

Formation Vue.js 3

Module de 3 jours (21 heures)

Apprendre à créer une application Web dynamique
et performante avec le framework Vue.js 3

Formateur : Jérémy Dumaye

Nous restons ouverts, nous proposons nos **formations à distance**

PRENEZ LE TEMPS D'APPRENDRE

Les formations recommandées par les développeurs pour les développeurs

NOS FORMATIONS



OBJECTIFS DE LA FORMATION

Organisation journée de formation :

- **9h** : début de la journée
- **11h – 11h15** : pause matin
- **12h30 – 14h** : pause déjeuner
- **16h – 16h15** : pause après-midi
- **17h30** : fin de la journée



Une application web simple avec Vue.js 3

Introduction de la formation, présentation, préparation setup, bases techniques et premiers exercices avec Vue.js



Une application faite de composants

Rôle d'un composant, et de ses attributs, mise en place des composants, Vue Build Tools, et exercices



Routeur, plugins et API

Mise en place du routeur pour créer les routes, extensions plugins, extension API, et exercices

Projet fil rouge des 3 jours :

Création d'un annuaire de films permettant de lister les films, et pouvoir créer, modifier, et supprimer un film. Gestion également des catégories de chaque film.

Extension Vue.js – les Mixins

Les mixins distribuent des fonctionnalités réutilisables pour les composants Vue. Un objet mixin peut contenir toutes les options de composant. Il existe deux types de mixins, les locaux, et les globaux.

Mixin global :

Le mixin global même s'il existe n'est pas des plus conseillés, car il affecte toutes les instances de Vue y compris celles des bibliothèques tierces.

```
app.mixin({
  data() {
    return {
      copyright: "",
    };
  },
  created() {
    const copyright = this.$options.copyright;

    if (copyright === true) {
      this.copyright = "© Jérémy Dumaye 2021";
    }
  },
});
app.mount("#app");
```

*Le mixin global s'ajoute directement dans le fichier **main.js**. Ici le but était de créer une option personnalisée du **ViewModel** que l'on a nommé **copyright**, et s'il est à **true** on ajoute la possibilité d'utiliser la variable **copyright** pour en afficher le message où l'on souhaite.*

```
<template>
  <p>{{ copyright }}</p>
</template>

<script>
export default {
  name: "App",
  copyright: true,
};
</script>
```

Extension Vue.js – les Mixins

Mixin local :

Le mixin local peut se définir directement dans un composant, mais vu qu'il a pour volonté de s'ajouter au cas par cas en fonction du composant et besoin, il sera plus judicieux de l'ajouter dans le dossier src, dans un dossier mixins avec le nom du mixin en question.

```
src > mixins > JS logoSite.js > ...
1  export default {
2    data() {
3      return {
4        logoSite: 'https://vue3-fr.netlify.app/logo.png'
5      },
6    }
7 }
```

```
<template>
|  <p>Mon logo :</p>
|  
</template>

<script>
import logoSite from '@/mixins/logoSite.js';

export default {
  name: "Compteur",
  mixins: [logoSite],
  data() {
    return {
      nbClics: 0,
    }
  },
</script>
```

Le mixin **logoSite.js** a été créé dans le dossier **mixins** et permettra de mettre à disposition l'url du logo via la variable **logoSite** lorsque ce sera nécessaire.

Pour avoir cette url, il suffira d'importer le mixin **logoSite** dans le composant en question, et d'enregistrer le mixin dans le ViewModel via **mixins: [logoSite]**

Exercice numéro 1

01

Créer un mixin dans le dossier mixins qui permettra de réaliser la méthode d'ajout d'incrémentation de + x au clic sur un bouton. Il sera à ajouter dans deux composants d'exemple pour prouver son côté fonctionnel. Le premier composant sera pour ajouter de x en x produit, et le deuxième pour augmenter le compteur de + x en x (exemple de 1 en 1, 2 en 2, ou 4 en 4, etc.)

Astuce : Il faudra penser à une data à incrémenter provenant du Model du mixin

Création de plugins dans Vue.js

Vue peut utiliser des plugins pour étendre ses possibilités. Il en existe déjà des existants très intéressants, mais il est tout à fait possible de créer ses propres plugins pour des besoins précis. Les plugins ajoutent généralement des fonctionnalités de niveau global à Vue.

Pour trouver des plugins très intéressants, il y a <https://github.com/vuejs/awesome-vue#components--libraries> qui permet de trouver tout type de plugins déjà fournis par la communauté pour éviter d'en créer pour rien s'ils existent déjà.

Création d'un plugin permettant d'avoir un mail grâce à un paramètre :

```
export default {
  install: (app) => {
    app.config.globalProperties.$mail = (key) => {
      if (key === 'admin') {
        return 'adminkey@site.fr'
      } else if (key === 'moderator') {
        return 'moderatorkey@site.fr'
      } else {
        return 'invalid key error.'
      }
    }
  }
}
```

```
1 import { createApp } from "vue";
2 import App from "./App.vue";
3 import mailAdmin from "./plugins/mailAdmin";
4
5 const app = createApp(App);
6
7 app.use(mailAdmin);
8
9 app.mount("#app");
```

```
<template>
  <a :href="`mailto:${$mail('admin')}`">Contacter l'admin</a>
  <a :href="`mailto:${$mail('moderator')}`">Contacter le modérateur</a>
</template>

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

On crée un plugin permettant d'avoir une fonction globale renvoyant l'adresse mail suivant un paramètre. On l'ajoute dans le main.js en **app.use(name)** pour pouvoir l'utiliser ensuite dans nos composants.

Exercice numéro 2

02

Créer un plugin permettant de définir une variable globale qui permettra d'ajouter automatiquement une balise bold à une variable via l'utilisation de app.config.globalProperties

Gérer des routes grâce à Vue Router

Construire une SPA complète grâce à Vue router, l'ensemble du site web sera sur une seule page avec une rapidité d'exécution importante et un temps de réponse optimal. Le router permet d'afficher des **views** précises, qui comprendront également des composants, en fonction d'une url fictive paramétrée qui permettra de faire croire qu'on change de page et qu'on navigue de page en page.

Installation de Vue Router :

```
› npm install vue-router@next
```

Une fois installées, nos routes seront à paramétrer dans un fichier **index.js** situé dans un dossier **router** qu'on créera dans le dossier **src**.

Il va également falloir créer nos pages (**views**) qu'on ajoutera dans un dossier **views** situé dans le dossier **src**.

Une fois tout créé et configuré, il faudra importer le router dans le **main.js** et faire un **app.use(router)**

Préparation du fichier router

```
src > router > JS index.js > ...
1 import { createRouter, createWebHistory } from "vue-router";
2 import Home from '@/views/Home.vue';
3 import Entreprise from '@/views/Entreprise.vue';
4
5 const routes = [
6   {
7     name: 'Home',
8     path: '/',
9     component: Home,
10    },
11   {
12     name: 'Entreprise',
13     path: '/entreprise',
14     component: Entreprise,
15   }
16 ];
17
18 const router = createRouter({
19   history: createWebHistory(),
20   routes,
21 })
22
23 export default router;
```

On importe les méthodes nécessaires pour créer la route.

On importe nos views permettant d'être les pages parentes de notre SPA

On crée un tableau routes qui contiendra les paramètres nécessaires demandés par vue-router avec notamment la déclaration du nom de la route, sont path, et quel composant est appelé.

On crée l'appel pour créer le vue-router en passant notre paramètre des routes.

On exporte le router pour pouvoir ensuite l'utiliser dans notre application.

```
import { createRouter, createWebHashHistory } from "vue-router";
```

Le mode `createWebHashHistory` peut également être utilisé pour le paramétrage `history` du router, mais il n'est pas très judicieux car il laisse un # dans l'url. localhost:8080/#/

Modification du fichier main.js

```
> JS main.js > ...
import { createApp } from "vue";
import App from "./App.vue";
import router from './router';

const app = createApp(App);

app.use(router);

app.mount("#app");
```

On importe le router provenant de notre dossier router et on utilise `app.use(router)` pour utiliser le router dans toute notre application.

Paramétrage de notre layout App.vue

```
▼ App.vue > Vetur > {} "App.vue"
<template>
  <nav>
    <router-link to="/">Accueil</router-link>
    <router-link to="/entreprise">Mon entreprise</router-link>
  </nav>
  <router-view/>
</template>

<script>
export default {
  name: "App"
};
</script>

<style>
</style>
```

Le layout global App.vue accueillera le `<router-view/>` qui sera le composant global qui permettra d'afficher les données de toutes les views en fonction du choix d'url. Ici pour voir comment créer un lien vers une page, nous avons mis le menu directement en évidence, mais nous aurons bien entendu pu le mettre dans un composant dédié [MenuApp](#).

Passer des paramètres dans l'url

```
import ViewProduit from '@/views/ViewProduit.vue';
```

```
{  
  name: 'Produit',  
  path: '/produit/:name',  
  component: ViewProduit,  
  props: true,  
  meta: {  
    title: 'Page produit'  
  },  
},
```

```
views > ViewProduit.vue > ...  
<template>  
| <h1>Page du produit : {{ name }}</h1>  
</template>  
  
<script>  
export default {  
  name: "ViewProduit",  
  props: ["name"],  
};  
</script>
```

Pour préparer l'acceptation de paramètres dans une url, j'ajoute dans le path un `:name` pour attendre une donnée qui sera retrouvée en props via la propriété `name`. Je n'oublie pas d'ajouter `props: true` pour dire que l'on passe des props.

Je crée la view `ViewProduit.vue` dans le dossier `views` qui sera la page de mes produits. Grâce à l'url et au paramètre `name`, je peux le récupérer en `props` pour l'utiliser en tant que variable.

```
<router-link :to="{name: 'Produit', params: {name: 'Formation web'}}">Produit formation</router-link>
```

Dans mon menu, j'ajoute le `router-link` avec un paramétrage dynamique pour donner une valeur à `name`.
© Jérémie Dumaye – <https://jeremydumaye.com/>

Envoyer un lien lors d'un événement clic

```
<button @click="showProduct('Référencement SEO')>Produit référencement</button>

<script>
export default {
  name: "App",
  methods: {
    showProduct(nameProduct) {
      this.$router.push({name: 'Produit', params: {name: nameProduct}});
    }
  }
}
</script>
```

Il arrive parfois de ne pas utiliser forcément la balise `<router-link>` pour définir un lien vers une page `vue-router`. Nous pouvons donc nous servir de la variable globale `this.$router` pour pousser manuellement vers la bonne route. Ici, au clic sur le bouton du Produit référencement, on déclenche la méthode `showProduct()` avec le paramètre du `name` pour le passer dans le `this.$router.push()`

Autres caractéristiques gestion vue-router

Gestion CSS des menus :

```
▼<nav>
  <a class="" href="/">Accueil</a> [event]
  <a class="router-link-active router-link-exact-active"
  href="/entreprise" aria-current="page">Mon entreprise
  </a> [event]
  <a class="" href="/produit/Formations%20web">
  Produit formation</a> [event]
  <button>Produit référencement</button> [event]
</nav>
```

Ajouter le title de la page :

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

router.afterEach((to) => {
  document.title = to.meta.title ? to.meta.title : "Page de mon application"
}

export default router
```

```
const routes = [
  {
    name: 'Home',
    path: '/',
    component: Home,
    meta: {
      title: 'Accueil'
    }
  },
  {
```

Via la méthode **afterEach()** il est possible de définir le **document.title** de toutes les routes en liant au **to.meta.title** de la route. Il suffira ensuite de créer la propriété **meta** et le **title** pour avoir le titre. Pour ajouter une titre dynamique provenant du paramètre (par exemple le **:name**) de la route il faudra ajouter :

```
router.afterEach((to) => {
  document.title = to.meta.title ? to.meta.title : "Page de mon application";
  document.title += to.params.name ? ` - ${to.params.name}` : "";
})
```

Création d'une route pour erreur 404

Pour créer une route pour les pages 404, il suffira de créer une nouvelle route avec le nom ‘NotFound’ et un path particulier :

```
,  
{  
  name: 'NotFound',  
  path: '/:pathMatch(.*)',  
  component: NotFound,  
  meta: {  
    title: 'Page non trouvée !'  
  }  
};
```

```
> views > ▼ NotFound.vue > Vetur > {} "NotFound.vue" > ⚙ style  
1   <template>  
2     <h1>Oups, page non trouvée !!</h1>  
3   </template>  
4  
5   <script>  
6   export default {  
7     name: "NotFound",  
8   }:
```

Utilisation de Bootstrap pour le design

Pour la partie design, il peut être intéressant d'installer la bibliothèque reconnue qu'est Bootstrap. Voici la marche à suivre :

- Installation de Bootstrap et PopperJs :

```
› npm install bootstrap @popperjs/core
```

- Puis ajouter ces instructions dans le fichier main.js :

```
import "bootstrap/dist/css/bootstrap.min.css"
import "bootstrap/dist/js/bootstrap.js"
import { createApp } from 'vue'
import App from './App.vue'
```

Pouvoir utiliser le préprocesseur SCSS

Il est souvent plus intéressant d'utiliser le préprocesseur SCSS dans nos projets plutôt que le CSS basique. Si vous n'avez pas choisi par défaut le préprocesseur SCSS lors de l'installation du projet, il faudra installer ce package :

```
› npm install sass
```

Il sera ensuite possible d'intégrer du SCSS directement dans le style des composants : `<style lang="scss">`

Il est également possible d'avoir un fichier de config pour pouvoir gérer notamment le SCSS global à vos composants (comme par exemple les variables) Il faut donc ajouter cela dans le fichier `vite.config.js` à la racine de votre projet :

```
JS vite.config.js > ...
1  import { fileURLToPath, URL } from 'url'
2
3  import { defineConfig } from 'vite'
4  import vue from '@vitejs/plugin-vue'
5
6  // https://vitejs.dev/config/
7  export default defineConfig({
8    plugins: [vue()],
9    resolve: {
10      alias: {
11        '@': fileURLToPath(new URL('./src', import.meta.url))
12      }
13    },
14    css: {
15      preprocessorOptions: {
16        scss: {
17          additionalData: `
18            @import "@/assets/scss/_variables.scss";
19          `
20        }
21      }
22    }
23  })
```

```
› scss > _variables.scss > ...
$blue: #rgba(52, 11, 236, 0.8);
```

```
<style lang="scss">
div {
  h1 {
    color: $blue;
  }
}
</style>
```

Exercice projet fil rouge



Projet fil rouge : Mises en pratique avec un annuaire de films :

Créer le projet fil rouge via Vue Build Tools et le Vue Router :

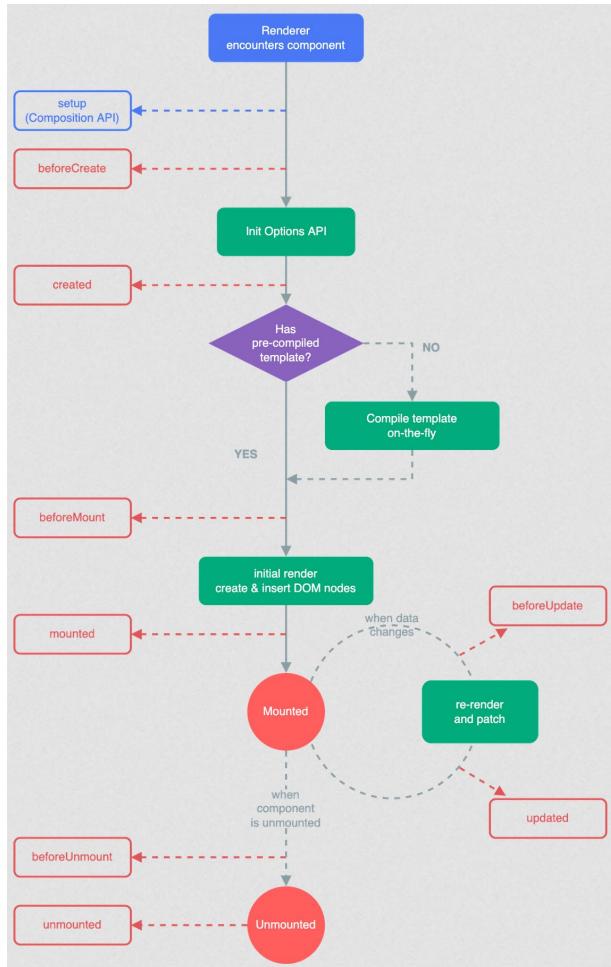
- Pouvoir lister les éléments d'un catalogue de films disponibles indiquant le nom du film, son année de sortie, son réalisateur, et sa catégorie. Ce listing sera sur la page d'accueil de l'application.
- Pouvoir supprimer ou éditer les éléments de la liste de la page d'accueil via deux boutons.
- Au clic sur le bouton supprimer, prévoir un prompt demandant si c'est sûr de vouloir supprimer
- Au clic sur le bouton éditer l'élément, envoyer vers une autre page avec la modification de l'élément à effectuer. Une fois sur le clic validé, renvoyer vers la page d'accueil
- Pouvoir ajouter un film via la page 'Ajouter un film' accessible depuis le menu général de l'application. Une fois validé, renvoyer vers la page d'accueil
- Pouvoir ajouter une catégorie à la liste déjà préparée via la page 'Ajouter une catégorie' accessible depuis le menu général et renvoyer vers la page d'accueil une fois ajouté
- Créer le composant du menu pour afficher les liens vers toutes ces views

En fonction de l'avancement, prévoir une partie design via Bootstrap pour le menu, le tableau du listing, les formulaires, etc. + prévoir de lister les catégories pour en supprimer ou modifier si besoin

BONUS

Point sur le cycle de vie dans Vue 3

Composition API avec le lifecycle :

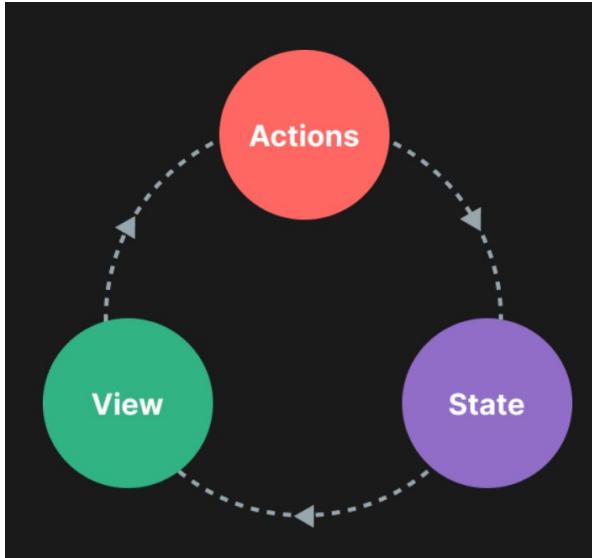


Voici le cycle de vie disponible dans Vue 3. Pour les appels notamment de la liaison avec une API via Axios, nous avons utilisé le hook `created()` jusqu'à présent. C'est ce que nous ferons avec la liaison avec le state global également quand il faudra appeler des datas qui ont besoin d'une API.

En autre hook, nous avons notamment `mounted()`, `updated()`, etc.

Plus d'infos : <https://vuejs.org/api/options-lifecycle.html>

Utilisation du state global avec Pinia



Pinia permet de stocker et centraliser toutes les données pour tous les composants de notre application Vue.js. Bien pratique pour avoir une gestion précise de l'ensemble et ainsi pouvoir récupérer, modifier, supprimer, créer des données précises. Par des actions depuis notre composant, nous pourrons récupérer ou modifier les données présentes et stocker grâce à **Pinia**.

Installation de Pinia : ➤ `npm install pinia`

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

import './assets/main.css'

const pinia = createPinia()
const app = createApp(App)

app.use(pinia)

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

Il suffit ensuite d'ajouter l'instance globale de Pinia dans le main.js

Nous allons ensuite pouvoir commencer le paramétrage de Pinia pour l'utiliser dans nos différents composants en fonction du besoin

Utilisation du state global avec Pinia

Commençons le paramétrage de Pinia :

```
stores > JS usersStore.js > ...
import { defineStore } from 'pinia'

export const useUsersStore = defineStore('users', {
  state: () => {
    return {
      users: [
        {
          prenom: 'John',
          nom: 'Duje',
          job: 'Développeur front'
        },
        {
          prenom: 'Laurie',
          nom: 'Tolas',
          job: 'CTO'
        }
      ]
    }
  },
  actions: {
    addUser(user) {
      this.users.push(user)
    },
    getters: {
      countAllUsers: (state) => state.users.length
    }
  }
})
```

On importe la méthode **defineStore** de Pinia

Les bonnes pratiques veulent que la variable const créée se compose du préfix **use** et que le nom du fichier js se termine par **Store**

Le store de Pinia a 3 paramètres :

- **Le state** : permet de retourner les datas du store défini. Les datas pourront évoluer en fonction des actions du store

- **Les actions** : comme son nom l'indique, cela représente les actions du store permettant de lancer un appel API pour récupérer des datas et les ajouter dans le state, ou encore définir des actions pour ajouter un nouvel élément, etc.

- **Les getters** : jouer le rôle d'une propriété calculée comme ce que nous connaissons via *computed* dans les composants. Permet de retourner un calcul provenant du state comme le nombre total d'utilisateurs par exemple

State global dans les composants

Nous allons à présent voir comment utiliser le state global via Pinia dans un composant

```
▼ App.vue > ...
<template>
  <div>
    {{ usersStore.users }}
  </div>
  <p>Il y a en tout {{ usersStore.countAllUsers }} utilisateurs</p>
</template>

<script>
import { mapStores } from 'pinia'
import { useUsersStore } from "@stores/usersStore"

export default {
  name: "App",
  computed: {
    ... mapStores(useUsersStore)
  }
}</script>
```

On importe l'utilisation de la méthode mapStores de pinia + le store qu'on a besoin d'utiliser

On se sert de la méthode mapStores de pinia pour garder en mémoire le store importé. A noter qu'il faut utiliser le mapStores dans la propriété calculée « computed » du composant.

Nous pourrons ensuite appeler toutes les méthodes du store, c'est-à-dire le state, les actions, ou les getters.

Exercice Bonus



Utiliser Pinia pour réaliser l'application suivante :

- Avoir un store permettant de récupérer les 10 inscrits via l'API randomuser.me
(<https://randomuser.me/api/?results=10>)
- Pouvoir ajouter, modifier, supprimer un inscrit depuis les différents composants et en passant par des actions du store
- Les 10 inscrits du départ seront affichés dans un tableau listing