Vue.js

Alan Piron-Lafleur

Mise en place

VueJS

C’est un framework frontend qui est relativement facile à prendre en main.
Il permet de découper nos applications en composants réutilisables et de les injecter facilement à des projets existants.

Pour ce cours, libre à vous de votre IDE.
Le mieux en IDE gratuit, c’est VsCode.

Créez un dossier coursVueJS sur votre ordinateur, ouvrez le dans votre IDE… c’est parti !

Si vous utilisez VSCode, voici des extensions :

- Volar : Facilite le codage en vueJS.
- Live server : pour refresh automatiquement l’affichage lorsqu’on sauvegarde un fichier.

Le modèle MVVM

Le modèle MVVM : model - view - viewModel est l'architecture logicielle utilisée par Vue.js.

Le M : model
Contient toutes les données liées à la logique métier (base de données, api externes). Sera en charge de la transmission des données au backend.

Le V : View

C'est l'interface graphique qui fait le lien entre les actions de l'utilisateur et le modèle. Elle définit où sont placés les éléments graphiques.

Le VM : ViewModel

Fait la transition entre les deux. Un changement dans le modèle (modification du prénom ?) est transmis à la vue par le ViewModel. Un changement sur la vue est transmis au modèle par le ViewModel.

Decouverte

Nous allons commencer avec une série de pratique et d’exercices pour vous montrer Vue.js

Créer un dossier cours_decouverte dans votre dossier CoursVueJS et placez y un fichier decouverte-01.html

Je vous montre comment s’utilise VueJS via le CDN.

Le modèle MVVM

- On tape html:5 pour générer un template de base.
- Rdv sur le site de vueJS -> get started -> quick start pour récupérer le CDN, à mettre dans le <head>
- Voici le code :

```
<div id="app">

   <h2>Je test {{appName}}</h2>

</div>

    <script>
        app = Vue.createApp({
            data(){
                return {
```

Du JS de partout

Ecrire entre les accolades s’appelle : utiliser les interpolations de Vue.js
C’est la formation la plus élémentaire de liaison de données du framework.

Il est possible d’écrire du code JS directement dans les accolades.
Je vous montre.

Du JS de partout

Modifier le <h2> comme suit :

```
<h2>Je test {{appName.toUpperCase()}}</h2>
```

Ajouter un paramètre “complexe” et un paragraphe pour tester :

```
return {
      appName: "mon super projet 1",
      complexe: false
}

<h2>Je test {{appName.toUpperCase()}}</h2>
<p>Le projet est {{ complexe ? "dur" : "simple" }}</p>
```

Propriétés calculées

Squelette

Pour nos prochains projets, créons un fichier squelette.html

- Copiez collez le fichier decouverte-01.html, nommez le fichier squelette.html
- Enlevez tout ce qui n'est pas générique, votre <body> ressemblera à ça :

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

<script>
    app = Vue.createApp({

    });
    app.mount("#app");
</script>
```

Propriétés calculées

Les fonctions que nous utiliserons dans notre application VueJS ne doivent pas être placées dans le modèle (pas dans data()).

Il existe une propriété computed() pour cela, c'est le fameux ModelView.

Créez un fichier decouverte-02.html à partir de squelette.html… je vous montre la suite.

Propriétés calculées

```
<div id="app">

        <h2>{{transformString}}</h2>

    </div>

    <script>
        app = Vue.createApp({
            data(){
                return {
                    string: "Une phrase au hasard"
                }
            },
            computed: {
                transformString(){
```

Exercice

Créez un dossier exercices dans cours_decouverte.

Créez un fichier exercice-01.html et réalisez l'affichage suivant :

"Il est précisément : HH:MM:SS (affichage FR)"

"Il est précisément : HH:MM:SS (affichage US)"

👉Aide pour coder propre :

- le Model donne les données nécessaires au programme (ici, une date)
- le ModelView utilise la donnée pour la transformer (en français, en US)
- toLocaleTimeString('en-US') est une fonction qui fonctionne !

Correction

```
<div id="app">

        <p>Il est précisément {{dateFr}} (affichage FR) </p>
        <p>Il est précisément {{dateUs}} (affichage US) </p>

    </div>

    <script>
        app = Vue.createApp({
            data(){
                return {
                    d: new Date()
                }
            },
            computed: {
```

Data binding

Data binding

Vue.js va nous permettre de binder les attributs de nos éléments html avec des données du modèle. C’est le data binding.

Je vous montre dans un fichier decouverte-03.html

Data binding

```
<div id="app">
        <img v-bind:src="url" /> <- on peut enlever le v-bind, ça marche aussi
    </div>

    <script>
        app = Vue.createApp({
            data(){
                return {
                    url: "https://vuejs.org/images/logo.png"
                }
            }
        });
        app.mount("#app");
    </script>
```

Data binding

Après le binding sur les attributs, on peut binder du texte brut et html, directement.

A la suite de decouverte-03.html, je vous montre.

Data binding

```
<div id="app">
        <p v-text="presentation"></p>
        <p v-html="presentationHTML"></p>
    </div>

    <script>
        app = Vue.createApp({
            data(){
                return {
                    presentation: "Salut, c'est Patrick",
                    presentationHTML: "Salut, c'est <strong>Patrick</strong>"
                }
            }
        });
        app.mount("#app");
    </script>
```

Data binding

Il est évidemment possible (et souhaitable) d’utiliser des objets JSON provenant du modèle pour les binder dans notre HTML.

Je vous montre.

Data binding

```
<div id="app">
        <p>Ici c'est {{ personne.prenom }}, j'ai {{ personne.age }} ans</p>
    </div>

    <script>
        app = Vue.createApp({
            data(){
                return {
                    personne : {
                        prenom: "Alan",
                        nom: "PL",
                        age: 31
                    }
                }
            }
        });
```

Modifier des données

Dans Vue.js, il sera simple de modifier l’affichage, après un clic sur un bouton, après le post d’un formulaire, etc ..

Il suffira de changer les propriétés de notre app au moment souhaité.

Voyons cela ensemble.

On met notre app dans une constante :

```
const appVue = app.mount("#app");
appVue.personne.prenom = "John";
```

Exercice

Créez un fichier html exercice-02.html et réalisez une horloge (qui se met donc à jour automatiquement) :

“Il est précisément : HH:MM:SS”

👉Aide pour coder propre :

- utilisez la fonction setInterval() pour modifier le modèle chaque seconde

Correction

```
<div id="app">
        <p>{{ horloge }}</p>
    </div>

    <script>
        app = Vue.createApp({

            data(){
                return{

                    horloge: new Date().toLocaleTimeString()

                }
            }

        });
        const appVue = app.mount("#app");
```

Two way binding

Nous venons de voir le fait de binder des données de notre modèle pour les afficher dans la view.

Nous allons maintenant voir l’inverse : binder un affichage de notre view pour modifier le modèle.

C’est parti dans decouverte-04.html

Two way binding

```
<div id="app">
        <input type="text" placeholder="Votre nom" v-model="nom">
        Bonjour {{ nom }}
    </div>

    <script>
        app = Vue.createApp({

            data(){
                return{
                    nom: ''
                }
            }

        });
        app.mount("#app");
    </script>
```

Two way binding : textarea

Le textarea est un peu spécial, dans le sens où ce qui sera tapé dedans ressortira sans saut de ligne.

```
<div id="app">
    <textarea v-model="unTexte" cols="40" rows="7"></textarea>
      <div>{{ unTexte }}</div>
</div>

      <script>
        app = Vue.createApp({

            data(){
                return{

                    unTexte: ""

                }
```

Two way binding : textarea

Pour que l’affichage se passe bien, il faut ajouter à la div qui affiche le texte la propriété CSS : `white-space: pre-line;"`

A propos de v-model

Appliqué à un élément html, l'attribut v-model de Vue.js ne remplace pas le même attribut :

- text et textarea : v-model s'attache à l'attribut value
  <input type="text" v-model="nom" /> <- modifie l'attribut value de l'input
- checkboxes et radiobuttons s'attache à l'attribut checked;
  <input type="checkbox" v-model="isChecked" /> <- modifie l'attribut checked
- select s'attache à l'attribut value de l'option
  <select v-model="selected"> <option>A</option></select> <- lorsque l'option sera choisie, selected sera égal à la valeur de l'option, soit A.

Gestion évènementielle

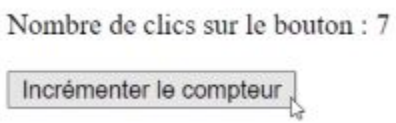
Utilisation d’event

Dans VueJS, et nous l’avons déjà fait, vous pouvez utiliser les directives.

En voici une nouvelle : v-on:nomevent.

Par exemple : v-on:click ou v-on:mousemove, etc

Exercice pour essayer : Réaliser l’affichage suivant dans un fichier exercice-03.html:



Je vous montrerai pour le clic sur le bouton

Utilisation d’event

```
<div id="app">
        <p>Nombre de clics sur le bouton : {{ compteur }}</p>
        <button v-on:click="compteur++">Incrémenter le compteur</button>
// on peut mettre @click plutôt que v-on:click
    </div>
    <script>
        app = Vue.createApp({
            data(){
                return {
                    compteur: 7
                }
            }
        });
        app.mount('#app');
```

Utilisation de l’objet event

Créons un fichier decouverte-05.html pour découvrir l’objet event.
Nous allons afficher les coordonnées du pointer lorsque l’on passe la souris sur une image.

Regardons ensemble.

Utilisation de l’objet event

```
<div id="app">
        <img
src="https://cdn.pixabay.com/photo/2022/09/18/14/23/baby-7463137_1280.jpg"
@mousemove="getCoords" />
        <div>{{ xy }}</div>
    </div>
    <script>
        app = Vue.createApp({

            data(){
                return {
                    xy: ''
                }
            },
            methods: {
```

Suffixe d’event

Chaque évènement peut être préfixé par un suffixe Javascript. Par exemple, on pourrait faire un @click.once pour n’exécuter qu’une fois l’event.
Je vous montre.

Suffixe d’event

Dans decouverte-05.html :

```
<div id="app">
        <button @click.once="action">Cliquez ici</button>
        <input type="text" @keyup.enter="action" />
    </div>
    <script>
        app = Vue.createApp({

            methods: {
                action(){
                    console.log("Je suis cliqué !")
                }
            }

        });
```

Suffixe d’event

Exercice dans exercice-04.html :

Affichez une image avec une width de 500px.
Au clic gauche sur l’image, passez la width à 300px.

Suffixe d’event

```
<div id="app">
        <img
src="https://cdn.pixabay.com/photo/2022/09/18/14/23/baby-7463137_1280.jp
g" v-bind:width="width" @click.left="updateWidth" />
    </div>
    <script>
        app = Vue.createApp({

            data(){
                return {
                    width: '300'
                }
            },
            methods: {
```

Les arguments dynamiques

Les arguments dynamiques

Les arguments d’un event peuvent être dynamiquements exprimés, je vous montre comment sur l’exercice 4

Les arguments dynamiques

Sur exercice 04 :

```
<div id="app">
        <select v-model="choix">
            <option value="click">Clic</option>
            <option value="dblclick">Double clic</option>
        </select>
        <img src="https://cdn.pixabay.com/photo/2022/12/03/15/59/pear-7632986_960_720.jpg"
v-bind:width="width" @[choix]="updateWidth" />
    </div>

    <script>
        app = Vue.createApp({

            data(){
                return{
                    width: 500,
```

Directive v-for

Directive v-for

v-for nous permet de boucler dans un tableau. Pour illustrer son utilisation, créons un fichier decouverte-0.html à partir du squelette.html

Je vous montre la suite.

Directive v-for

```
<div id="app">
        <p>Quels langages de prog connaissez vous ?</p>
        <input type="text" v-model="unLang" />
        <input type="button" value="valider" @click="ajouter" />
        <div>{{ langages }}</div>
    </div>
    <script>
        app = Vue.createApp({
            data(){
                return {
                    unLang:'',
                    langages: []
                }
            },
            methods: {
```

Directive v-for

Maintenant, plutôt que d’afficher tout le tableau langages, je vais vouloir afficher une liste à puce avec chaque langage, un par un.
Je vous montre.

Directive v-for

A la place de la <div{{ langages }}

Je met :

```
<ul>
    <li v-for="langage in langages">{{ langage }}</li>
</ul>
```

et si je veux qu'un langage s'ajoute lorsque j'appuie sur "entrer" ?

Je rajoute : `@keyup.enter="ajouter"` sur l'input

Directive v-once

Directive v-once

Nous pouvons nous retrouver dans le cas où nous souhaitons afficher les informations d'une donnée sans pour autant modifier l'affichage dynamiquement.

Par exemple, pour réaliser l'affichage suivant :

Valeur de départ : 15
Valeur finale : XX

<bouton pour incrémenter>

Directive v-once

```
<div id="app">
        <div v-once>Valeur de départ : {{ nombre }}</div>
        <div>Valeur finale : {{ nombre }}</div>
        <button type="button" @click="button++" >Incrémenter</button>
    </div>
    <script>
        app = Vue.createApp({
            data(){
                return {
                    nombre:15
                }
            }
        });
        app.mount('#app');
    </script>
```

Classes conditionnelles

Classes conditionnelles

Nous pouvons binder des classes dynamiquement à un élément HTML. Cela s'écrit :

```
<div :class="{d-none:isOK}">Une div</div>
```

Ainsi, si isOK est égal à false, la classe “d-none” ne sera pas appliquée. Si isOK est égal à true, la classe “d-none” sera appliquée.

Je vous montre un exemple.

Classes conditionnelles

```
<div id="app">
        <p>Quels langages de programmation connaissez-vous ?</p>
        <input type="text" v-model="unLang">
        <input type="button" value="valider" @click="ajouter">

        <ul>
            <li v-for="langage in langages">
                <input type="checkbox" v-model="langage.etatCase"> <span
:class="{rouge:langage.etatCase}">{{ langage.lang }}</span>
            </li>
        </ul>
    </div>
    <script>
        app = Vue.createApp({
            data(){
```

Les conditions

Les conditions

Il nous est possible d’exprimer des conditions sur nos éléments HTML avec v-if, v-else-if et v-else.

Je vous montre.

Les conditions

Reprenons le code précédent. Si la case est cochée, on veut afficher “le texte HTML est coché”, sinon, “le texte HTML est décoché”.
Si le langage est Javascript et qu’il est coché,, on affichera en plus : “mon langage préféré !”

Voici l’input à modifier :

```
<input type="checkbox" v-model="langage.etatCase"> Le texte {{ langage.lang }}
<span v-if="langage.etatCase && langage.lang == 'javascript'">est coché, mon langage préféré !</span>
<span v-else-if="langage.etatCase">est coché</span>
<span v-else>est décoché</span>
```

Fin de la première partie : découverte

A ce stade, vous savez :

Créer une application Vue.js 3

Définir des propriétés dans le modèle et y accéder dans la vue avec une {{ interpolation }}

Effectuer un binding bidirectionnel avec v-model et v-bind

Définir des methods au niveau de votre application

Utiliser v-html dans un élément pour l'html soit interprété :

```
<p>En utilisant la directive `v-html` : <span v-html="rawHtml"></span></p>
```

Gérer les events avec @event

Créer des boucles dans la vue avec v-for

Fin de la première partie : découverte

Utiliser des arguments dynamiques dans la vue avec [unArgument].
Exemple : @[eventDynamique] = “fonction”

Créer des classes conditionnelles avec :class=”uneClasse:unBooleen”

Fin de la première partie : TP 1

Il est temps d’appliquer tout ce que vous avez vu dans un projet de plus grande ampleur !

C’est l’heure du TP 1 !

TP1

Introduction aux composants Vue.js

Introduction aux composants Vue.js

Pour cette partie, nous allons construire un site web de location de jeux de sociétés.

Les petits exercices intermédiaires seront indépendants du site web.

Créez un dossier cours_composants et placez y un fichier site-v1.html à partir du squelette.

Ensuite, incorporez bootstrap (css et JS) via son CDN à votre fichier.

Introduction aux composants Vue.js

Les composants nous permettent de définir des blocs de code auxquels on va pouvoir donner des options pour les paramétrer.

Ces composants seront réutilisables.

Un de leur grand intérêt est de découper un code lourd en plusieurs petits composants légers qui s'imbriqueront parfaitement.

Je vous montre un exemple.

Introduction aux composants Vue.js

```
<div id="app">
        <menu-site></menu-site>
    </div>

    <script src="composants/menu.js"></script>

    <script>
        app = Vue.createApp({
            components: {
                "menu-site" : menu
            }
        });
        app.mount('#app');
    </script>
```

Introduction aux composants Vue.js

Puis, dans cours_composants/composants/menu.js :

```
const menu = {
    template : `collez la navbar bootstrap (entre backquotes). Pensez à nettoyer le menu si vous ne voulez pas tout afficher. Nous garderons un seul onglet du menu`
}
```

Expliquer ensuite qu'on peut ajouter des datas, des methods, etc à un composant

```
const menu = {
    template : `...`,
    methods: …,
    data() ….
}
```

Introduction aux composants Vue.js

Créez un fichier cours_composants/exercices/exercice-05.html :

Réaliser l’affichage suivant avec un composant :

Au clic sur le bouton, le nombre augmente de 1.

0

Introduction aux composants Vue.js

composants/compteur.js :

```
const compteur = {
    template: '<button type="button" @click="nombre++">{{ nombre }}</button>',
    data() {
        return {
            nombre: 0
        }
    }
}
```

fichier html :

```
app = Vue.createApp({
            components: {
```

Introduction aux composants Vue.js

Mode avec tout dans un ficher :

```
<div id="app">
        <increment-number></increment-number>
    </div>

    <script>

        app = Vue.createApp({
            components: {
                'increment-number' : {
                    data() {
                        return {
                            nombre: 0
                        }
```

Passer des données aux composants avec les props

Passer des données aux composants avec les props

Pour vous présenter l’utilité des props, créons un nouveau composant “Jeu”.

Passer des données aux composants avec les props

```
<div id="app">
        <menu-site></menu-site>
        <liste-jeux></liste-jeux>
    </div>

    <script src="composants/menu.js"></script>
    <script src="composants/liste_jeux.js"></script>

    <script>
        app = Vue.createApp({
            components: {
                "menu-site" : menu,
                "liste-jeux": liste_jeux
            }
        });
```

Passer des données aux composants avec les props

Fichier liste_jeux.js :

```
const liste_jeux = {
    template: `<div v-for="jeu in jeux">
        <p>{{ jeu.nom }}</p>
        <p>{{ jeu.description }}</p>
    </div>`,
    data(){
        return {
            jeux: [
                {
                    'nom': "Dixit",
                    'description': "Un jeu avec des cartes qui ne veulent rien
dire et qui vont pourtant nous servir à communiquer"
                },
```

Passer des données aux composants avec les props

C’est un peu moche, utilisons une card bootstrap et mettons le composant liste_jeux dans un container, ce sera mieux :

```
const liste_jeux = {
    template: `<section class=”d-flex mt-5”>
        <div class="card" style="width: 18rem;" v-for="jeu in jeux">
            <div class="card-body">
                <h5 class="card-title">{{ jeu.nom }}</h5>
                <p class="card-text">{{ jeu.description }}</p>
            </div>
        </div>
    </section>`,

     // etc
}
```

Passer des données aux composants avec les props

Imaginons que nous souhaitons une liste de jeu avec un titre de liste qui varie dynamiquement.

C’est possible avec les props

Passer des données aux composants avec les props

Modification de l'appel du composant :

```
<liste-jeux titre="Nos derniers jeux"></liste-jeux>
```

Modification du composant :

```
template: `<h2>{{ titre }}</h2>
    <section style="display:flex;" class="mt-4">
        <div class="card" style="width: 18rem;" v-for="jeu in jeux">
            <div class="card-body">
                <h5 class="card-title">{{ jeu.nom }}</h5>
                <p class="card-text">{{ jeu.description }}</p>
            </div>
        </div>
    </section>`,
```

Passer des données aux composants avec les props

Notons qu'il est aussi possible de récupérer une prop dans le modèle avec this.nomdelaprop :

```
props:['titre'],
data(){
    return {
        unTitre: this.titre
    }
}
```

Passer des données de l’app vers les composants

Passer des données de l’app vers les composants

Dans l’exemple précédent, c’est le composant list-jeux qui déclarait toutes les données qu’il utilisait.

Mais que se passerait-il si l’on voulait que ce composant <dont le rôle est d’afficher une liste de jeux à partir d’un tableau d’objets de jeux> affiche une autre liste de jeux.

Egalement, les données utilisées par l’application proviendront généralement d’un appel à une API effectué par l’application, non par un composant.

Nous avons donc besoin de passer les données (notre liste de jeux) de l’application vers le composant liste_jeux.

Passer des données aux composants avec les props

Modifiez la ligne du composant :

```
<liste-jeux titre="Nos derniers jeux" :listjeux="jeux"></liste-jeux>
```

Ajoutez les data à l'application :

```
app = Vue.createApp({
            data(){
                return {
                    jeux: [
                        {
                            'nom': "Dixit",
                            'description': "Un jeu avec des cartes qui ne
veulent rien dire et qui vont pourtant nous servir à communiquer"
                        },
                        {
```

Passer des données aux composants avec les props

Modifiez le composant dans liste_jeux.js :

```
const liste_jeux = {
    template: `<h2>{{ titre }}</h2>
    <section style="display:flex;" class="mt-4">
        <div class="card" style="width: 18rem;" v-for="jeu in listjeux">
            <div class="card-body">
                <h5 class="card-title">{{ jeu.nom }}</h5>
                <p class="card-text">{{ jeu.description }}</p>
            </div>
        </div>
    </section>`,
    props:[‘titre’,'listjeux']
}
```

Créer des événements custom

Créer des événements custom

Nous cherchons maintenant à créer un bouton “checker disponibilité” pour chacun de nos jeux.

Il nous faudra un événement click sur un bouton qui déclenchera une fonction checkDisponibilite.

Jusque là, rien de neuf, faisons le ensemble.

Créer des événements custom

Ajout d’un bouton sous le paragraphe dans composants/liste_jeux.js :

```
<button class="btn btn-sm btn-primary" @click="checkDisponibilite"
type="button">Checker disponibilité</button>
```

Ajout de l’event dans composants/liste_jeux.js :

```
methods:{
        checkDisponibilite(){
            alert('je check la dispo');
        }
    }
```

Créer des événements custom

Cela fonctionne parfaitement lorsque la fonction appelée est déclarée au niveau du composant qui l'appelle.

Si l'on essaie de déclencher à partir d'un composant une méthode globale (située dans l'application), cela ne fonctionnera pas.

Voyons cela en déplaçant la method checkDisponibilite() du composant vers l'application.

Créer des événements custom

Pour rendre possible le déclenchement de methodes de l’application dans un composant, nous devons créer un événement customisé à partir du composant enfant grâce à la méthode $emit().
Je vous montre.

Créer des événements custom

Modifiez le bouton :

```
<button class="btn btn-sm btn-primary" @click="$emit('checkdispo')"
type="button">Checker disponibilité</button>
```

Puis appelez l'événement customisé à partir du composant :

```
<liste-jeux titre="Nos derniers jeux" :listjeux="jeux"
@checkdispo="checkDisponibilite" ></liste-jeux>
```

Exercice

Dans le fichier exercice-06.html :

Réalisez l’affichage suivant : appelez trois fois le composant compteur.
Au clic sur un bouton, le chiffre augmente en partant de zero.

A droite des boutons, la somme des compteurs sera affichée.

Exercice

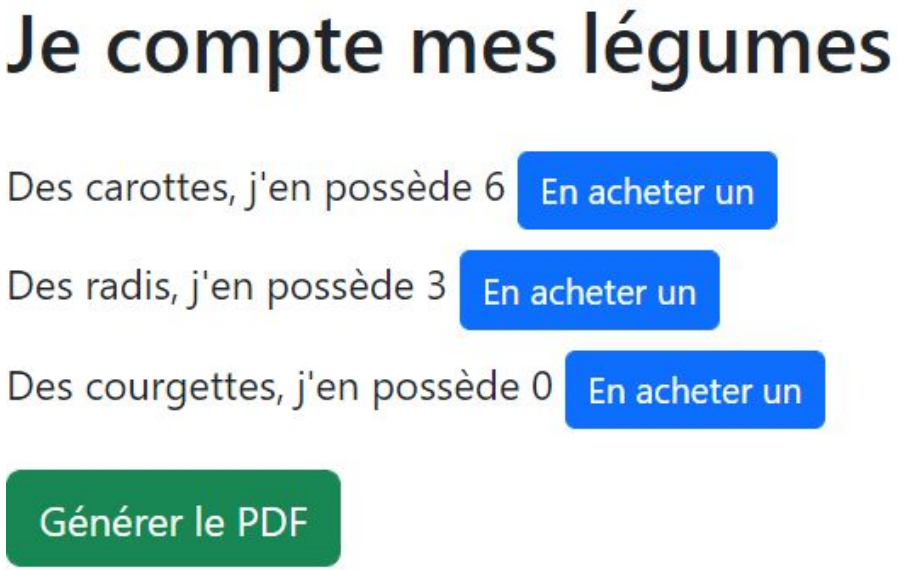
```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8"/>

    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>

    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <title>Exercice 6</title>
    <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>

    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"
rel="stylesheet"

integrity="sha384-EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOmLASj
C" crossorigin="anonymous" />
```

fichier composant

```
const incrementalButton = {
    template: `<button
@click="valueButton++;$emit('add')">{{valueButton}}</button>`,
    data() {
        return {
            valueButton: 0,
        }
    }
}
```

Exercice un peu plus costaud

Voici l'affichage à réaliser dans exercice-06.html :



En tant que codeur qui codez propre, vous ferez évidemment un composant pour la liste des légumes.

Le bouton "En acheter un" augmente le compteur.

Le bouton "générer le PDF" fait une alerte "PDF généré !"

Exercice un peu plus costaud

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exo 6</title>
    <script src="http://unpkg.com/vue@next"></script>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css"
rel="stylesheet"
integrity="sha384-Zenh87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeuOxjzrPF/et3URy9Bv1
WTRi" crossorigin="anonymous">
</head>
<body>
```

Exercice un peu plus costaud

```
const liste_legumes = {
    template: `
    <div v-for="legume in legumes" class="mt-2">
        Des {{ legume.nom }}, j'en possède {{ legume.quantite }}
        <button type="button" class="btn btn-sm btn-primary"
@click="legume.quantite++">En acheter un</button>
    </div>
    <button type="button" class="mt-3 btn btn-success"
@click="$emit('genererpdf')">Générer le PDF</button>`,
    props:['legumes']
}
```

Faire un call API et utiliser un hook

Faire un call API et utiliser un hook

Regardons comment faire pour réaliser un call vers une API sous Vue.js

Revenons dans le cas de notre site internet. Dupliquer le fichier site-v1.html pour créer site-v2.html.
Dans cette v2, nous allons vouloir afficher une liste de loueurs de jeux de sociétés en dessous de la liste des jeux.

La liste de nos faux loueurs de jeux proviendra de l'API randomuser.me

Faire un call API et utiliser un hook

A vous de le faire :

1. créer un composant liste-users dans site_v2.html
2. créer un fichier liste_users.js (gardez l'affichage sous forme de cards pour l'instant)
3. la liste des users sera fournie au composant par l'application pour le moment, users sera un tableau vide
4. affichez le composant liste-users sous le composant liste-jeux en lui passant le titre "Les derniers loueurs inscrits"

Faire un call API et utiliser un hook

Création du nouveau composant dans notre app :

```
"liste-users": liste_users
```

Création du fichier liste_users.js (ne pas oublier de le link)

```
const liste_users = {
    template: `<h2>{{ titre }}</h2>
    <section style="display:flex;" class="mt-4">
        <div class="card" style="width: 18rem;" v-for="user in listusers">
            <div class="card-body">
                <h5 class="card-title"></h5>
                <p class="card-text"></p>
            </div>
        </div>
    </section>`,
    props:['listusers' 'titre']
```

Faire un call API et utiliser un hook

Sous la liste de jeux :

```
<liste-users titre="Les derniers loueurs inscrits"
:listusers="users"></liste-users>
```

// on fait la suite : aller chercher de faux users

-> Allons sur le site axios pour récupérer le cdn et voir comment ça fonctionne

L'incorporer à votre code :

```
users: axios.get(adresse).then(function(reponse){
      console.log(reponse);
})
```

-> Allons sur le site random user api -> documentation -> how to use

Remplacer "adresse" par https://randomuser.me/api/?results=3

-> Regarder la console

Faire un call API et utiliser un hook

On voit que dans reponse.data.results, on retrouve un tableau d'objet JSON, parfait pour que notre v-for boucle dedans ensuite !

```
users:
axios.get('https://randomuser.me/api/?results=3').then(function(reponse){
    return reponse.data.results
})
```

Malheureusement, ça ne fonctionnera pas car le temps que notre app récupère les données de l'API (quelques millisecondes), notre app sera déjà créée et montée.

Faire un call API et utiliser un hook

Nous devons donc créer un hook qui se lancera une fois que notre app sera créée :

Dans data() de notre app, users devient un tableau vide :

```
users: []
```

Nous créons le hook entre data et component par exemple :

```
created(){
    axios.get('https://randomuser.me/api/?results=3').then(function(reponse){
        vm.users = reponse.data.results
    })
},
```

Nous spécifions que la liste des users est stocké dans la vue montée pour pouvoir y accéder : `let vm = app.mount('#app');`

Faire un call API et utiliser un hook

Modifiions nos cards pour avoir un plus bel affichage dans composants/liste_users.js :

```
const liste_users = {
    template: `<h2>{{ titre }}</h2>
    <section style="display:flex;" class="mt-4">
        <div class="card" style="width: 18rem;" v-for="user in listusers">
            <img :src="user.picture.large" class="card-img-top" />
            <div class="card-body">
                <h5 class="card-title">{{ user.name.first }} {{ user.name.last
}}</h5>
                <p class="card-text">De {{ user.location.city }}</p>
            </div>
        </div>
    </section>`,
    props:['listusers','titre']
}
```

Faire un call API et utiliser un hook

Petit bonus : la liste des users est trop collée… mais je ne veux pas mettre un margin-top sur le <h2> de la liste des users car que se passerait-il si je n'en voulais pas ? Pour rendre cela dynamique, utilisons le component :

```
<liste-users titre="Les derniers loueurs inscrits" :listusers="users"
margin_top="mt-5"></liste-users>
```

puis

```
<h2 :class="margin_top">{{ titre }}</h2>
```

puis

```
props:['listusers','titre','margin_top']
```

Les slots

Les slots

Nous souhaitons réaliser l'affichage suivant :

Nous pourrions passer par une prop… mais elle serait longue et, en plus, le html ne serait pas interprété.

Nous pourrions sinon écrire le code du texte dans le composant liste_jeux mais cela reviendrait à toujours afficher ce texte dès que l'on liste des jeux.

Les slots

Utilisons les slots :

```
<liste-jeux titre="Nos derniers jeux" :listjeux="jeux" @checkdispo="checkDisponibilite">
      <p>Retrouvez nos derniers jeux de
      <span class="text-primary">stratégie</span>,
      <span class="text-danger">réflexion</span> ou
      <span class="text-success">d'ambiance</span>.</p>
</liste-jeux>
```

Et dans liste_jeux.js, en dessous du <h2> :

```
<slot></slot>
```

Les slots nommés

Plaçons nous maintenant dans le cas où nous voulons deux slots dans notre composant… nous sommes embêtés.
Heureusement, Vue.js a prévu les slots nommés, je vous montre.

Les slots nommés

```
<liste-jeux titre="Nos derniers jeux" :listjeux="jeux" @checkdispo="checkDisponibilite">
                <template v-slot:presentation>
                    <p>Retrouvez nos derniers jeux de
                        <span class="text-primary">stratégie</span>,
                        <span class="text-danger">réflexion</span> ou
                        <span class="text-success">d'ambiance</span>.
                    </p>
                </template>
                <template v-slot:phrase_finale> <- peut s’écrire #phrase_finale
                    <p>N'hésitez pas à <a href="#">proposer de nouveaux jeux</a>.</p>
                </template>
            </liste-jeux>
```

Et dans liste_jeux.js :

```
<slot name="presentation"></slot>
```

Les slots portés

Dans les deux cas précédents, nous envoyons des données de notre vue vers notre composant grâce aux slots.

Nous pourrions avoir besoin de récupérer les données du composant pour un traitement dans la vue.

Exemple pour illustrer : comment faire pour avoir le nom des jeux en gras si je le désire ?
Je vous montre

Les slots portés

Dans le composant :

```
<h5 class="card-title"><slot name="jeunom" :nomjeu="jeu.nom"></slot></h5>
```

Dans la vue :

```
<template v-slot:jeunom="slotProps">
     <i>{{ slotProps.nomjeu }}</i>
</template>
```

Les propriétés calculées pour filtrer

Les propriétés calculées

Les filtres n’existent plus depuis Vue.js 3. Il faut utiliser les propriétés calculées pour appliquer des filtres à nos éléments.
Exemple avec une propriété calculée qui modifierait l’affichage des titres de nos jeux pour LeS rEnDrEs CoMmE çA (c’est pas beau, c’est juste pour vous montrer !)

Les propriétés calculées

Dans liste_jeux.js, modifiez le h2 :

```
<h2>{{ capitalizeBizarre }}</h2>
```

Puis ajoutez la propriété calculée :

```
computed:{
        capitalizeBizarre(){
            const lettre = this.titre.split('');
            for(i=0;i<lettre.length;i++){

                if(i%2 == 0){
                    lettre[i] = lettre[i].toUpperCase();
                }

            }
            return lettre.join('');
```

Les watchers

Les watchers

Les watchers ou observateurs en français, vont nous permettre d'exécuter du code lorsqu'une valeur du modèle change.

Pour illustrer le watcher, nous allons créer un input text au niveau de notre liste d'utilisateurs qui sera censé afficher les utilisateurs selon le texte inséré (nous ne coderons pas cette fonctionnalité). Si un mot interdit est tapé, l'input sera vidé (c'est ça que nous coderons).

Les watchers

Nous voulons un input pour notre liste d'user. Mauvaise pratique : le mettre dans le composant car cela le rendrait systématique. On va donc faire un slot :

```
<liste-users titre="Les derniers loueurs inscrits" :listusers="users"
margin_top="mt-5">
                <template #rechercheUser>
                    <input type="text" v-model="rechercheUserText" />
                </template>
</liste-users>
```

Et dans liste_users.js, sous le h2 :

```
<slot name="rechercheUser"></slot>
```

Les watchers

On ajoutera la propriété rechercheUserText dans le modèle puis le watcher :

```
watch:{
                rechercheUserText(nvlleValeur){
                    if(nvlleValeur.indexOf('interdit') != -1){
                        this.rechercheUserText = '';
                    }
                }
            }
```

Transitions CSS

Transitions CSS

Vue.js intègre un gestionnaire de transition très efficace. Nous allons découvrir comment créer un bouton “lire la suite” qui affichera un texte ou le cachera.

Transitions CSS

Dupliquez site-v2 vers site-v3.

Sous la liste des utilisateurs, mettez un h2 : `<h2 class="mt-5">Notre histoire</h2>`

Dans un paragraphe, générez du texte via cupcake ipsum par exemple.

Ajoutez un bouton `<button @click="lireLaSuite" class="btn btn-sm btn-secondary">Afficher la suite</button>`

Créez un attribut dans le model : lireSuiteHistoire:false (car pas visible au début)

Transitions CSS

Puis créez la méthode :

```
lireLaSuite(){
                    if(this.lireSuiteHistoire){
                        this.lireSuiteHistoire = false;
                    }else{
                        this.lireSuiteHistoire = true;
                    }
}
```

Créez la transition (voir la doc) :

```
<Transition name="t">
     <div v-if="lireSuiteHistoire">
            <p> Un texte généré via cupcake</p>
      </div>
</Transition>
```

Transitions CSS

Enfin, créez le css :

```
<style>
        .t-enter-from, .t-leave-to {
            opacity: 0;
        }
        .t-enter-active, .t-leave-active {
            transition: opacity;
        }
    </style>
```

Testez.

Transitions CSS

Vous pouvez améliorer le bouton :

```
<button @click="lireLaSuite" class="btn btn-sm btn-secondary">
                <span v-if="lireSuiteHistoire">Cacher</span>
                <span v-else>Afficher</span>
                la suite
            </button>
```

Vous pouvez écrire le click différemment (etat A, etat B) :

```
<button @click="lireSuiteHistoire=!lireSuiteHistoire" class="btn btn-sm
btn-secondary">
```

Transitions dynamiques avec animate.css

Transitions dynamiques avec animate.css

Plutôt que des transitions faites à la main, nous pouvons également utiliser une librairie qui se configure facilement avec Vue.js : animate.css

Allons voir la documentation ensemble

Transitions dynamiques avec animate.css

Il nous suffira de mettre le cdn d’animate.css

Puis d’écrire la transition comme suit (refaisons la transition de lire la suite) :

```
<Transition enter-active-class="animate__animated animate__bounceInLeft"
leave-active-class="animate__animated animate__bounceOutRight">
```

TP2

Création d’une SPA avec Vue CLI

Création d’une SPA avec Vue CLI

Une SPA (Single Page Application) n’est pas un site web one page.

Il s’agit d’une application qui ne charge qu'un seul document web, puis met à jour le contenu du corps de ce document.

Ainsi, une seule page est chargée = pas de chargement de page pendant la navigation.

Création d’une SPA avec Vue CLI

Nous avons jusqu’ici créé nos applications Vue.js en utilisant le CDN et en utilisant une arborescence maison, ce qui est possible dans le cas de petits projets.

Cela nous a également servi à aborder les fondamentaux de Vue.js

Maintenant que les v-bind, les v-model, les events, les customs events

les composants, les props, les hooks, les methods, les computed et les calls API n’ont plus de secret pour vous, tout comme les slots, intéressons nous à la création de projet Vue.js sous architecture professionnelle… on fera la même chose, mais nos fichiers seront rangés différemment.

Création d’une SPA avec Vue CLI

Allons voir la doc de Vue Cli : https://cli.vuejs.org/guide/installation.html

Attention, vous devez d’abord vous assurer que node soit à jour (version > 10 pour vue Cli mais version > 16 pour vue js 3).
Faites un node -v pour le savoir.

Création d’une SPA avec Vue CLI

Si problème pour installer node avec linux :
https://github.com/nodejs/help/wiki/Installation

Si problème avec le proxy pour le npm install vue cli :
Il faut connaître l’adresse du proxy et faire un : npm config set proxy <adresse>

Création d’une SPA avec Vue CLI

npm install -g @vue/cli

puis

vue create cours_spa -> choisir vue 3

ensuite, un cd cours_spa puis un npm run serve

Votre app est dispo sur localhost:8080

Installer Vue Router

Installer Vue Router

Comme nous l'avons vu précédemment, le terme « Applications monopages » (SPA) indique qu'une seule page est chargée. Cette dernière est ensuite mise à jour dynamiquement en fonction des besoins.

Bien que cela soit puissant, il y a tout de même un défaut principal : les utilisateurs ont besoin d'URL différentes pour distinguer les différentes pages les unes des autres. Sinon, à chaque fois qu'un utilisateur essaie de revenir en arrière, il obtiendra toujours la même page !

Installer Vue Router

Installons Vue Router : vue add router

A la demande d'installation d'history mode, répondons "non". L'history mode est une configuration qui permet diverses opérations sur les urls dont nous n'aurons pas besoin, la version de base étant suffisante.

Plusieurs fichiers ont été modifiés :

main.js -> router importé et configuré

App.vue -> la logique de notre app a été automatiquement déplacée dans des views.

Arborescence de notre application

Arborescence de notre application

public/index.html : le fichier qui contient notre app qui sera montée. Les fichiers JS seront automatiquement injectés dedans.

src/assets : dans ce dossier, nous mettrons toutes les images et fichiers js et css de notre application. Ils sont gérés par WebPack.

components : dossier qui contiendra les composants (donc les pages) de l'application.

router/index.js : la configuration de notre router. Il importe les views et se charge de la configuration de la navigation.

views : toutes les vues de l'application sont ici.

app.vue : c'est LE fichier qui configure notre application et qui appelle les différents composants à utiliser.

Arborescence de notre application

main.js : le point d’entrée de notre application qui est injecté dans index.html. C’est ce fichier qui indique comment monter l’application

A quoi vous fait penser cette arborescence ?

A l’architecture M V C !
Nous venons de mettre en place quelque chose qui ressemble au MVC au niveau de notre frontend, pas mal ! 😍

Dans Vue.js, c’est Le MVVM

Arborescence de notre application

Si vous naviguez de la page Home à la page About, vous verrez qu'un diese s'intercale dans l'url. Cela vient du mode history du router.

Rdv dans router/index.js pour changer : createWebHashHistory en createWebHistory

(dans l'import ligne 1 et dans la config history ligne 21)

Principe de base de l’archi MVVM

Le MVVM sous Vue.js fonctionne ainsi—---> une requête arrive !

-> public/index.html est chargé

-> mains.js est lu : il importe les dépendances, utilise le router et monte l’app

-> App.vue est lu : il déclare obligatoirement <router-view/> car c’est dans ce composant hérité de Vue Router que viendra se placer le contenu de la page demandée par le visiteur.

-> le router est utilisé via son fichier router/index.js : création des routes et association des composants

-> la page demandée est affichée par <router-view> : le router sait quelle page charger

-> (optionnellement) les composants sont chargés et appelés par la vue

Exercice

Afficher le tableau suivant.
Précision : les data se mettent dans la vue, pas dans le composant.
Précision2 : le v-for s'écrira ainsi (explications à la correction) :

```
<tr v-for="personne in list_inscrits" :key="personne">
```

Correction

Voir le repo github associé au cours

Le call API dans une SPA

Le call API dans une SPA

Rien d’exceptionnellement nouveau, le call API dans notre SPA se fera via axios. La vue est responsable des données, elle passera les users au composant qui fera l’affichage.

Je vous montre.

Le call API dans une SPA

Pour installer axios : npm install axios

Dans InscritsList.vue :

```
data() {
    return {
      personnes: []
    }
  },
  methods: {
    // Récupère 3 utilisateurs via l'API Random Users
    async fetchUsers() {
      const userResponse = await
axios.get('https://randomuser.me/api/?results=3');
      this.personnes = userResponse.data.results;
    }
```

Le call API dans une SPA

Dans le composant InscritsList :

```
<tbody>
      <tr v-for="personne in list_inscrits" :key="personne">
        <td>{{ personne.name.first }}</td>
        <td>{{ personne.name.last }}</td>
        <td>{{ personne.registered.age }}</td>
      </tr>
    </tbody>
```

Utiliser le router

Utiliser le router

Le router est un composant essentiel de Vue.

Il permet de faire des liens de page en page, mais également de poster des formulaires en GET… regardons ensemble.

Utiliser le router

Ce simple code enverra vers la route /rechercher/slug au clic sur le bouton.

```
<input type="text" placeholder="Votre nom" v-model="nom"/>

<router-link :to="`/rechercher/${nom}`">
 <button type="submit">Rechercher</button>
</router-link>
```

Côté router, comment faire pour récupérer cette route ?

```
{
 path: "/rechercher/:nom",
 name: "rechercher",
 component: () => import("../views/MaVue.vue"),
}
```

L’anti-pattern Vue.js

L’anti-pattern Vue.js

Pour comprendre l’anti-pattern de Vue.js, vous allez réaliser un exercice plutôt insolite : l’objectif sera de faire apparaître une erreur.

Nous voulons réaliser l’habituel bouton compteur : au clic sur le bouton, la valeur est multipliée par deux.

1 : créez une nouvelle route “Compteur” et ajoutez la au menu

2 : créez votre vue et le composant qui affichera un bouton

3 : la vue passe la valeur du compteur via une props au composant

4 : le composant utilise la props et l’affiche dans le bouton

5 : créez un événement incremente dans votre composant : au clic sur le bouton, la valeur du compteur est multipliée par deux

Message d’erreur à obtenir : Unexpected mutation of "val" prop

L’anti-pattern Vue.js

Votre vue :

```
<template>
    <ButtonCompteur :val="val" />
</template>

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

export default {
    name: 'CompteurDisplay',
    components: {
        ButtonCompteur
    },
    data(){
```

L’anti-pattern Vue.js

Votre composant ::

```
<template>
  <button type="button" @click="incremente">{{ val }}</button>
</template>

<script>
export default {
  name: 'ButtonCompteur',
  props: {
    val: Number
  },
  methods:{
    incremente(){
      this.val = this.val * 2 ;
```

L’anti-pattern Vue.js

Vue.js considère le fait de modifier la valeur d’une prop passée à un composant comme une opération interdite.
La valeur de la prop doit restée “intacte”.

Pour qu’un composant Vue.js puisse récupérer la valeur d’une prop et la modifier, il faut qu’il la clone dans ses datas.
Ainsi, le composant agira non sur la prop mais sur ses propres datas

L’anti-pattern Vue.js

Ajoutez ce code à composant :

```
data(){
    return{
      compteur:this.val
    }
  },
```

Les mixins

Les mixins

Pour illustrer le principe de mixin, imaginons que nous voulons envoyer un mail à deux endroits de notre super site :

- si le compteur dépasse 30 : on envoie un mail
- si on clic sur le bouton envoyer l'email d'inscription (que l'on va rajouter) sur la liste des utilisateurs : on envoie un mail

Créons une fonction sendMail() qui fera un alert("Email envoyé"); dans le composant InscritsList et ButtonCompteur

Les mixins

- Dans ButtonCompteur, ça donne :

```
methods:{
    incremente(){
      this.compteur = this.compteur * 2;
      if(this.compteur > 30){
        this.sendMail();
      }
    },
    sendMail(){
      alert('Email envoyé !');
    }
  }
```

Les mixins

Le problème d’un code non DRY (Don’t Repeat Yourself) est évident : la méthode d’envoie d’email se répète une fois de trop.

Les mixins offrent une manière flexible de créer des fonctionnalités réutilisables pour les composants de Vue.

Créons un dossier mixins et un fichier sendMail.js. Je vous montre.

Les mixins

Le problème d’un code non DRY (Don’t Repeat Yourself) est évident : la méthode d’envoie d’email se répète une fois de trop.

Créons un dossier mixins et un fichier sendMail.js

Les mixins

Dans mixins/sendMail.js

```
export default {
    methods: {
      sendMail() {
        alert('Email envoyé');
      }
    }
  };
```

Les mixins

Dans vos composants :

```
import sendMail from "@/mixins/sendMail";

export default {
  name: 'InscritsList',
  props: {
    list_inscrits: Array
  },
  mixins: [sendMail]
}
```

Les modes et variables globales

Les modes

Pour finir, intéressons nous aux deux modes d’environnements : developpement et production.

Le mode développement : npm run serve … nous le connaissons, c’est pour le dev.

En production, on ne veut pas que d’éventuelles erreurs s’affichent.. on préfèrera une belle page.

Créeons un composant NotFound avec une simple phrase.

Les modes

```
<template>
    <h1>Oups, vous avez trouvé les coulisses !</h1>
</template>

<script>

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

Les modes

Dans index.js (router) :

```
import NotFound from '../components/NotFound.vue'
```

Ajouter au router :

```
  {
    path: '/404', name: 'NotFound', component: NotFound
  },
  {
    path: '/:catchAll(.*)', redirect: '404'
  }
```

Les modes

Le mode production de Vue se génère grâce à : npm run build

Cela créera un dossier dist/

Pour mettre notre site en ligne, il suffira uniquement de mettre le contenu de dist/ sur un serveur web (dans un dossier www ou htdocs généralement) pour le rendre live !

Variables globales

En ce qui concerne les variables globales, l'utilisation est la suivante :

Modifiez votre main.js : `createApp(App).use(router).mount('#app')`

Devient :

```
const app = createApp(App)
app.use(router).mount('#app')
```

Ensuite, il suffit de déclarer les globales :

```
app.config.globalProperties.$prenom = "Bob"
```

Et d'y accéder : `this.$prenom`

Comprendre Vite

Vers Vite

Aujourd'hui, Vue3 bascule vers une utilisation totale et native de Vite.

Pour comprendre ce que fait Vite, il faut d'abord connaître la notion d'ESM.

Pendant longtemps, c'est Webpack qui a été LE favori indiscutable pour build des applications modernes en JavaScript/TypeScript.

Webpack peut intelligemment lire le point d'entrée de l'app (main.js par exemple) puis importer les bibliothèques, les images, les assets, le code, etc et les packager en un fichier servi au navigateur.

Mais depuis peu, une super alternative du nom d'esbuild est apparue. Contrairement à Webpack, esbuild n'est pas écrit en JS mais en Go. Il est 10 à 100 fois plus rapide que WebPack.

Pourquoi ne pas utiliser esbuild plutôt que WebPack ?

Vers Vite

Toutefois, esbuild ne produit pas de fichier lisible par le navigateur comme Webpack. esbuild produit des fichiers ECMAScript natifs (appelés ESM natifs).

En 2018, Firefox a officiellement supporté les ESM natifs puis, en 2019, les autres navigateurs ont suivi. A ce moment là, Evan You, créateur de Vue, était en train de développer Vue 3.

Il a alors développé un outil qui servirait rapidement les fichiers au navigateur grâce au format ESM via esbuild dans un projet Vue.js.

Cet outil, c'est Vite. 

Vite

Maintenant, lorsque nous chargeons une page dans un navigateur, nous ne chargeons pas un seul gros fichier JS contenant toute l'application : nous chargeons juste les quelques ESM nécessaires pour cette page, chacun dans leur propre fichier (et chacun dans leur propre requête HTTP).

Si un ESM a des imports, alors le navigateur demande à Vite ces fichiers également.

Vite est donc principalement un serveur de développement, chargé de répondre aux requêtes du navigateur, en lui envoyant les ESM demandés.

Les avantages de Vite pour le dev

Si nous changeons quelque chose dans un fichier, alors Vite n'envoie que le module qui a changé au navigateur, au lieu d'avoir à reconstruire toute l'application comme le font les outils basés sur Webpack !

Si l'app utilise une bibliothèque avec des tonnes de fichiers, Vite "pré-bundle" cette bibliothèque en un seul fichier grâce à esbuild et l'envoie au navigateur en une seule requête plutôt que quelques dizaines/centaines.

Cette tâche est faite une seule fois au démarrage du serveur, nous n'avons donc pas à payer le coût à chaque fois que nous rafraîchissons la page.

Vite n'est pas vraiment lié à Vue : il peut être utilisé avec Svelte, React, etc …
Bon pour nous, ce sera avec Vue..

Un oeil sur le futur

L’actuel et le futur

Aujourd’hui, la grande majorité des projets Vue.js sont fait avec Vue 2.
Quelques uns avec Vue 3 et Options API.

Très peu avec Vue 3 / Vite et Composition API, car c’est nouveau… mais certainement le futur de Vue.

Allons voir la doc de vuejs, et cliquer sur le point d’interrogation en haut à gauche, à côté de “Composition”

L’actuel et le futur

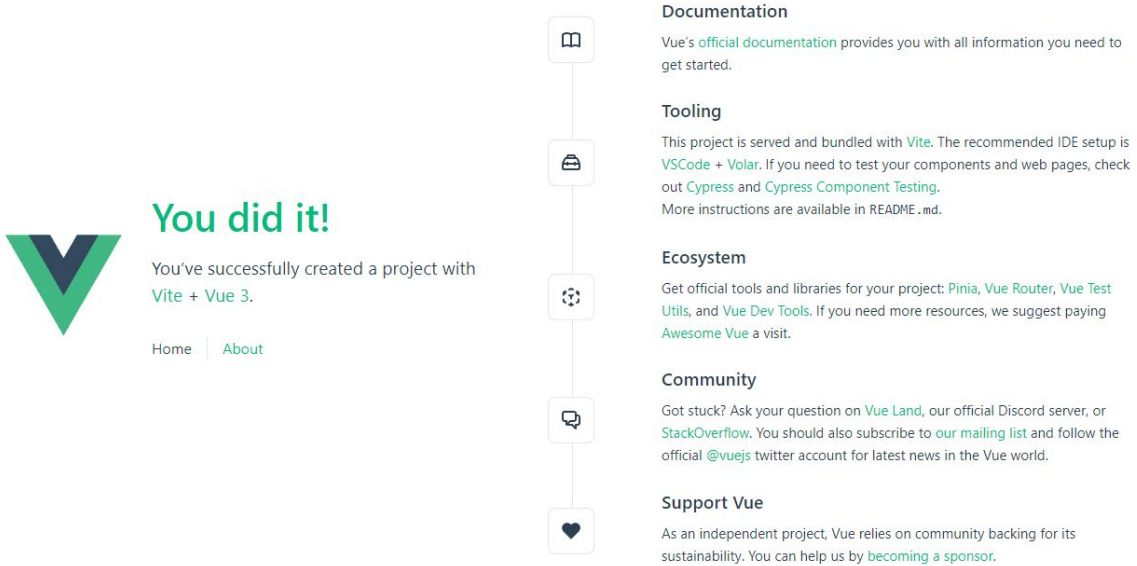
Un composant est présenté en mode Options API et en mode Composition API.
Explications.

Options API = plus simple à prendre en main car proche de la POO
Composition API = plus complexe car il faut être à l’aise avec Vue et avoir déjà codé avec pour comprendre le système de réactivité.

Les bonnes pratiques

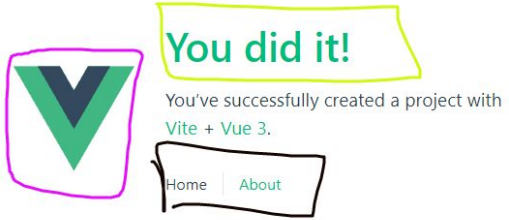
1 - Penser composants et sous composants

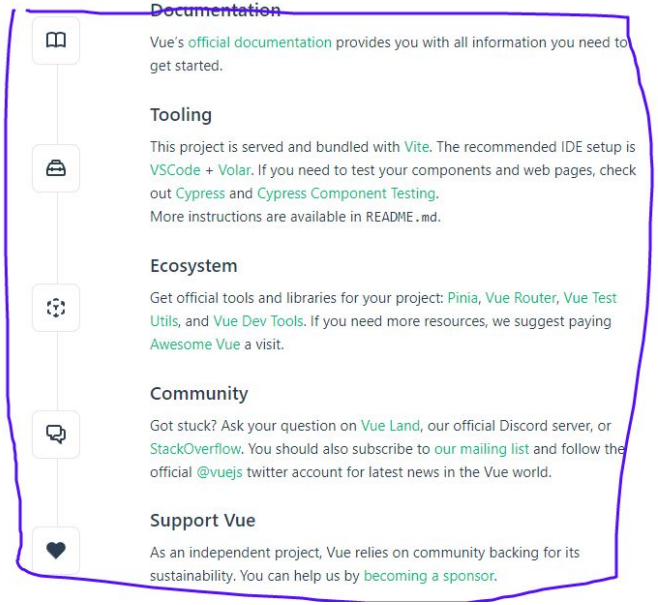
Je vais vous passer cette image et vous demander de dessiner où sont les composants de cette page.



1 - Penser composants et sous composants

Un peu comme ça par exemple. Ensuite, quand je vous le dirai (pas avant), tout le monde postera son dessin en même temps.





1 - Penser composants et sous composants

Pour voir comment cette page doit être pensée, créons un projet Vite.
Rdv sur la doc de Vue.

Lançons le projet et regardons comment il est fait.

2 - Mots multiples

N’oubliez pas que vos composants doivent être à mots multiples.
ClientListe

Seules exceptions à la règle : le composant racine App et les composants natifs comme <Transition>.

Ceci afin de prévenir les conflits avec des éléments HTML futurs ou existant car toutes les balises HTML n’ont qu’un seul mot.

3 - Détaillez vos props

Pas bien :

```
props: ['status']
```

Bien :

```
props: {
  status: String,
  required: true
}
```

4 - Des clés pour v-for

key avec v-for est toujours requis sur les composants afin de maintenir l'état des composants internes. Même pour les éléments, c'est une bonne pratique pour garder un comportement prédictible pour de la stabilité d'objet dans les animations.

```
<ul>
 <li v-for="todo in todos" :key="todo.id">
   {{ todo.text }}
 </li>
</ul>
```

5 - Pas de v-if avec v-for

On aimerait faire : v-for="user in users" v-if="user.isActive"

Mais on ferait mieux de faire .. ?

Une computed au lieu de users : v-for="user in usersActifs"

6 - Faites du scss avec Vue

Plutôt que du CSS, n’hésitez pas à faire du SCSS si vous êtes à l’aise.

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

Stockez vos variables dans un fichier variables.scss par exemple et utilisez le :

```
@import '@/assets/scss/variables.scss' ;
```

7 - Utilisez des librairies CSS

Bootstrap s’installe facilement :

```
npm install bootstrap
```

puis, dans main.js :

```
import 'bootstrap'
import 'bootstrap/dist/css/bootstrap.min.css'
```

Et pourquoi ne pas utiliser “bootstrap vue” directement.

Testez le sur vos prochains projets (bootstrap vue dans google).

8 - Importez vos données globales

Une clé API à utiliser dans 2 vues différentes ? Pas de problèmes.

Non :

```
let key = "56456d4qs3d4qsd32qsdda"
```

Oui :

```
let key = import.meta.env.API_MACHIN_KEY;
```

Et dans un fichier .env à la racine du site :

```
API_MACHIN_KEY = 56456d4qs3d4qsd32qsdda
```

TP3

THE END