement émis par le
composant enfant

Les fonctions composables ≈ Mixin

Les mixins n'existent pas en composition API et c'est une bonne chose! Il est possible d'obtenir un fonctionnement similaire en utilisant des fonctions "composables"

```
//code de la fonction composable (mixin)  
// @/composables/useCounter.js
```

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

```
export default function() {  
  const counter = ref(0);
```

```
  const incrementAction = () => counter.value++;
```

```
  return { incrementAction, counter };  
}
```

import du fichier

import de variables ou fonctions

```
// Composant
```

```
<template>
```

```
<h1>{{ counter }}</h1>
```

```
<button @click="incrementAction">Increment</button>  
</template>
```

```
<script setup>
```

```
  import useCounter from "@/composables/useCounter";
```

```
  const { counter, incrementAction } = useCounter();
```

```
</script>
```

Cycle de vie d'un composant

Des opérations et calculs peuvent être réalisés à différents moments de la vie d'un composant :

1. `beforeCreate` : n'existe pas avec composition API. Le contenu est à placer dans la fonction de `setup`
2. `created` : n'existe pas avec composition API. Le contenu est à placer dans la fonction de `setup`
3. `onBeforeMount` : Les templates sont compilés
4. `onMounted` : Le composant est monté et le DOM disponible. Le composant est complètement disponible.
5. `onBeforeUpdate` : Lancé au changement de données
6. `onUpdated` : Le rendu du DOM a été effectué après changement des données
7. `onBeforeUnmount` : appelé juste avant qu'un composant soit démonté
8. `onUnmounted` : appelé une fois le composant démonté
9. `onErrorTracked` : appelé quand un composant enfant reçoit une erreur
10. `onRenderTracked` : appelé une fois et indique quels sont les opérations trackées par le composant
11. `onRenderTriggered` : s'exécute à chaque fois qu'une opération trackée par le composant est réalisée
12. `onActivated` : utilisé quand un composant utilisant `<keep-alive>` est utilisé et activé (exemple : tabulation)
13. `onDeactivated` : utilisé quand un composant utilisant `<keep-alive>` est désactivé

Cycle de vie d'un composant utilisation

Des opérations et calculs peuvent être réalisés à différents moments de la vie d'un composant :

Les plus utilisés sont au montage avec "onMounted".

Attention, avec Composition API il faut bien penser à importer les fonctions de lifecycle avant de les utiliser, contrairement à l'Options API

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

  onMounted() {
    alert("Mon composant est monté")
  }
</script>
```



Le templating

Templating : les conditions

Les conditions utilisent l'instruction "v-if" pour tester si l'instruction est positive. Cette instruction doit être insérée directement dans le HTML. L'instruction "v-else" peut succéder à une instruction "v-if". Il est aussi possible d'exécuter des expressions ternaires : `{{ ok ? 'OUI' : 'NON' }}`. Exemple :

Partie script

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

  const personne= ref({ nom: "", sexe: "" })

  personne.nom = "Dupont";
  personne.sexe = "h";

</script>
```

Partie template

```
<template>
  <div>
    <h1 v-if="personne.sexe === 'h'">Bonjour monsieur {{
personne.nom }}</h1>
    <h1 v-else>Bonjour madame {{ personne.nom }}</h1>
  </div>
</template>
```

Les boucles

Les boucles “v-for” permettent de boucler sur un tableau. Elles doivent comporter une clef “key” permettant d’identifier le bloc généré dans le DOM.

L’exemple à droite affiche chaque élément du tableau dans une balise “<p>”.

A noter que key prend “:” devant car il s’agit d’une propriété dynamique.

```
<template>
  <div>
    <p v-for="element of tableau" :key="element">{{ element }}</p>
  </div>
</template>

<script setup>
  const tableau= ["Jean", "Catherine", "Jeanne"]
</script>
```

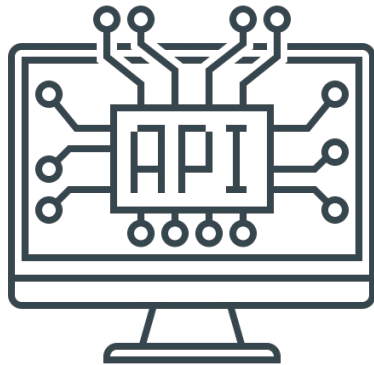

Le styling d'application

Chaque composant peut être stylé avec son propre css. Par défaut les styles de “App.vue” vont se répercuter sur les composants enfants.

Pour déclarer un bloc de style on utilise en fin de template un bloc style. Il est possible d'utiliser scss et sass :

```
<style lang="scss" scoped>
  a{
    color:pink;
  }
</style>
```

L'instruction “scoped” permet de contenir le style du composant et de le rendre “étanche”. Son style ne débordera ni sur ses parents, ni sur ses enfants.



APIs

Communication avec le monde

Votre application Vue.js n'est pas isolée et peut communiquer vers l'extérieur grâce aux APIs.

Vue peut effectuer des appels API nativement avec l'instruction `fetch()` ou avec le module Axios que nous utiliserons.

Pour simuler un serveur d'APIs nous allons utiliser JSON-Server qui permet de monter un faux serveur de test à partir de fichiers JSON. On lui ajoute un plugin d'authentification.

Nous allons l'installer avec : `npm install -D json-server json-server-auth`

Il suffit ensuite de créer un fichier "db.json" à la racine contenant au moins un objet.

Pour lancer le serveur il suffit d'exécuter la commande :

```
json-server ./db.json -m ./node_modules/json-server-auth --watch
```

Par défaut les points d'entrées se trouvent sur "http://localhost:3000/monObjet"

Axios

Axios est un module permettant de faire des requêtes HTTP.

Pour l'installer : `npm install axios`

Pour l'utiliser :

```
import axios from 'axios'  
  
axios.get("monURL")
```

Axios : get

Axios permet de récupérer des données avec la méthode get :

“this.axios.get(url)”.

Axios a un fonctionnement asynchrone, ce qui signifie que Javascript n’attend pas le résultat pour poursuivre le code. Il ne faut pas oublier d’utiliser les promises ou async/await pour éviter les bugs !

Le résultat retourné est un objet contenant toutes les informations sur la requête. Pour accéder aux données il faut utiliser la propriété “data” comme dans l’exemple :

// Version async / await

```
async myFunction() {  
  const url = "http://localhost:3000/cities";  
  const result = await axios.get(url)  
  return result.data  
},
```

// Version promises

```
myFunction() {  
  const url = "http://localhost:3000/cities";  
  axios.get(url).then((result)=>console.log(result.data))  
},
```

Axios : Get avec paramètres

Envoyer des paramètres à Axios il faut envoyer un objet “params” contenant les paramètres (sous forme d’objets eux aussi).

Exemple :

```
async created() {  
  const url = "http://localhost:3000/cities";  
  const target = { id:2}  
  const getCities = await this.axios.get(url,{params:target})  
  this.cities = getCities.data  
  console.log("cities",this.cities)  
},
```

Axios : authorisation

Il est possible de joindre à la requête axios un token d'authentification à travers l'objet header. A noter que le token doit avoir la chaine "Bearer" devant lui.

Exemple :

```
async created() {  
  const headers = { headers: { Authorization: `Bearer ${token}` } };  
  const url = "http://localhost:3000/cities";  
  const target = { id:2}  
  const getCities = await axios.get(url,{params:target},headers)  
  this.cities = getCities.data  
  console.log("cities",this.cities)  
},
```

Autres méthodes

Pour poster des données on utilise : `this.axios.post(urlApi,mesDonnees,headers)`

```
const city={title:"Strasbourg",isFavorite:false}  
const postCity = await axios.post(url,city)
```

Pour supprimer des données : `this.axios.delete(urlApi,headers)`

```
await axios.delete(`${url}/10`)
```

Pour mettre à jour les données : `this.axios.put(urlApi,mesDonnees, headers)`

```
const cityUpdated={title:"Dijon",isFavorite:false}  
const updateCity= await axios.put(`${url}/7`,cityUpdated)
```


Gestion des erreurs

Lorsque l'on manipule des requêtes HTTP, il est une bonne pratique de tester si l'opération s'est déroulées comme prévu. Pour cela on utilise le "try/catch".

Exemple :

Ici L'URL n'est pas valide

Tente d'exécuter le code dans le try. Si une erreur survient le bloc "catch" est exécuté

Code à exécuter en cas d'erreur. Catch prend en paramètre un objet contenant le détails de l'erreur

```
const url = "http://localhost:300000/cities";
const city={title:"Montpellier",isFavorite:false}

try{
  const postCity = await axios.post(url,city)
  console.log("postCity",postCity)
}catch(error){
  console.log("error",error)
  alert("Une erreur s'est produite !")
}
```



Le routing

Vue Router

En Vue, les pages d'une application sont aussi des composants.

Pour créer différentes pages dans notre application, il va falloir utiliser un router qui va faire le lien entre l'URL demandée et le composant souhaité.

Vue ne possède pas de router intégré, il va falloir intégrer un module officiel complémentaire : vue-router.

Pour cela utiliser la commande : `npm install vue-router@4`

Nous utiliserons le composant `<router-view />` pour charger le composant demandé par la route

Notre fichier main.js doit être mis à jour pour intégrer le router :

```
// main.js
...
import router from './router'
...

const app = createApp(App);

app.use(router);
```

Création du fichier de router

Le fichier contenant le fonctionnement doit être créé (celui là même qui est importé dans le main.js)

```
// router.js
import {
  createRouter,
  createWebHistory,
  type RouteRecordRaw,
} from "vue-router";

const routes: RouteRecordRaw[] = []

const router = createRouter({
  history: createWebHistory(),
  routes,
});

export default router;
```

Importe

- createRouter : pour créer notre router
- createWebHistory : permet une navigation en mode history
- type RouteRecordRaw -> types pour Typescript (à supprimer en js)

Création du router avec un paramètre le type de navigation et les routes de notre app

Créer un lien vers une page

Pour qu'un composant "page1" puisse emmener un visiteur vers un composant "page2", il va falloir ajouter une route. Dans le fichier "router/index.js" modifiez le tableau de routes :

Import du composant cible

```
...  
import Home from "@components/Home"
```

```
Vue.use(VueRouter)
```

```
const routes = [  
  {  
    path: '/',  
    name: 'Home',  
    component: Home  
  },  
  {  
    //Lazyloading de route  
    component: () => import("@views/Login.vue"),  
    path: "/login",  
    name: "Login",  
  },  
]
```

Une route est un objet content :

- path : l'url cible
- name : Le nom de la route
(on peut ainsi avoir des URLs indépendantes et de les changer sans casser son code)
- component : le nom du composant cible

Créer un lien

Créer un lien dans vue se fait avec l'instruction dans le template :

```
// vers une URL
<router-link to="/page2">page2</router-link>

//Vers un nom
// Notez le ":" devant le "to" comme il s'agit d'une propriété réactive
<router-link :to="{name : 'Page2'}">page2</router-link>
```

Faire un lien dans le script vue (pour une redirection par exemple) :

```
<script setup lang="ts">
// Import et initialisation du router
import { useRouter } from "vue-router";
const router = useRouter();

//Changement de page
router.push({ name: "myPage" });
</script>
```

Les liens avec des paramètres

Il faut tout d'abord indiquer au router qu'une page possède un paramètre. Dans le fichier "router/index.js" ajoutons une nouvelle page :

```
...
import Page3 from "@components/Page3"
...
const routes = [
  ...
  {
    path: '/page3/:userId',
    name: 'Page3',
    component: Page3
  }
]
```

Paramètre "userId". Les ":" devant indiquent que ce paramètre de route est dynamique

Un lien se fera alors de cette façon :

```
<router-link :to="{ name: 'Page3', params: { userId: 123 }}">User</router-link>
```

Récupération des données

Les données envoyées par la route sont récupérables dans le code vue :

```
<script setup>

import { useRoute } from "vue-router";

const id = useRoute().params.userId;

</script>
```


Les subrouters (nested routes)

Il est possible de définir des sous-routes. Par exemple, si on imagine un dashboard utilisateur de blog on pourrait avoir une route “/user” pourrait avoir des sous-routes comme “/users/profils”, “/users/posts” et “/users/comments”. Au niveau du routeur cela se traduit de la façon suivante :

```
const routes=[
  {
    path: '/user',
    name: User,
    component: UserComponent,
    children:[
      {
        path:"",
        name:"UsersPosts",
        component: UserPostsComponent,
      },
      {
        path:"profile",
        name:"UserPofile",
        component: UserProfileComponent,
      }
    ]
  }
]
```

Accessible par le nom de route UserPosts et disponible à l'adresse : /user . En effet path est vide.

Accessible par le nom de route UserProfile et disponible à l'adresse : /user/profile.

Les routes guards

Ce système permet de limiter l'accès à certaines routes. C'est très utiles pour créer des espaces personnels ou des consoles d'administration par exemple.

Pour les utiliser il faut :

1. Dans le router, ajouter un objet "meta" permettant de tagger les routes à protéger. Par exemple j'ajoute la balise "meta" "needAuth" à une de mes routes :

```
const routes = [  
  ...  
  {  
    path: '/page3/:userId',  
    name: 'Page3',  
    component: Page3,  
    meta: { needAuth: true }  
  }  
]
```

2. Bloquer la route grâce à la fonction “router.beforeEach” qui va se lancer avant chaque lancement de route. Cette fonction prend comme paramètre “to” qui correspond à la route voulue, “from” la route précédente et “next” la fonction permettant de rediriger vers la prochaine route. Voici un exemple :

S'exécute avant chaque route

Pour les besoins de l'exemple, notre utilisateur est authentifié 'en dur'

Vérifie que le route demandée possède le tag “needAuth”

Si l'utilisateur n'est pas authentifié et que le route est protégée, alors l'utilisateur est renvoyé sur la page de login

Laisse l'utilisateur accéder à la route demandée

```
// dans router/index
router.beforeEach((to, from, next) => {
  const isAuthenticated = true;
  const isProtected = to.matched.some(route => route.meta.needAuth);

  if (!isAuthenticated && isProtected) {
    next({ name: 'Login' });
  } else {
    next();
  }
});
```



Les Stores



Pinia

Pourquoi Pinia ?

Lors des spécification du Vuex 5 il s'est avéré qu'un side project les intégrant existait déjà. Il a donc été décidé que le Store officiel est maintenant Pinia. Il est possible qu'une fusion entre Vuex et Pinia ait lieu.

Avantages de Pinia :

- Les mutations ne sont plus nécessaires
- Pas d'organisation en modules
- Typescript beaucoup plus aisé qu'avec Vuex
- Compatible SSR
- Compatible avec les devtools
- Globalement Pinia est plus simple et léger à utiliser dans les composants

Installation de Pinia

Pinia s'installe grâce à la commande : `npm install pinia`

Puis dans le main.js :

```
import { createPinia } from "pinia";
```

```
const app = createApp(App);
```

```
app.use(createPinia());
```

Création d'un store en mode option

```
// dans stores/myStore.ts
```

```
import { defineStore } from "pinia";
```

```
//Pour typescript
```

```
type RootState = {  
  myState: MyInterface[];  
};
```

Permet de définir le contenu du state en le typant. Nécessaire uniquement pour Typescript

```
export const useMyStore = defineStore({
```

Création d'un store

```
  id: "myStore",
```

```
  state: () =>
```

```
  ({
```

```
    myState: [],
```

```
  } as RootState),
```

```
  getters: {
```

```
    myGetter: (state) => {
```

```
      return state.myState;
```

```
    },
```

```
  },
```

```
  actions: {
```

```
    updateMyState(myNewState){
```

```
      this.myState=myNewState
```

```
    }
```

```
  }
```

```
});
```

Création du state de notre application

Les getters permettent d'effectuer des traitements sur le state automatiquement.

Les actions servent à manipuler le state

Création d'un store en mode setup

// dans stores/myStore.ts

```
import { defineStore } from "pinia";  
import { ref, computed } from "vue"  
import { type MyInterface } from "@interfaces/myInterface"
```

```
export const useMyStore = defineStore("myStore", () => {
```

```
  const myState = ref<MyInterface>();
```

```
  const myGetter = computed(() => myState.value);
```

```
  const updateMyState = (newValue: MyInterface) => myState.value = newValue;
```

```
  return { myState, myGetter, updateMyState }
```

```
});
```

Création d'un store avec un identifiant

Création du state de notre application. Il s'agit d'une ref comme dans un composant

Les getters permettent d'effectuer des traitements sur le state automatiquement. Ce sont des computed

Les actions servent à manipuler le state

Expose les éléments du store

Utilisation dans un composant

```
<script setup>  
import { useMyStore } from "@stores/myStore";
```

```
const myStore = useMyStore();
```

Initialisation de mon store

```
myStore.myState = ["Hello Pinia"]
```

Modification directe du state

```
console.log(myStore.myState)
```

Affichage du contenu de mon state

```
console.log(myStore.myGetter)
```

Utilisation de mon getter

```
myStore.updateMyState(myNewState)
```

Exécution d'un action (update le state)

```
const { myState } = storeToRefs(myStore)
```

Convertit une valeur de mon state en ref

```
myStore.$subscribe((mutation, state) => {  
  console.log({mutation, state})  
});
```

Observe les changement du store

```
</script>
```



Vuex

Pourquoi Vuex ?

L'un des principal problème lorsque l'on crée une application JS réactive est la gestion de son état. En effet lorsque l'on modifie les données de l'application dans un composant comment répercuter les changement dans toute l'application ?

Nous avons bien les événements et props qui permettent d'échanger les données entre composant mais cette solution pose beaucoup de problèmes à mesure que le nombre de composants et d'opérations augmente.

La meilleure solution est donc de centraliser les données de l'application dans un seul et même endroit : un store d'application. Vuex est l'équivalent de Redux pour React et permet à tous les composant d'aller interagir avec des données synchronisées à un seul et même endroit : le store d'application.

Pour utiliser Vuex, il faut l'installer avec la commande : `vue add vuex`

L'installateur va créer un répertoire dans notre application

Fonctionnement de Vuex

Le fichier “store/index.js” va contenir notre store par défaut. On peut y voir 4 objets

state => Contient les propriétés que l'on souhaite avoir dans le store. exemple : persons:[]

mutations => Mettent à jour les données contenue dans le state. En effet Il n'est pas autorisé de modifier directement les données du store dans ses composants.

actions => Comme dans les composants, ces fonctions sont l'équivalent des méthodes. Elles sont les points d'entrée dans le store. Une action va ensuite utiliser les mutations pour interagir avec les données dans le state.

modules => Permet de découper notre store en plusieurs modules de store pour éviter d'avoir un fichier de store énorme. En général il est préférable de créer tout de suite un store par objet dont on veut gérer l'état.

Les modules Vuex

Utiliser les modules Vuex est très simple. Dans le fichier “store/index.js”, il suffit d’importer son module et de l’ajouter dans l’objet “modules”.

Exemple avec un module “movies”:

```
import Vue from "vue";
import Vuex from "vuex";
import movies from "@store/modules/movies";

Vue.use(Vuex);

export const store = new Vuex.Store({
  modules: {
    movies,
  },
});
```

Contenu d'un module

Le contenu d'un module est similaire à celui de l'index du store. Il est générable automatiquement grâce à la commande "vmodule".

L'objet modules a disparu et l'objet getters a fait son apparition.

Getters permet de récupérer les données qui se trouvent dans le state. Il est possible de les utiliser pour créer des fonctions de tri par exemple.

```
export default {
  namespaced:false,
  state: {
    value: 'my value'
  },
  getters: {
    value: state => {
      return state.value;
    }
  },
  mutations: {
    updateValue(state, payload) {
      state.value = payload;
    }
  },
  actions: {
    updateValue({commit}, payload) {
      commit('updateValue', payload);
    }
  }
};
```

Utilisation de Vuex dans un composant

Vuex est automatiquement importé dans les composants.

Pour récupérer des données du state : `this.$store.state.monModule.stateCible`

Pour utiliser une action : `this.$store.dispatch("monAction", paramètres)`

Attention à bien mettre l'action cible entre " "

Pour utiliser les getters : `this.$store.getters['monGetter'](parametres)`

Exemple d'utilisation de Vuex

```
// Dans n'importe quel composant
```

```
// Utilise une action pour initialiser les données du store  
const movies = [{ id: "1", title: "Le chant du loup" }, { id: "2", title: "Matrix" }]  
this.$store.dispatch("updateMovies", movies);
```

```
// Récupère les données grâce au state  
const moviesFromState = this.$store.state.movies.movies;  
console.log("movies venant du state", moviesFromState)
```

```
// Récupère les données avec le getters. Ici le getter permet de récupérer un //  
film par rapport à un index donné en paramètre  
const moviesFromGetters = this.$store.getters['getMovie'](0);  
console.log("movies venant de getter", moviesFromGetters)
```

```
// store/modules/movies  
export default {  
  state: {  
    movies: []  
  },  
  getters: {  
    getMovie: state => index => {  
      return state.movies[index];  
    }  
  },  
  mutations: {  
    updateMovies(state, payload) {  
      state.movies = payload;  
    }  
  },  
  actions: {  
    updateMovies({ commit }, payload) {  
      commit('updateMovies', payload);  
    }  
  }  
};
```

Les mapGetters

Pour éviter d'avoir des appels du store complexe comme “this.store.state.achats.achats” on peut utiliser les mapGetters. Ils agissent “comme des computed” properties d'un composant mais dans un store.

Création d'un getter dans le store :

```
export default {  
  state: {  
    value: 'my value'  
  },  
  getters: {  
    monGetter: state => {  
      return state.value;  
    }  
  },  
  mutations: {  
    ...  
  },  
  actions: {  
    ...  
  }  
};
```

Les mapGetters (suite ...)

Pour utiliser les getters dans un composant il faut importer mapGetters avec “import { mapGetters } from "vuex";” puis dans les computed ajouter les getters souhaités avec “...mapGetters(["monGetter"])”. Les getters seront disponible dans le code comme n’importe quelle computed properties.

```
<script>
import { mapGetters } from "vuex";

export default {
  data() {
    return {
      ...
    };
  },
  computed: {
    ...mapGetters(["monGetter"]),
  },
  ....
}
```

Composition API avec les stores

```
<template>...</template>
```

```
<script>  
import { useStore } from 'vuex';
```

Import de useStore qui permet d'utiliser le store uniquement dans une fonction setup

```
export default ({
```

```
  setup() {
```

```
    const store = useStore()
```

Initialisation du store

```
    store.dispatch("actionToTrigger", ressources)
```

Exécute une action

```
    const ressources= store.state.monModule.ressources
```

Récupération de données du state

```
    const ressourcesGetter= store.getters["monGetter"]?(paramOptionnel)
```

Utilisation d'un getter

```
  }
```

```
})
```

```
</script>
```

L'API Setup - nouveauté Vue 3.2

L'API composition a une nouvelle manière d'être utilisée en mode "setup". Elle est encore plus simple à utiliser et lisible. Cependant il ne s'agit pas véritablement d'une nouvelle façon d'écrire un composant mais de "sucre syntaxique".

Avantages :

- Plus besoin de return les variables et constantes
- Plus besoin d'enregistrer les composants
- Plus besoin de commencer le composant par export default

Exemple Setup API

```
<template>
  <ChildComponent />
  <h1>{{ counter }}</h1>
  <p>Affichage d'une props {{ props.myProps }}</p>
  <button @click="incrementAction">Increment</button>
</template>
```

Activation du mode setup

Plus d'enregistrement du composant enfant

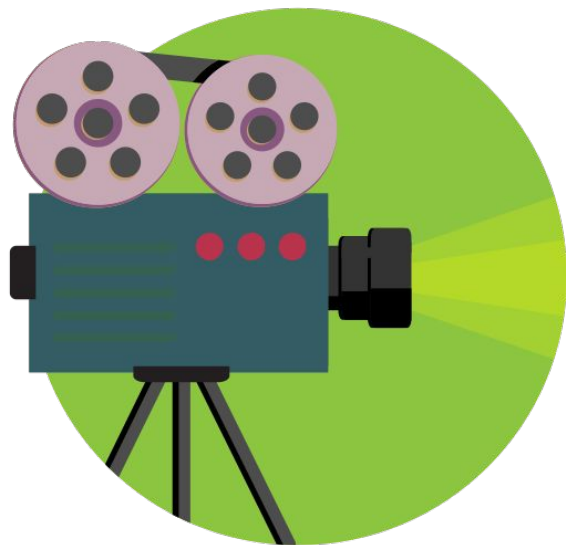
```
<script setup>
import ChildComponent from "@/components/ChildComponent";
import { ref, defineProps, defineEmits } from "vue";
```

```
const props = defineProps({ myProps: String });
const counter = ref(0);
const emit = defineEmits(["my-custom-event"]);
```

defineProps va nous permettre de définir les props passées en paramètre

```
const incrementAction = () => {
  emit("my-custom-event", "payload");
  counter.value++;
};
</script>
```

defineEmits permet de définir les événements custom (pour passer des infos du composant enfant au parent par exemple)



Les animations

Les animations entre les pages

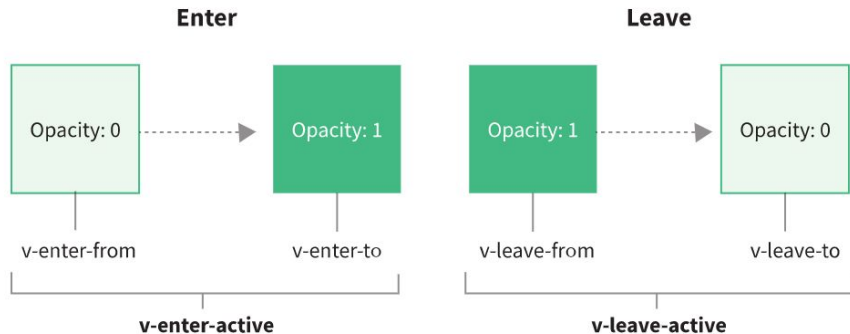
Pour activer les animations il faut ajouter la balise transition autour du router-view :

```
<router-view v-slot="{ Component }">  
  <transition name="slide" mode="out-in">  
    <component :is="Component"></component>  
  </transition>  
</router-view>
```

Les transitions portent un nom qui permet de les identifier.

Le mode “out-in” permet de lancer l’animation d’entrée une fois l’animation de sortie terminée

Les transitions entre les pages fonctionnent selon le schéma suivant (remplacer le “v-” par “nomtransition-”)



Transition entre pages : CSS

Les animations sont gérées par le CSS et s'effectuent dans l'ordre du schéma précédent. Voici un exemple sur une transition appelée "fade" :

Explications :

- L'état de départ est fixé avec une opacité à 0 et décalé sur la gauche de 100px pour donner un effet de slide d'entrée
- L'état de sortie est fixé à 0 et décalé sur la droite pour donner un effet de slide de sortie
- ".page-transition-enter-to" et ".page-transition-leave" ne sont pas nécessaires car la transition réinitialise à zéro les modifs css précédentes
- Les transitions sont fixées à 0.5s en utilisant l'effet "ease-out". Cela concerne les classes d'entrée et de sortie

```
.page-transition-enter-from {  
  opacity: 0;  
  transform: translateX(-100px);  
}  
  
.page-transition-leave-to {  
  opacity: 0;  
  transform: translateX(100px);  
}  
  
.page-transition-enter-active,  
.page-transition-leave-active {  
  transition: all 0.5s ease-out;  
}
```

Animer les listes

Il peut être intéressant d'animer les listes pour que l'utilisateur voit l'impact de ses actions sur l'application.

Le fonctionnement est similaire à la transition de pages à la différence qu'il faut ajouter une balise "transition-group" comportant un propriété "name" et "tag". Tag correspond à la balise qui va entourer la liste.

Exemple :

```
<transition-group name="liste-transition" tag="ul">  
  <li v-for="index in 10" :key="index">{{ index }} </li>  
</transition-group>
```

Il faut ensuite ajouter le css qui gèrera l'animation comme lors des pages transitions. Exemple pour une suppression d'un élément de liste qui slide sur la droite :

```
.liste-transition-leave-to {  
  opacity: 0;  
  transform: translateX(100px);  
}  
  
.liste-transition-leave-active {  
  transition: all 0.5s ease-out;  
}
```

Il est aussi possible d'ajouter un css qui sera lancé en case de changement de place des éléments de la liste :

```
.liste-transition-move {  
  transition: transform 0.5s ease-out;  
}
```



Les tests unitaires

Pré-requis

Pour effectuer nos tests nous utiliserons le moteur de test Jest. Pour cela nous aurons besoin d'installer "vue-test-utils" pour pouvoir manipuler nos éléments Vue.js avec Vitest le framework de test inspiré de Jest.

- Installer Vitest avec `npm install -D vitest`
- Installer JsDom avec `npm install jsdom`
- Installer les vue-test-utils avec `npm install --save-dev @vue/test-utils`
- Installer le plugin VSCode "Vitest"

Modifier le fichier "package.json" et ajouter dans la partie "scripts" :

```
{  
  ...  
  "test": "vitest --environment jsdom",  
  "coverage": "vitest run --coverage"  
}
```

Prérequis (suite)

En cas d'utilisation de Typescript ajouter dans le fichier "tsconfig.vite-config.json"

```
{
  "extends": "@vue/tsconfig/tsconfig.node.json",
  "include": ["vite.config.*"],
  "compilerOptions": {
    "composite": true,
    "types": ["node", "vitest"]
  }
}
```

Effectuer un test simple

Les fichiers de test doivent porter l'extension ".spec.ts".

Il est alors possible d'exécuter les tests avec la fonction "npm run test"

```
import { describe, it, expect } from "vitest";  
import { addNumbers } from "../index";  
  
describe(">>>>>addNumbers", () => {  
  it("should return sum of two numbers", () => {  
    const a = 5;  
    const b = 7;  
  
    const expected = 12;  
    expect(addNumbers(a, b)).toBe(expected);  
  });  
});
```

Import des fonctions utiles de vitest

Import de la fonction à tester

describe permet de créer un groupe de tests

it permet de définir un test

expect compare le résultat prévu au résultat obtenu

Vérifier un résultat

```
//Trouve un composant et vérifie la présence d'un texte  
expect(wrapper.find('[data-test-id="message"]').text()).toContain(monTexte);
```

```
//Test l'existence d'un composant  
expect(wrapper.find('[data-test-id="message"]').exists()).toBe(false);
```

```
//Tests sur les nombres  
expect(2 + 2).toBe(4);  
expect(value).toBeGreaterThan(3);  
expect(value).toBeGreaterThanOrEqual(3.5);  
expect(value).toBeLessThan(5);  
expect(value).toBeLessThanOrEqual(4.5);
```

```
//Test une expression régulière  
expect('Christoph').toMatch(/stop/);
```

```
//test de tableau  
const monTableau = ["chocolat", "pizza", "biere"]  
expect(monTableau).toContain("biere");
```

```
//Test la présence de HTML  
expect(wrapper.html()).toContain('<span>0</span>')
```


Tester un composant

Pour tester un composant il faudra en plus de vitest utiliser les vue test utils qui donnent des outils pour permettre d'interagir avec les composants. Il faut tout d'abord charger un composant une fonction de montage :

- "mount" pour charger un composant et ses sous composants
- "shallowMount" charge uniquement un composant "propsData"
- ajouter le commentaire `//@vitest-environment jsdom` en haut du fichier pour simuler un dom en node

Exemple :

```
const wrapper = mount(RessourceItem, {  
  props: { ressource: testRessource, isBookmark: false },  
  global: {  
    plugins: [ElementPlus, router, createPinia()],  
  },  
});
```

props permet de charger des props au montage du composant

plugins permet de charger des éléments nécessaire au fonctionnement du composant comme la librairie UI

Déclencher un événement

On utilise “.trigger” pour déclencher des événements. Ils s'exécutent de manière asynchrone. Ils faut donc penser à utiliser au choix les callbacks, les promises ou async/await

```
//Click sur un événement avec les promises
wrapper.find('[data-test-id="buttonAction"]').trigger('click').then(() => {
  expect(wrapper.find('[data-test-id="message"]').text()).toContain('Fabien');
});


//Click sur un événement avec async await
it('Should display name', async() => {
  await wrapper.find('[data-test-id="buttonAction"]').trigger('click')
  expect(wrapper.find('[data-test-id="message"]').text()).toContain('Fabien');
})
```

Mocker les fonctions

Il est possible de remplacer des fonctions par de fausses fonctions Vitest. Cela permet d'éviter des appels de fonction longues qui accèdent à des API par exemple et aussi d'espionner si les fonctions ont bien été appelées. On peut mocker des services, le router, le store, axios etc...

Pour créer une fonction mock on utilise : `vi.fn()`

```
const mockFn = vi.fn();  
mockFn();  
expect(mockFn).toHaveBeenCalled();
```



Vérifie que la fonction a bien été appelée



Les tests E2E

Cypress

Cypress est un utilitaire permettant de faire des tests end to end. A travers le navigateur, Cypress va simuler des actions utilisateur définies et vérifier que les tests fonctionnent (ou pas).

- Installer Cypress : `npm install cypress -D`
- Lancer Cypress : `npx cypress open` (ouvre l'interface de Cypress)

Le fichier “cypress.config.ts” permet de configurer Cypress. Cela peut être utile, notamment pour les variables d'environnement :

```
export default defineConfig({  
  env: {  
    login_url: `http://localhost:3000/login`,  
    admin_validation_url: `http://localhost:3000/admin/validation`,  
  },  
  ...  
})
```

Ajout de deux routes dans les variables d'environnement. Si leur URL change, cela ne cassera pas les tests.
Pour les utiliser on pourra écrire :
`Cypress.env("admin_validation_url")`

Structure d'un test

L'écriture des tests avec Cypress reprend les mêmes concepts que pour les tests unitaires avec les mots "describe", "it", "beforeEach"...

Exemple

```
/// <reference types="cypress" />

describe("Nom du groupe de test", () => {

  it("Explication du test attendu", () => {
    ...
  });
});
```

Quelques fonctions utiles

Ouvrir une page : `cy.visit("url")`

Comparer : `cy.elementAComparer.should("be.equals", "valeurDeTest");`

Récupérer un élément dans la page : `cy.get("[data-test-id=valeur]")`

Cliquer sur un bouton : `cy.get("[data-test-id=monBouton"]').click();`

Ecrire dans un champ : `myField.type("valeurDeTest");`

Vérifie l'existence d'un élément : `cy.get("[data-test-id=monElement]").should("exist");` (ou `not.exist`)

Nettoyer un champ : `cy.get("[data-test-id=monChamp"]').clear();`

Attendre 100ms : `cy.wait(1000)`

Reloader la page (ajouter `true` pour vider aussi le cache) : `cy.reload(true)`

Récupérer et vérifier des cookies : `cy.getCookie('token').should('have.property', 'value', '123ABC')`

Définir des cookies : `cy.setCookie('foo', 'bar')`

Vider les cookies : `cy.clearCookies()`

Quelques codes “should” utiles

Doit inclure : `.should('include', 'Kitchen Sink')`

Doit être présent dans un tableau : `.should('be.oneOf', [200, 304])`

Doit être invisible / visible : `.should('not.be.visible') / .should('be.visible')`

Doit avoir une classe : `.should('have.class', maClasse)`

Doit contenir un chaîne : `.should('contain', 'maChaine')`

Vérifie l'égalité : `.should('eq', 'ma chaîne')`

Vérifie la nullité : `.should('be.null')`

Vérifie que c'est vide : `.should('be.empty')`



Aller plus loin avec Vue.js

Watcher vs Computed

Les watchers permettent d'observer une propriété réactive et d'effectuer des changements au moment de la mise à jour.

A partir de la valeur modifiée, il est possible de récupérer son contenu avant et après la mise à jour.

Les watchers sont donc exécutés une fois, tandis que les computed sont exécutées en continue. Les computed sont aussi plus performantes en termes de vitesse d'exécution car elles sont "cachées" alors que les watchers sont calculés à chaque fois.

```
import { watch } from "vue";  
...  
watch(valeur, (nouvelleValeur, ancienneValeur) => {  
  //Opérations à réaliser  
  console.log(`Ancienne valeur ${ancienneValeur}`);  
  console.log(`Nouvelle valeur ${nouvelleValeur}`);  
});
```

Directives personnalisées

Les directives sont les mots commençant par “v-” dans les templates (comme v-if ou v-for). Il est possible d’en créer de nouvelles !

//Dans la partie template

```
<template>  
<div v-directive-perso="test">Lorem  
ipsum</div>  
</template>
```

Utilisation de la directive personnalisée dans le template

```
app.directive("directive-perso", {  
  created: (el, binding) => {  
    console.log(binding.value);  
  },  
});
```

Définition de la directive avec app.directive(). Elle prend en paramètre le nom et un objet : created, beforeMount, updated, mounted, unmounted

Prend en paramètre “el” pour l’élément du dom en cours (exemple : la div en cours) et “binding” qui contient les données passées en paramètres

Création d'applications mobiles & desktop

Il est possible de créer des applications mobiles & desktop en utilisant Vue.js.
Plusieurs systèmes permettent de le faire :



Ionic



Vue Native
(deprecated)



Quasar

Quasar

Permet de réaliser des applications javascript qui sont exécutables sur Android, IOS mais aussi sur ordinateur. Une seule code base permet de faire tourner son application partout. Pour la version mobile Cordova est utilisé pour embarquer l'application et Electron est utilisé pour la version ordinateur. Le principal problème des systèmes basé sur Cordova est le manque de performance.

- Plus d'infos sur <https://quasar.dev/>
- Nombres d'étoiles Github : 14.900 au 03/06/2020

Ionic

A un fonctionnement similaire à Quasar (fonctionne avec Capacitor au lieu de Cordova). Ionic est au départ basé sur Angular. Devant la perte de popularité de ce dernier l'équipe de Ionic a décidé de rendre leur système non dépendant du framework et on peut l'utiliser depuis la version 4 avec React et Vue. En pratique, il reste encore lié à Angular et s'ouvre un peu plus à React mais le manque de documentation pour l'implémenter en Vue.js me fait préférer Quasar dans le cadre d'un développement avec Vue.

- Plus d'infos : <https://ionicframework.com/>
- Github stars : 41.000

Search Engine Optimisation

Les applications single page sont plus difficile à référencer que les applications “classiques”. En effet leur code source n’est pas visible directement. Pour s’en rendre compte il suffit d’afficher le code source d’une page. Il n’y figure que quelques lignes et le contenu à indexer est absent.

Google a bien progressé et peut maintenant indexer les SPA. Une sous réserve que les requêtes asynchrones renvoyant le contenu arrive en moins de 300ms (source <https://www.smashingmagazine.com/2019/05/vue-js-seo-reactive-websites-search-engines-bots/>).

En règle général il vaut mieux éviter de se fier à JS pour avoir un bon référencement. On va donc voir deux méthodes : Le Server Side Rendering (SSR) et le pré-rendering

Le Server Side rendering

Utilise un serveur Node.js pour générer le code de l'application sur le serveur. Le code source est donc bien visible. Créer son application avec Nuxt.js permet de faciliter sa mise en place.

Avantages

- + Le code est disponible pour toutes les pages (comme avec du PHP par exemple)
- + Adapté aux sites très dynamiques (exemple forum, sites collaboratifs)

Inconvénients

- Complexe à mettre en place (Sauf avec Nuxt)
- Demande de la puissance serveur et risque de ralentir l'application
- Si l'application est lente le référencement sera dégradé

Le pré-rendering

Certaines pages sont générées à la compilation de l'application. Cela permet d'avoir le code source des pages essentielles au référencement et un temps de réponse extrêmement rapide. C'est, selon moi, la solution à privilégier si l'application ne possède pas beaucoup de données dynamiques.

Avantages

- + Le code est disponible sur certaines pages
- + Plus facile à mettre en place que le SSR
- + Rapide
- + Sollicite peu le serveur

Inconvénients

- Il faut configurer les pages à générer
- Pas adapté aux applications très dynamiques (exemple forum, réseau sociaux ...)

Renseigner le header de son application

Un module complémentaire est nécessaire : vue-header. Utilisez la commande pour l'installer : `npm install vue-meta`

Ajoutez le module à votre main.js :

```
...  
import VueMeta from 'vue-meta'  
...  
Vue.use(VueMeta)
```

Et ajouter dans le router :

```
// Ajouter dans le router  
import Router from 'vue-router'  
import Meta from 'vue-meta'  
  
Vue.use(Router)  
Vue.use(Meta)
```

Dans les composants de votre application ajoutez le bloc head :

Code pour renseigner le header de page

```
export default {  
  ...  
  metaInfo() {  
    return {  
      title: "titre de la page",  
      titleTemplate: null,  
      meta: [  
        {  
          name: "description",  
          content: "Description de la page"  
        }  
      ]  
    };  
  },  
  ...  
}
```

Créer un sitemap pour son site

Pour être sûr que toutes les pages de mon application sont bien découvertes par les moteurs de recherche, il est utile d'avoir un sitemap.

Pour cela, il faut créer un fichier sitemap.xml à l'intérieur du répertoire “/public”.

Exemple de fichier sitemap.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.example.com/</loc>
    <lastmod>2005-01-01</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.8</priority>
  </url>
</urlset>
```

Les variables d'environnement avec Webpack

Pour définir une variable d'environnement il suffit de rajouter un fichier “.env” à la racine du projet et d'y ajouter les variables commençant par “VUE_APP”.

Exemple : `VUE_APP_MA_VARIABLE="http://monsitedeprod.com"` .

Il est possible d'avoir un fichier “.env.development” qui sera chargé automatiquement en mode développement ou “.env.production” qui sera chargé en production. Il est aussi possible d'ajouter l'extension “.local” pour exclure les fichiers de git. Exemple : “.env.production.local”.

Attention lors de la modification de ces fichiers le serveur doit être redémarré.

Pour les utiliser dans les fichiers “.vue” il faut utiliser la syntaxe : `process.env.VUE_APP_MA_VARIABLE`

Les variables d'environnement avec Vite

Pour définir une variable d'environnement il suffit de rajouter un fichier “.env” à la racine du projet et d'y ajouter les variables commençant par “VITE_”.

Exemple : `VITE_MA_VARIABLE="http://monsitedeprod.com"`.

Il est possible d'avoir un fichier “.env.development” qui sera chargé automatiquement en mode développement ou “.env.production” qui sera chargé en production. Il est aussi possible d'ajouter l'extension “.local” pour exclure les fichiers de git. Exemple : “.env.production.local”.

Attention lors de la modification de ces fichiers le serveur doit être redémarré.

Pour les utiliser dans les fichiers “.vue” il faut utiliser la syntaxe : `import.meta.env.VITE_MA_VARIABLE`

Créer un bus d'événements

Un bus d'événement est un pattern qui permet de faire communiquer à travers un événement n'importe quel composant. Pour cela il faut par une librairie externe : MITT.

Pour l'installer : `npm install --save mitt`

Puis créer un fichier JS, par exemple “@/plugins/eventBus.js” avec le code :

```
import mitt from "mitt";  
export default mitt();
```

Utilisation du bus d'événements

```
//Composant émetteur
import EventBus from "@plugins/eventBus";
...
maMethode() {
  ....
  //Envoi de l'événement
  EventBus.emit("nom-evenement", payload);
},
```

```
//Composant récepteur
import EventBus from "@plugins/eventBus";
...

//Réception de l'événement
EventBus.on("nom-evenement", payload=> {
  //opérations à réaliser
});
```


VueI18N installation

Installation avec : **npm i vue-i18n@9**

Ajouter dans le fichier vite.config.ts :

```
alias: {  
  ...  
  'vue-i18n': 'vue-i18n/dist/vue-i18n.cjs.js',  
},
```

Ajouter dans le fichier tsconfig.json

```
{  
  "compilerOptions": {  
    "resolveJsonModule": true,  
  },  
  "include": [  
    "src/locales/*.json"  
  ]  
}
```

VueI18N installation (suite)

Créer un répertoire “src/locales” et ajouter des fichiers de langue. exemple de fichier “fr.json”

```
{  
  "messages": {  
    "hello": "Bonjour le monde !",  
    "bestFilms": "Meilleurs films ",  
  }  
}
```

Dans le fichier main.ts ajouter : `import { createI18n } from "vue-i18n"`

```
const i18n = createI18n({  
  legacy: false,  
  locale: "fr",  
  fallbackLocale: "fr",  
  messages: { fr, en }  
})  
app.use(i18n);
```

Obligatoire avec setupapi

Langue par défaut

Langue à utiliser s'il manque une traduction

Vue18N utilisation

Dans les templates : `{{ $t("texteATraduire") }}`

Exemple de système de changement de langue (avec Vuetify :

```
import { useI18n } from "vue-i18n";
```

```
...
```

```
const { availableLocales, t } = useI18n();
```

```
console.log( t( "myTranslation" ) )
```

```
const itemsLocales = computed(() =>
```

```
  availableLocales.map((locale) => ({ text: locale, value: locale })))
```

```
);
```

```
<template>
```

```
  <v-select
```

```
    v-model="$i18n.locale"
```

```
    label="Choix de la langue"
```

```
    :items="itemsLocales"
```

```
    item-title="text"
```

```
    item-value="value"
```

```
  />
```

```
</template>
```

Permet d'utiliser vue18n dans la partie script des composants

Génération d'un tableau avec les langues disponibles

Permet le switch de langue

Ressources du projet

L'exercice lié à ce cours est disponible sur Github :

<https://github.com/fgrx/Vue3-Typescript-Pinia-Vuetify>

