Vue.js

M2i

Alan Piron-Lafleur

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-04.html :

Réaliser l'affichage suivant avec un composant :

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

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-05.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

Autre écriture possible :

```
components:{
        'compteur': {
            template : `<button @click="ajouterUn">{{ compteur }}</button>`,
            data(){
                return{
                    compteur:0
                }
            },
            methods:{
                ajouterUn(){
                    this.compteur++;
                    this.$emit('calculer')
                }
            }
```

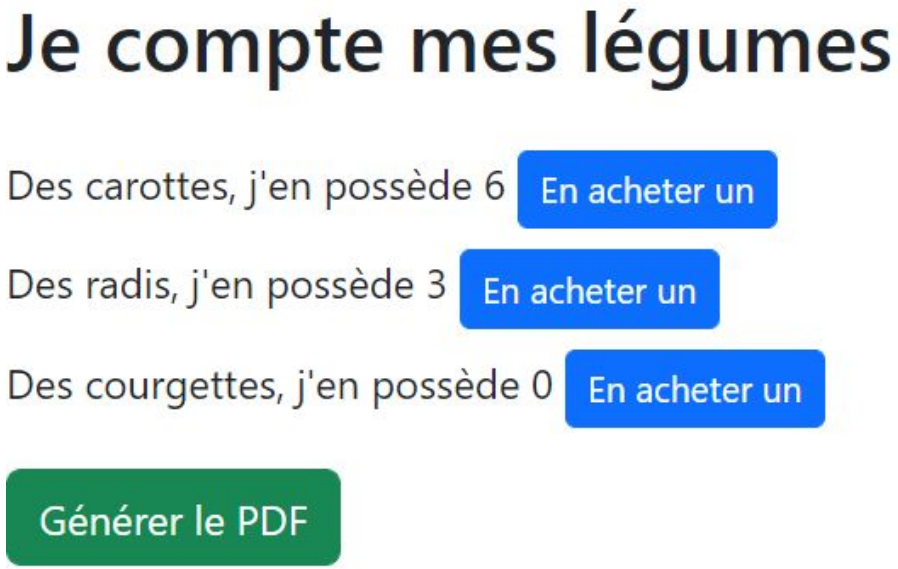
Exercice

```
<div id="app">
        <compteur @calculer="calculerSomme"></compteur>
        <compteur @calculer="calculerSomme"></compteur>
        <compteur @calculer="calculerSomme"></compteur>
        <span style="margin-left:10px;">{{ somme }}</span>
    </div>

    <script>
        app = Vue.createApp({
            data(){
                return{
                    somme: 0
                }
            },
            components:{
```

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
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 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 #input>
                    <input type="text" v-model="rechercheUserText" />
                </template>
</liste-users>
```

Et dans liste_users.js, sous le h2 :

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

TP 2