Table des matières

1 Construire une application Vue 3

Dans ce cours, nous allons construire une application de niveau production en utilisant Vue 3. Nous commencerons par créer le projet en utilisant ***create-vue***. Ensuite, nous découvrirons les composants *.vue* à fichier unique et la manière dont ils peuvent être utilisés pour créer une application à page unique (Single Page Application). Nous couvrirons les principes fondamentaux de ***Vue Router*** afin de pouvoir naviguer entre les différentes vues de notre application, et nous irons même chercher des données externes réelles en utilisant des appels API avec Axios. Nous terminerons par l'apprentissage du processus de construction et de la manière de déployer notre application en production.

Si vous êtes prêt à construire une application Vue 3 dans le monde réel, je vous donne rendez-vous dans la suite du cours.

2 Create-Vue - créer le projet

Dans ce tutoriel, nous allons créer notre projet en utilisant *create-vue*, qui est un outil CLI basé sur un outil de construction appelé *Vite.js*. Nous allons visiter le projet que l'outil CLI génère afin de nous familiariser avec le travail dans ces fichiers et dossiers.

2.1 Pourquoi une CLI ?

Comme vous le savez probablement, CLI signifie Command Line Interface (interface de ligne de commande) et fournit un système complet pour le développement rapide de *Vue.js*. Cela signifie qu'il fait beaucoup de travail fastidieux pour nous et nous fournit des fonctionnalités précieuses prêtes à l'emploi.

Il nous permet de sélectionner les bibliothèques que notre projet va utiliser

Il les intègre ensuite automatiquement dans le projet.

Il est basé sur Vite.js

Tous nos fichiers JavaScript, notre CSS et nos dépendances sont livrés dans un build rapide comme l'éclair pour le développement.

Il nous permet d'écrire notre HTML, CSS et JavaScript comme nous le souhaitons

Nous pouvons utiliser des composants *.vue* à fichier unique, TypeScript, SCSS, Pug, les dernières versions d'ECMAScript, etc.

> *Qu'est-ce que Pug ?*
>
> *Pug est un moteur de template de haute performance implémenté avec JavaScript pour Node.js et les navigateurs. Il permet d'écrire le code HTML de manière simplifiée et plus lisible grâce à une syntaxe spécifique.*

Il permet le remplacement des modules à chaud (HMR)

Lorsque vous enregistrez votre projet, les changements apparaissent instantanément dans le navigateur.

2.2 Création d'un projet Vue

Pour créer une nouvelle application à l'aide de *create-vue* :

```
npx create-vue real-world-vue
```

Cette commande lancera la création d'un projet Vue, avec le nom "real-world-vue".

Cela nous guidera à travers un processus où nous sélectionnerons les options pour configurer notre projet :

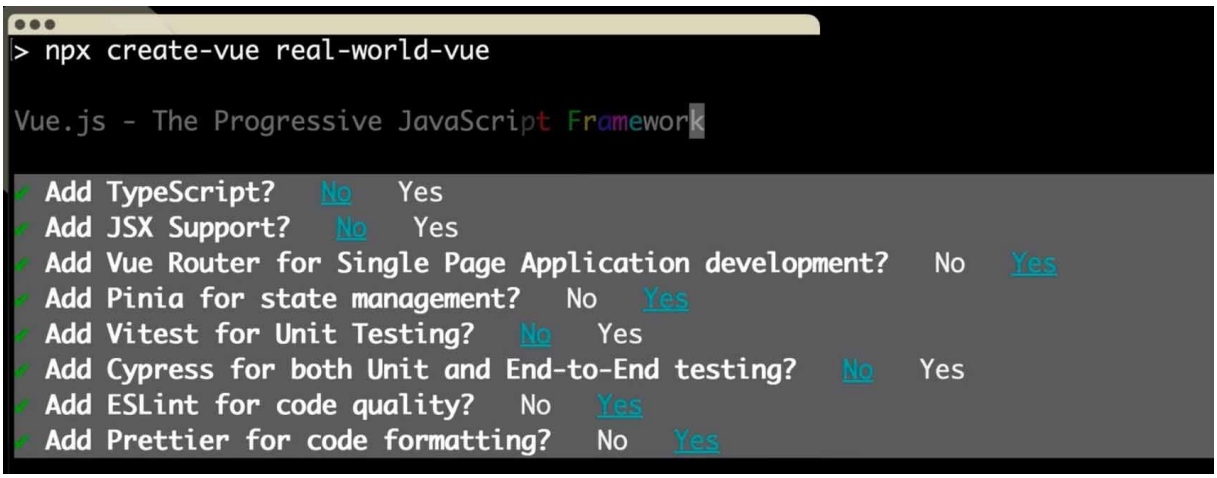
Nous pouvons choisir *Oui/Non* pour chaque option.

Nous allons choisir *Non* pour **TypeScript** et **JSX Support**.

Pour **Vue Router** et **Pinia**, nous choisirons Oui.

Ensuite, nous choisirons *Non* pour **Unit Testing** et **Cypress**.

Enfin, nous choisirons Oui pour **ESLint** et **Prettier** pour le contrôle du code (erreurs) et le formatage respectivement.



Après avoir passé en revue les options, notre projet sera créé automatiquement.

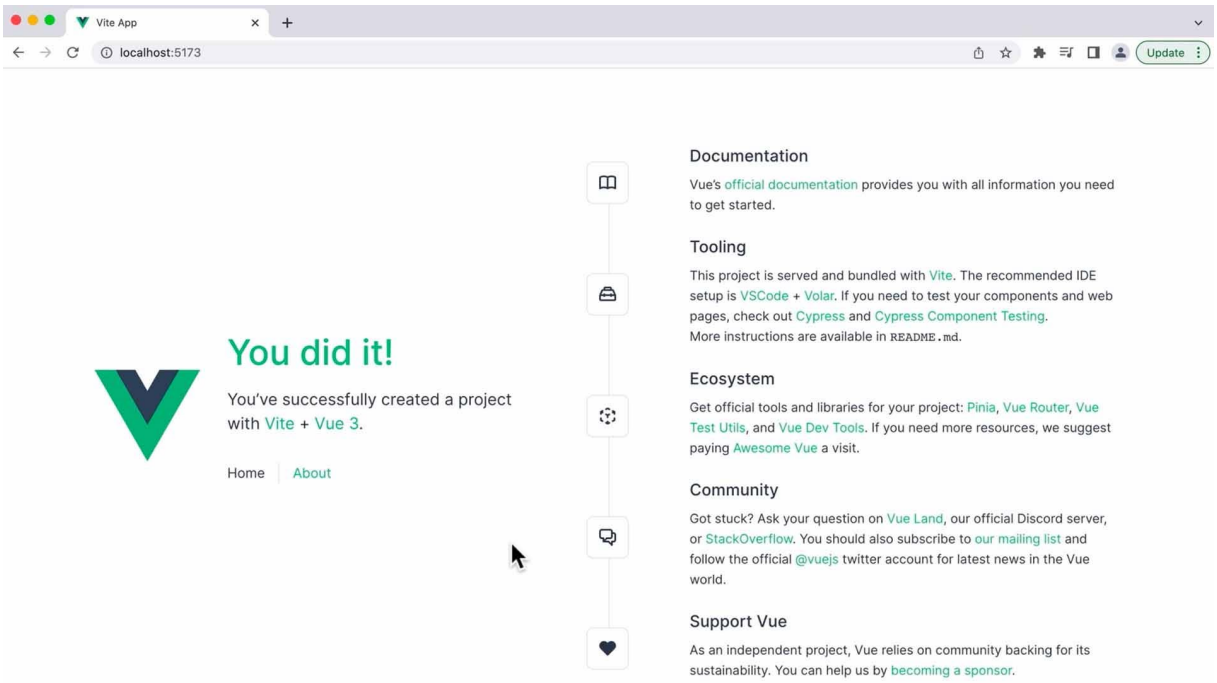
2.3 Servir notre projet

Une fois la création de notre projet terminée, nous pouvons y accéder (`cd real-world-vue/`)

Installer toutes les dépendances pour notre nouveau projet :

```
npm install
```

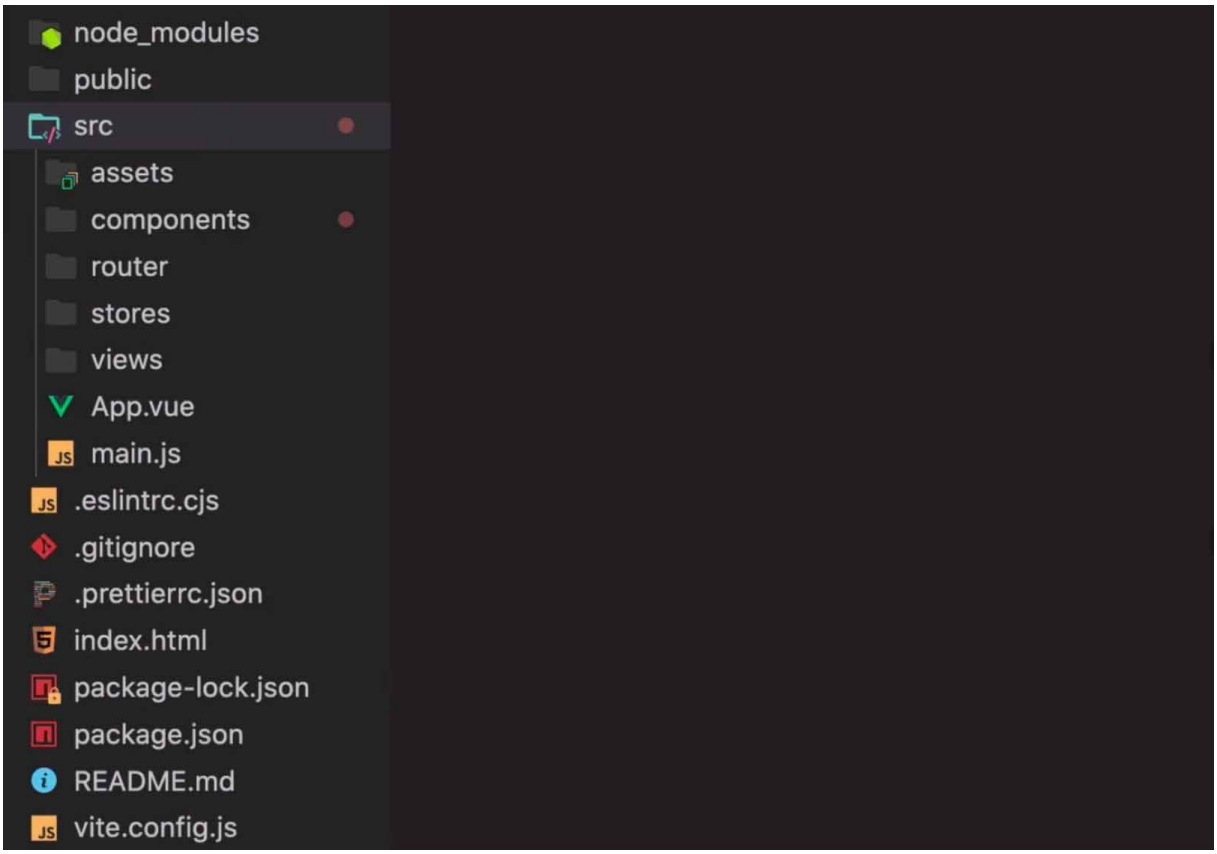
Afin de le visualiser en direct dans notre navigateur, nous allons exécuter la commande `npm run dev` qui compile l'application et la diffuse en direct sur un hôte local (http://localhost:5173).



Ci-dessus se trouve notre application, fonctionnant en direct dans le navigateur. Elle comporte déjà deux pages, la page **d'accueil** et la page " **À propos** ", entre lesquelles nous pouvons naviguer car elle utilise Vue Router.

2.4 Visite de notre projet Vue

Maintenant que nous savons comment créer notre projet à partir du terminal et de l'interface utilisateur, jetons un coup d'œil au projet qui a été créé pour nous.



Le répertoire **node_modules** est celui où sont stockées toutes les bibliothèques dont nous avons besoin pour construire Vue.

Le répertoire **src** est celui où vous passerez le plus de temps car il contient tout le code de l'application.

Vous voudrez placer la majorité de vos ressources, telles que les images et les polices, dans le répertoire **assets** afin qu'elles puissent être optimisées par Vite.

Le répertoire **components** est l'endroit où nous stockons les composants, ou blocs de construction, de notre application Vue.

Le dossier **router** est utilisé pour Vue Router, qui permet la navigation sur notre site. Nous utilisons Vue Router pour afficher les différentes "vues" de notre application à page unique.

Le dossier **stores** est l'endroit où nous plaçons le code Pinia, qui gère l'état de l'application. À la fin de ce cours, vous aurez une compréhension de base de ce à quoi sert Pinia, mais nous n'implémenterons aucun code Pinia. Ce cours est un cours de base qui vous prépare à nos autres cours sur Pinia.

Le répertoire **views** est l'endroit où nous stockons les fichiers de composants pour les différentes vues de notre application, que Vue Router charge.

Le fichier **App.vue** est le composant racine dans lequel tous les autres composants sont imbriqués.

Le fichier **main.js** est ce qui *rend* notre composant **App.vue** (et tout ce qu'il contient) et le *monte* dans le DOM.

eslintrc.cjs et **.prettier.json** sont des fichiers de configuration pour ESLint et Prettier.

Le fichier **index.html** est l'endroit où le composant **App.vue** sera monté.

Ensuite, nous avons un fichier **.gitignore** où nous pouvons spécifier ce que nous voulons que git ignore, ainsi que notre **package.json**, qui aide npm à identifier le projet et à gérer ses dépendances, et un **README.md**.

Enfin, il y a un **vite.config.js** puisque c'est une application qui tourne sur Vite.js

2.5 Comment l'application est-elle chargée ?

Vous vous demandez peut-être maintenant comment l'application est chargée. Jetons un coup d'œil à ce processus.

src/main.js

```
import { createApp } from 'vue'
import { createPinia } from 'pinia'

import App from './App.vue'
import router from './router'

import './assets/main.css'

const app = createApp(App)

app.use(createPinia())
app.use(router)

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

Dans notre fichier **main.js**, nous importons la méthode `createApp` de Vue, ainsi que notre composant **App.js**. Nous exécutons ensuite cette méthode, en alimentant l' `App` (le composant racine qui inclut tout le code de notre application, puisque tous les autres composants sont imbriqués en son sein).

Comme le nom de la méthode l'indique explicitement, cela crée l'application et nous lui disons d'`utiliser` le `routeur` et une instance Pinia nouvellement créée. Enfin, l'application est montée dans le DOM via la méthode `mount`, qui prend un argument pour spécifier *où* dans le DOM l'application doit être montée. Mais où se trouve exactement cet identifiant `"#app"` ?

Si nous jetons un coup d'œil à l'intérieur de notre fichier **index.html**, nous pouvons voir qu'il y a une div avec l'id `"app"` :

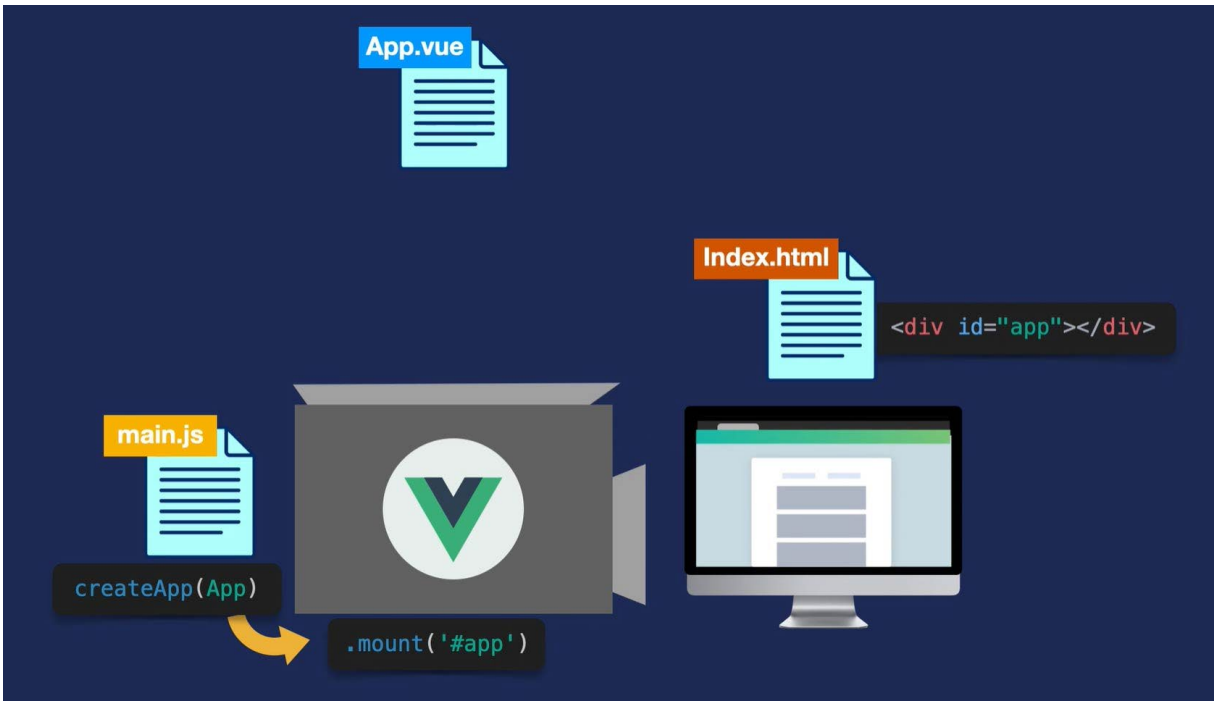
index.html

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

C'est là que notre application Vue est montée. Plus tard, nous comprendrons mieux comment cet **index.html** sert de "page unique" à notre application monopage.

2.6 Assembler tout cela

Examinons ce processus d'un point de vue plus visuel :



2.7 Récapitulation

Vous devriez maintenant avoir compris comment créer un projet Vue et comment le gérer à partir de l'interface utilisateur Vue. Nous avons également exploré le projet qui a été créé pour nous afin d'être prêts à commencer à le personnaliser. Dans la prochaine leçon, nous construirons notre premier composant .vue à fichier unique.

3 Composants à fichier unique

Maintenant que nous avons créé notre projet avec **create-vue**, nous sommes prêts à le personnaliser pour construire notre propre application.

Si vous êtes en train de coder (ce que je vous encourage à faire), vous voudrez bien consulter la branche L3-start de notre projet pour récupérer le code de départ (L3 est l'abréviation de Lesson 3). Dans ce code, je veux attirer votre attention sur ce fichier que j'ai ajouté :

.prettierrc.json

```
{
  "singleQuote": true,
  "semi": false
}
```

Ici, j'ai établi quelques règles pour que Prettier remplace les guillemets doubles (") par des guillemets simples (') et supprime les points-virgules (;). Je ne plaide pas pour ou contre les points-virgules et les guillemets. Il s'agit d'un exemple simple de la manière dont vous pouvez ajouter des règles de configuration Prettier à votre projet. Nous pourrions également faire quelque chose de similaire pour ESLint. Pour un aperçu plus approfondi de la façon dont vous pouvez configurer ESLint + Prettier et tirer le meilleur parti de VS Code, vous pouvez consulter cet article.

Ressources pour cette leçon

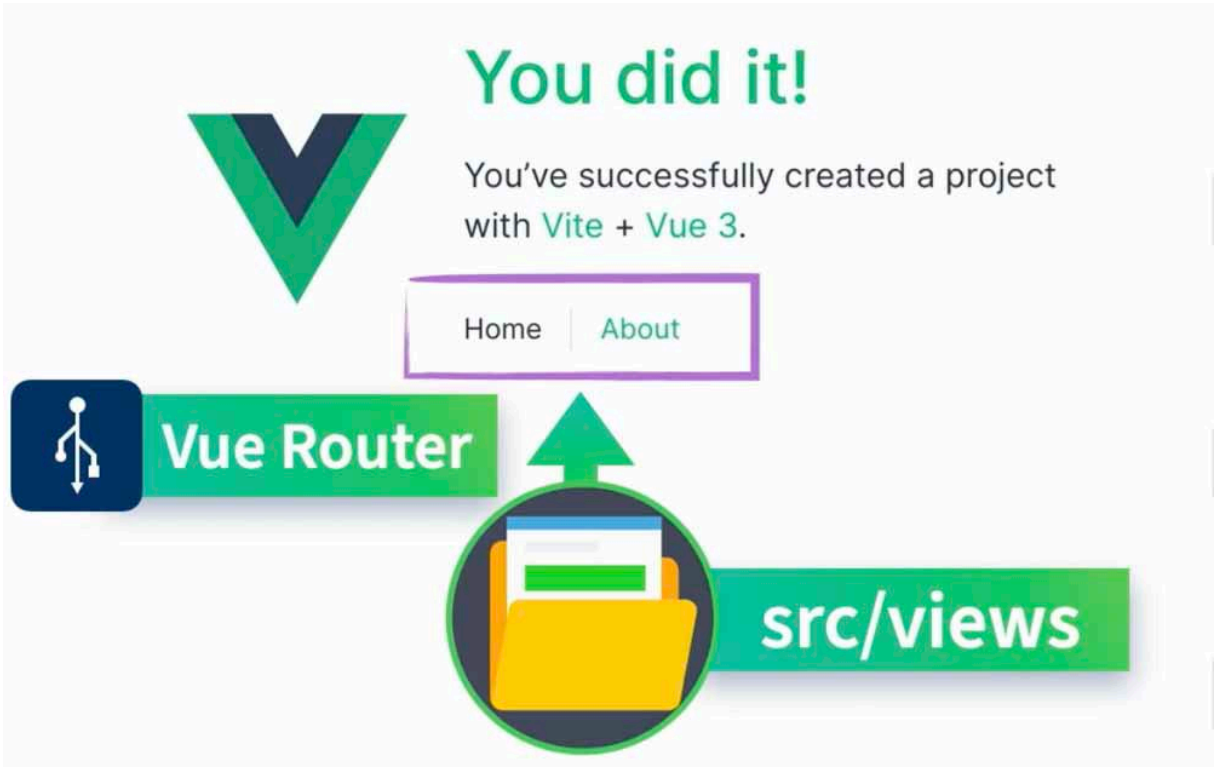
Code source :

- Code de départ
- Code final
- Configuration de VS Code, ESLint + Prettier

3.1 Que sont ces fichiers .vue ?

Afin de commencer à construire notre application, nous devons comprendre comment les choses fonctionnent dans l'application de démonstration que le CLI a créée pour nous, y compris le répertoire **views**, qui comprend deux fichiers *.vue* uniques : **HomeView.vue** et **AboutView.vue**

Il s'agit des composants que Vue Router charge lorsque nous naviguons vers les routes Home et About, respectivement.



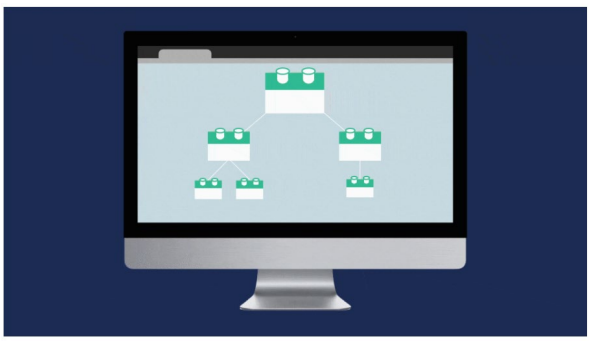
Dans la prochaine leçon, nous explorerons les éléments essentiels de Vue Router, mais pour l'instant, vous devez simplement comprendre que ces composants "view" sont les différentes vues qui peuvent être vues (ou vers lesquelles on peut naviguer) dans notre application. Ils peuvent contenir des composants enfants qui sont imbriqués en eux, et leurs enfants seront également affichés dans cette vue. Par exemple, le composant **HomeView.vue** a un enfant : **TheWelcome.vue**, qui contient un tas de code de ~~modèle~~ template qui s'affiche lorsque nous sommes sur la route Home.

Chacun de ces fichiers *.vue* est un composant à fichier unique, et c'est ce que cette leçon explore : Comment sont composés les composants à fichier unique et comment les utiliser pour créer une application Vue ?

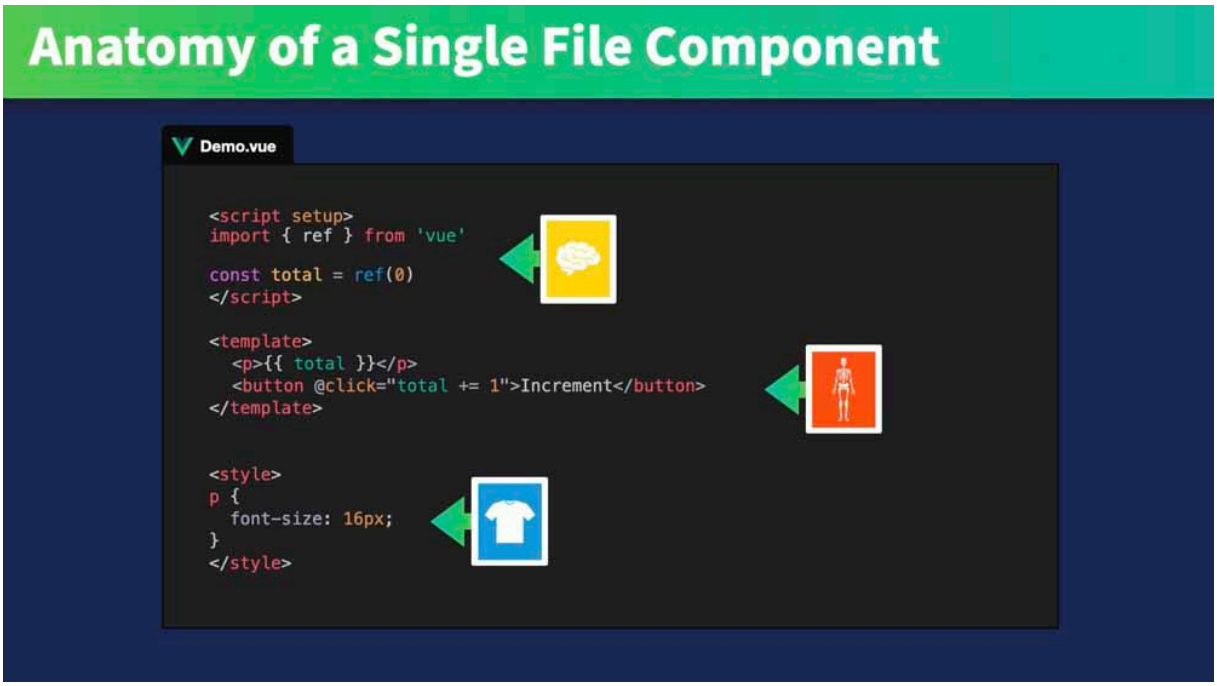
3.2 Anatomie d'un composant à fichier unique

Lorsque nous parlons d'applications Vue, nous parlons en fait d'une collection de composants Vue.

L'interface de l'image de gauche n'est en fait que le résultat d'un arbre de composants.



À quoi ressemblent donc ces composants à fichier unique sous le capot ?



Un composant *.vue* typique comporte trois sections : `<script>`, `<template>` et `<style>`.

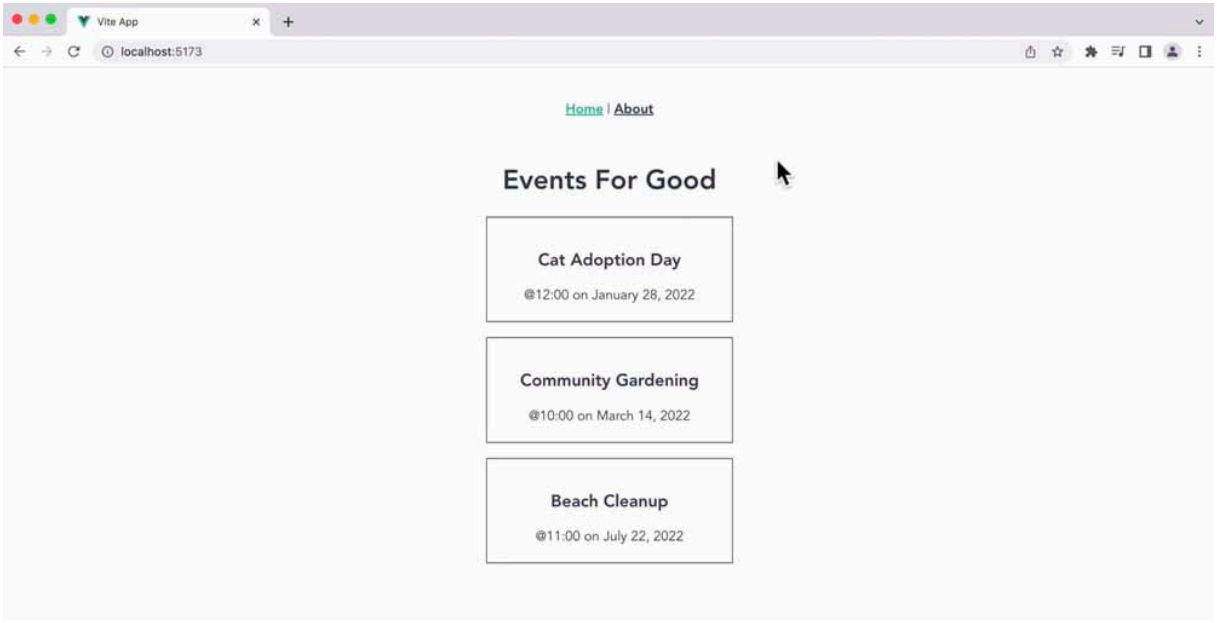
Pour reprendre l'analogie du corps humain, vous pouvez considérer le ~~modèle~~ template comme le squelette de votre composant puisqu'il lui donne une structure, et la section script est le cerveau, qui fournit l'intelligence et le comportement. La section style est exactement ce qu'elle semble être : les vêtements, le maquillage, la coiffure, etc.

Traditionnellement, ces sections sont écrites en HTML, JavaScript et CSS. Cependant, avec une configuration adéquate, vous pouvez également utiliser des alternatives telles que Pug, TypeScript et SCSS.

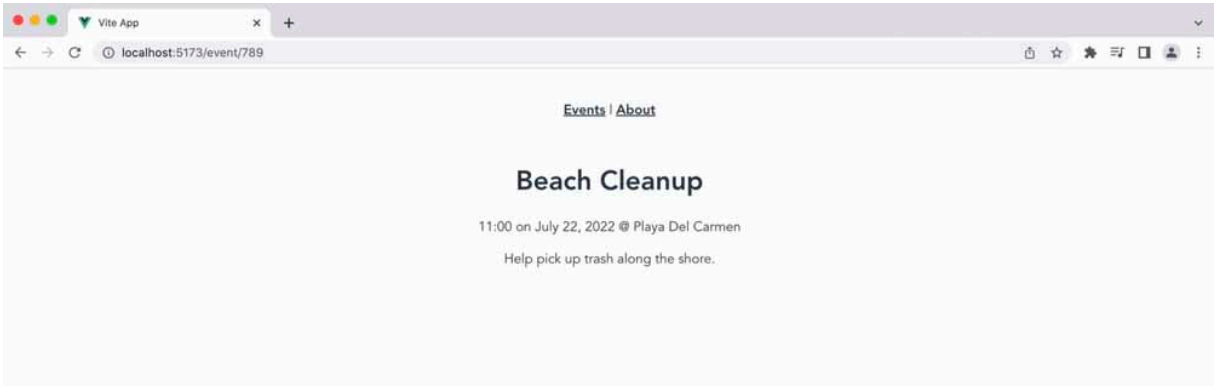
Maintenant que nous commençons à comprendre les composants à fichier unique, nous pouvons commencer à construire les nôtres. Mais tout d'abord, *que* construisons-nous exactement dans ce cours ?

3.3 L'application que nous construisons

À la fin de ce cours, nous aurons construit une application qui affiche des événements.



Les événements seront extraits d'un appel API externe et affichés sur la page d'accueil. Nous pourrons cliquer sur l'événement pour en voir les détails.



3.4 Notre premier composant à fichier unique

Pour commencer à construire notre premier composant, nous allons simplement supprimer le code qui se trouve dans les sections `<script>`, `<template>` et `<style>` de **HelloWorld.vue**. Pendant que nous y sommes, renommons ce fichier **EventCard.vue**, puisqu'il s'agit de la carte qui affiche des informations pour chaque événement.

📁 **src/components/EventCard.vue**

```vue
<script setup>
// defineProps({
//   msg: {
//     type: String,
//     required: true,
//   },
// })
</script>

<template>
  <div class="greetings"></div>
</template>

<style scoped></style>
```

Et puisqu'il n'y a plus de composant **HelloWorld**, nous devons le supprimer de **App.vue** :

📁 **src/App.vue**

```vue
<script setup>
import { RouterLink, RouterView } from 'vue-router'
// import HelloWorld from './components/HelloWorld.vue'
</script>

<template>
  <header>
    <img
      alt="Vue logo"
      class="logo"
      src="@/assets/logo.svg"
      width="125"
      height="125"
    />

    <div class="wrapper">
      <!-- <HelloWorld msg="You did it!" /> -->

      <nav>
        <RouterLink to="/">Home</RouterLink>
        <RouterLink to="/about">About</RouterLink>
      </nav>
```

3.5 Mise en place de la mise en page

Pendant que nous modifions **App.vue**, changeons également le code ici pour une nouvelle présentation.

Tout d'abord, supprimez la balise `<img>`. Placez le tout dans une nouvelle `div` (avec `id="layout"`). Si vous le souhaitez, ajoutez un séparateur (`|`) entre le lien Accueil et le lien À propos.

Le `<template>` doit maintenant ressembler à ceci :

📁 **src/App.vue**

```vue
<template>
  <div id="layout">
    <header>
      <div class="wrapper">
        <nav>
          <RouterLink to="/">Home</RouterLink> |
          <RouterLink to="/about">About</RouterLink>
        </nav>
      </div>
    </header>
    <RouterView />
  </div>
</template>
```

Ensuite, descendez jusqu'à `<style>` et remplacez le CSS existant par le suivant :

📁 **src/App.vue**

```css
#layout {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
}
nav {
  padding: 30px;
}
nav a {
  font-weight: bold;
  color: #2c3e50;
}
nav a.router-link-exact-active {
  color: #42b983;
}
```

Avec ce nouveau code CSS, nous n'aurons plus besoin du fichier **main.css** importé par défaut dans **main.js**.

Supprimons donc l'importation de `main.css` dans **main.js** :

📁 **src/main.js**

```js
import { createApp } from 'vue'
import { createPinia } from 'pinia'

import App from './App.vue'
import router from './router'

// import './assets/main.css'
```

Pendant que nous y sommes, supprimons également certains fichiers de composants dont nous n'avons plus besoin.

- tous les fichiers dans **src/composants/icons**
- **TheWelcome.vue**
- **WelcomeItem.vue**

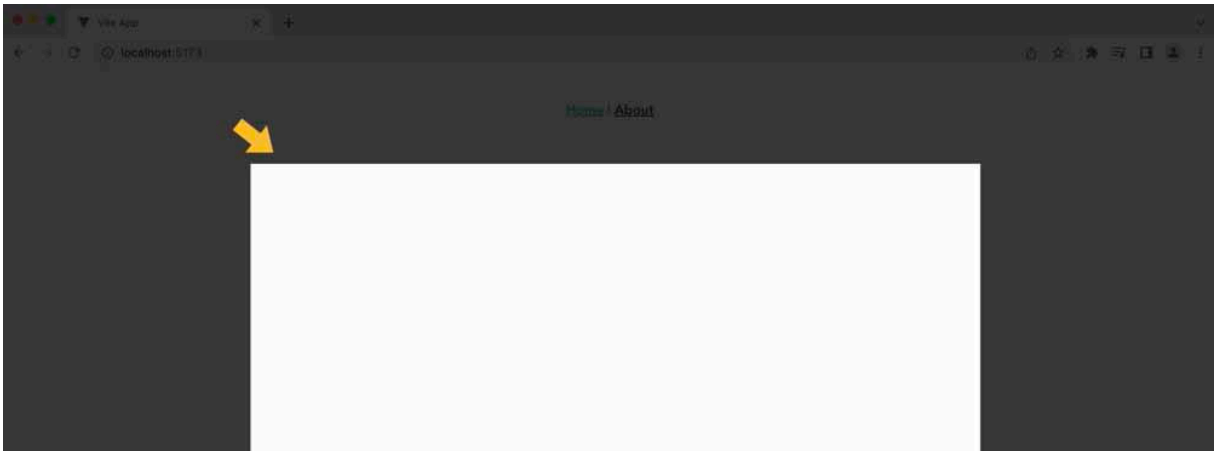
Enfin, nous devons aller dans **HomeView.vue** et supprimer l'importation de **TheWelcome.vue**. Et remplacer le code du modèle par une nouvelle div (avec class="home").

HomeView.vue devrait ressembler à ceci maintenant :

```
<script setup></script>

<template>
  <div class="home"></div>
</template>
```

Avec toutes les modifications ci-dessus, vous devriez maintenant être en mesure de voir la nouvelle mise en page dans le navigateur :

Le contenu de la page apparaîtra au centre :



Maintenant que la mise en page est prête, continuons à construire le composant **EventCard**.

15.04.24

3.6 Retour à EventCard

Tout d'abord, ajoutons quelques styles. Pour ce faire, nous allons modifier le nom de la classe de la `div`.

src/components/EventCard.vue

```
<script setup>
// defineProps({
//   msg: {
//     type: String,
//     required: true,
//   },
// })
</script>

<template>
  <div class="event-card"></div>
</template>

<style scoped></style>
```

Et ajouter quelques styles pour la carte d'événement (y compris un effet de survol) :

src/components/EventCard.vue

```
.event-card {
  padding: 20px;
  width: 250px;
  cursor: pointer;
  border: 1px solid #39495c;
  margin-bottom: 18px;
}
.event-card:hover {
  transform: scale(1.01);
  box-shadow: 0 3px 12px 0 rgba(0, 0, 0, 0.2);
}
```

Maintenant, la `div` a les styles appropriés, y compris un effet de survol. Si vous vous demandez ce que signifie l'attribut `scoped`, il nous permet d'étendre et d'isoler ces styles à ce seul composant. Ainsi, ces styles sont spécifiques à ce composant et n'affecteront aucune autre partie de notre application. Vous me verrez utiliser des styles délimités tout au long de ce cours.

Puisque nous voulons afficher des informations sur l'événement sur cette **EventCard**, nous devons lui donner un événement à afficher. Ajoutons donc cela à l'aide d'un `ref` dans notre section `<script>`.

> *Dans Vue.js 3, avec l'API de composition, **ref** est une fonction utilisée pour déclarer une variable réactive. Lorsque vous utilisez l'API de composition pour construire vos composants, **ref** vous permet de créer une référence à une valeur qui peut être réactive. Cela signifie que chaque fois que cette valeur change, Vue réagira à ces changements et mettra à jour la vue en conséquence.*

📁 **src/components/EventCard.vue**

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

// defineProps({
//   msg: {
//     type: String,
//     required: true,
//   },
// })

const event = ref({
  id: 5928101,
  category: 'animal welfare',
  title: 'Cat Adoption Day',
  description: 'Find your new feline friend at this event.',
  location: 'Meow Town',
  date: 'January 28, 2022',
  time: '12:00',
  petsAllowed: true,
  organizer: 'Kat Laydee',
})
</script>
```

Maintenant, dans le <template>, nous pouvons afficher certaines de ces données d'événement avec des expressions JavaScript, comme suit :

📁 **src/components/EventCard.vue**

```
<template>
  <div class="event-card">
    <h2>{{ event.title }}</h2>
    <span>@{{ event.time }} on {{ event.date }}</span>
  </div>
</template>
```

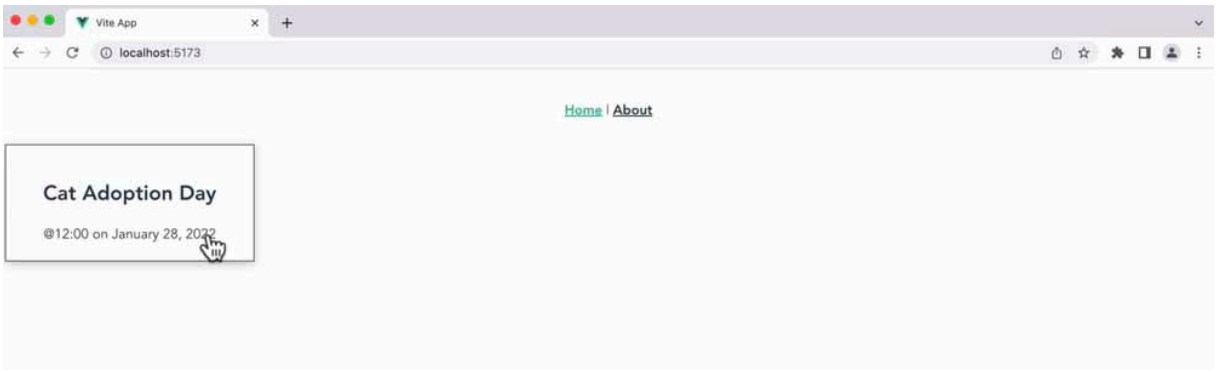
C'est tout pour le composant pour l'instant.

Pour que cette **EventCard** soit affichée, elle doit être placée à un endroit où elle peut être acheminée, comme le fichier **HomeView.vue** dans notre répertoire views. Nous devrons importer **EventCard.vue**, puis nous pourrons l'utiliser dans le modèle.

📁 **src/views/HomeView.vue**

```
<script setup>
import EventCard from '@/components/EventCard.vue'
</script>

<template>
  <div class="home">
    <EventCard />
  </div>
</template>
```

Maintenant, nous devrions voir notre EventCard s'afficher dans le navigateur lorsque nous sommes dans la vue Accueil.



3.7 Refonte pour un cas d'utilisation plus proche de la production

Nous faisons de grands progrès, mais n'oublions pas que nous voulons que l'**EventCard** s'affiche au milieu de la page d'accueil. Et, comme nous aurons éventuellement une collection d'événements que nous tirerons d'un appel à l'API, nous devons faire un peu de refactoring pour rendre ce cas d'utilisation plus prêt pour la production.

Nos étapes de remaniement sont les suivantes

- Déplacer les données d'événements vers le parent (**HomeView.vue**)
- Le parent crée un composant **EventCard** pour chaque événement dans ses données.
- Le parent fournit à chaque **EventCard** son propre événement à afficher
- Le parent affiche les **EventCards** dans un conteneur Flexbox.

Commençons par ce refactor.

3.8 Déplacer les données des événements vers le parent

Notre première étape consiste à supprimer les données d'événement de la carte **EventCard**. Nous allons ensuite ajouter une propriété d'événement à la place, de sorte que le parent puisse fournir à ce composant un objet d'événement à afficher. Il nous reste alors ce code :

📂 **src/components/EventCard.vue**

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

defineProps({
  event: {
    type: Object,
    required: true,
  },
})
</script>

<template>
  <div class="event-card">
    <h2>{{ event.title }}</h2>
    <span>@{{ event.time }} on {{ event.date }}</span>
  </div>
</template>

<style scoped>
.event-card {
  padding: 20px;
  width: 250px;
  cursor: pointer;
  border: 1px solid #39495c;
  margin-bottom: 18px;
}
.event-card:hover {
  transform: scale(1.01);
  box-shadow: 0 3px 12px 0 rgba(0, 0, 0, 0.2);
}
</style>
```

Maintenant que EventCard est configuré pour recevoir un événement, nous pouvons ajouter les données relatives aux événements au parent, **HomeView.vue**.

📁 src/views/HomeView.vue

```
<script setup>
import EventCard from '@/components/EventCard.vue'
import { ref } from 'vue'

const events = ref([
  {
    id: 5928101,
    category: 'animal welfare',
    title: 'Cat Adoption Day',
    description: 'Find your new feline friend at this event.',
    location: 'Meow Town',
    date: 'January 28, 2022',
    time: '12:00',
    petsAllowed: true,
    organizer: 'Kat Laydee',
  },
  {
    id: 4582797,
    category: 'food',
    title: 'Community Gardening',
    description: 'Join us as we tend to the community edible plants.',
    location: 'Flora City',
    date: 'March 14, 2022',
    time: '10:00',
    petsAllowed: true,
    organizer: 'Fern Pollin',
  },
  {
    id: 8419988,
    category: 'sustainability',
    title: 'Beach Cleanup',
    description: 'Help pick up trash along the shore.',
    location: 'Playa Del Carmen',
    date: 'July 22, 2022',
    time: '11:00',
    petsAllowed: false,
    organizer: 'Carey Wales',
  },
])
</script>

<template>
  <div class="home">
    <EventCard />
  </div>
</template>
```

3.9 Le parent crée des composants EventCard

Maintenant que **HomeView.vue** dispose des données relatives aux événements, nous pouvons les utiliser pour créer une nouvelle **EventCard** pour chacun des objets d'événement contenus dans ces données, à l'aide de la directive v-for.

📁 src/views/HomeView.vue

```
<template>
  <div class="home">
    <EventCard v-for="event in events" :key="event.id" :event="event" />
  </div>
</template>
```

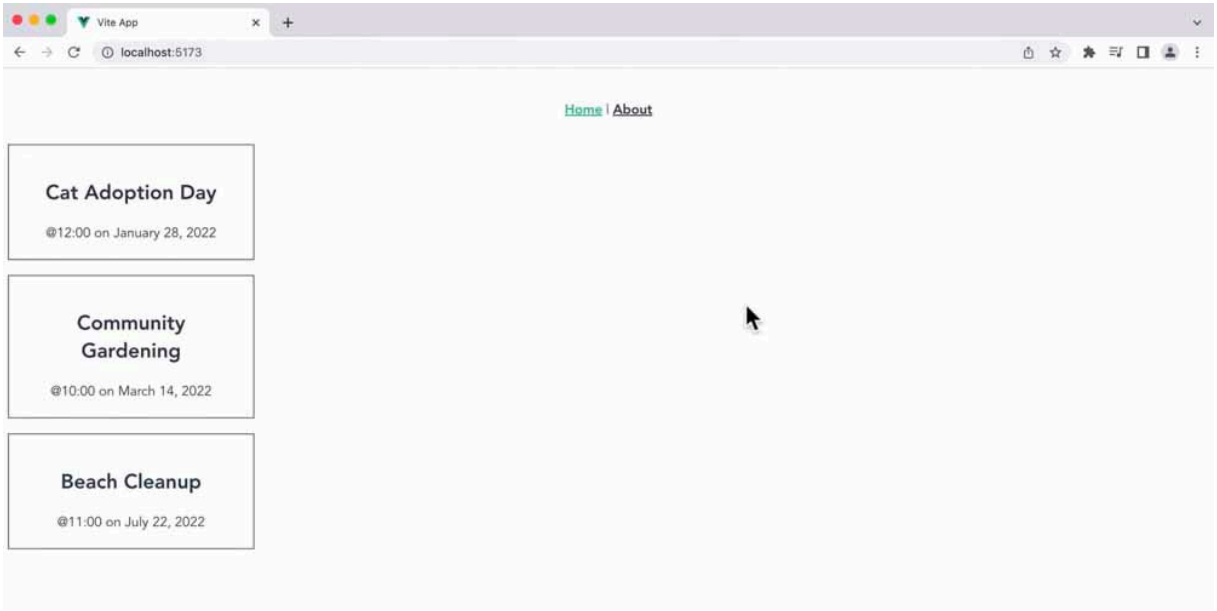
Remarquez que nous lions l'identifiant de l'événement à l'attribut :key. Cela permet à Vue.js d'identifier et de suivre chaque **EventCard**.

3.10 Le parent alimente chaque EventCard avec son propre événement

De plus, lorsque nous itérons sur le tableau d'événements pour créer une nouvelle **EventCard** pour chaque objet d'événement, nous passons cet objet d'événement dans une nouvelle propriété :event que nous avons ajoutée à l'**EventCard**. De cette manière, chaque **EventCard** (carte d'événement) dispose de toutes les données nécessaires pour afficher les informations relatives à son propre événement.

3.11 Le parent affiche les EventCards dans un conteneur Flexbox

Si nous vérifions dans le navigateur, nous constatons que cela fonctionne. Nous avons créé une EventCard pour chaque événement dans les données de **HomeView.vue**.



Enfin, il nous suffit de placer ces événements dans un conteneur Flexbox pour obtenir l'aspect que nous souhaitons. Allons dans le fichier HomeView.vue et changeons le nom de la classe de la div dans laquelle notre EventCard est imbriquée, et ajoutons quelques styles Flexbox.

src/views/HomeView.vue

```
...

<template>
  <div class="events">
    <EventCard v-for="event in events" :key="event.id" :event="event" />
  </div>
</template>

<style scoped>
.events {
  display: flex;
  flex-direction: column;
  align-items: center;
}
</style>
```

Désormais, nos **EventCard**s (cartes d'événements) seront affichées dans une colonne alignée au centre.

3.12 Qu'en est-il des styles globaux ?

Jusqu'à présent, nous avons discuté des styles délimités et de la manière dont l'attribut `scoped` nous permet d'ajouter des styles qui ciblent le composant spécifique qui nous intéresse. Mais qu'en est-il des styles globaux que nous voulons appliquer à l'ensemble de notre application ? Bien qu'il existe différentes façons d'y parvenir, la manière la plus simple de commencer est de se rendre dans le fichier **App.vue**. N'oubliez pas qu'il s'agit du composant racine de notre application.

Et supprimez le `scoped` de `<style>` :

src/App.vue

```
<style>
...
</style>
```

Maintenant, tous les styles dans **App.vue** sont appliqués à l'ensemble de l'application. Ici, nous pourrions ajouter une nouvelle règle. Comme cela :

src/App.vue

```
<style>
...
h2 {
  font-size: 20px;
}
</style>
```

Désormais, tous les `h2` de notre application auront une taille de police de `20px`. Puisque le modèle de notre **EventCard** a un `h2`, cet élément recevra ce nouveau style global.

src/components/EventCard.vue

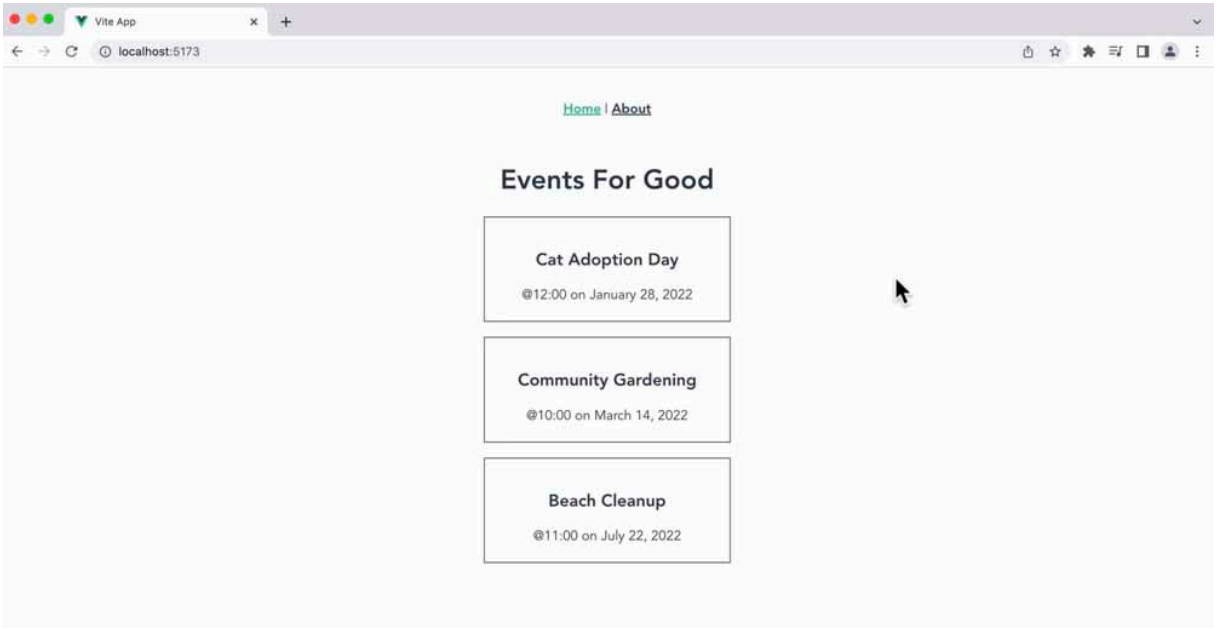
```
<template>
  <div class="event-card">
    <h2>{{ event.title }}</h2>
    <span>@{{ event.time }} on {{ event.date }}</span>
  </div>
</template>
```

En parlant d'éléments globaux dans notre application Vue, que se passerait-il si nous ajoutions quelque chose comme un `h1` au modèle de notre **App.vue** ?

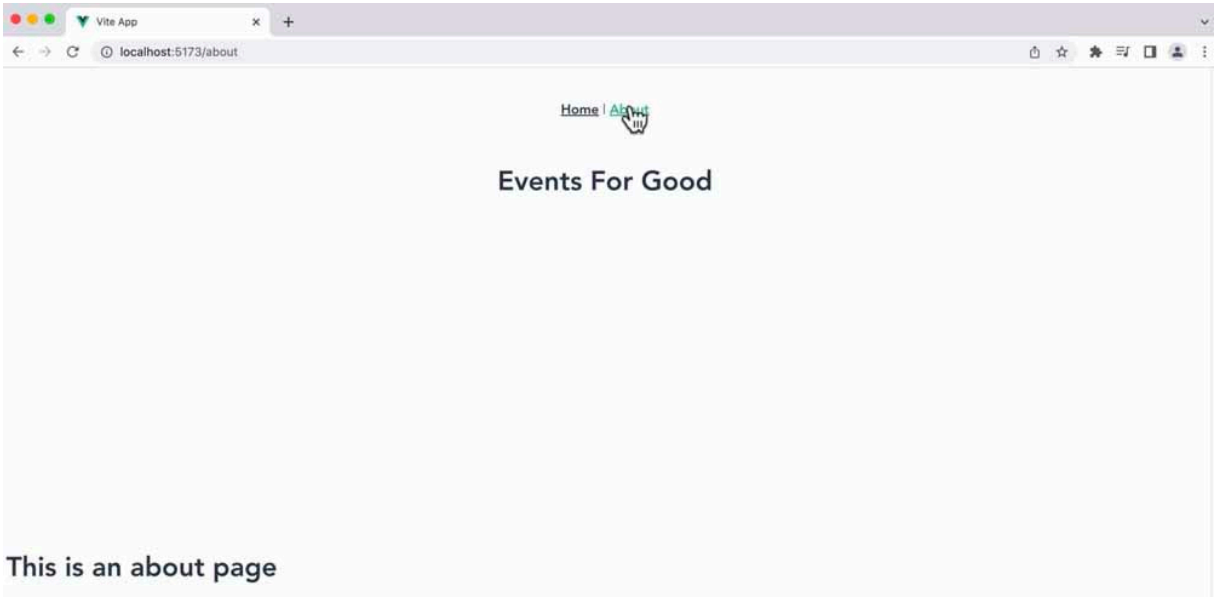
📂 **src/App.vue**

```vue
<template>
  <div id="layout">
    <header>
      <div class="wrapper">
        <nav>
          <RouterLink to="/">Home</RouterLink> |
          <RouterLink to="/about">About</RouterLink>
        </nav>
      </div>
    </header>
    <h1>Events For Good</h1> <!-- new element -->
    <RouterView />
  </div>
</template>
```

Entrons dans le navigateur et jetons-y un coup d'œil.



Nous constatons plusieurs choses. Tout d'abord, notre conteneur Flexbox fonctionne (✅) et les titres des événements sont maintenant un peu plus grands (`20px`) grâce à la nouvelle règle de style globale `h2 et pas h4` que nous avons ajoutée. Et remarquez ce qui se passe lorsque nous naviguons vers la route "À propos".



(Remarquez que le contenu de cette page *"Ceci est une page à propos"* apparaît dans le coin, nous corrigerons ce positionnement dans un instant).

Nous voyons toujours ce `h1` affichant "Events For Good". Cela nous indique que nous pouvons placer dans le modèle de notre **App.vue** du contenu que nous voulons afficher globalement dans toutes les vues de notre application. Cela peut être utile pour des choses comme une barre de recherche, un en-tête, ou bien sûr une barre de navigation comme celle que nous avons déjà ici.

Mais pour notre cas d'utilisation, nous n'avons pas besoin que ce titre s'affiche dans chaque vue, nous allons donc le placer dans le fichier **HomeView.vue**.

📁 **src/views/HomeView.vue**

```
...
<template>
  <h1>Events For Good</h1>
  <div class="events">
    <EventCard v-for="event in events" :key="event.id" :event="event" />
  </div>
</template>
...
```

Désormais, ce titre n'apparaîtra que sur la route Home.

Enfin, pour résoudre le problème de positionnement de la page About, il suffit de supprimer les styles par défaut suivants dans **AboutView.vue** :

```vue
<template>
  <div class="about">
    <h1>This is an about page</h1>
  </div>
</template>

<style>
/* @media (min-width: 1024px) {
  .about {
    min-height: 100vh;
    display: flex;
    align-items: center;
  }
} */
</style>
```

Il devrait maintenant ressembler à ceci :

3.13 Résumons

Nous avons couvert beaucoup de choses. Nous avons appris ce qu'est un composant .vue à fichier unique, comment il est composé (avec les styles scoped et global) et comment commencer à utiliser ces composants .vue pour construire une application Vue. Dans la prochaine leçon, nous allons nous plonger plus profondément dans les éléments essentiels de Vue Router pour mieux comprendre comment configurer la navigation dans l'application. Au plaisir de vous y retrouver !

4 L'essentiel du Vue Router

Dans cette leçon, nous allons vous présenter les outils que Vue utilise pour naviguer entre les pages (ou les vues) de notre application. Nous aborderons les points suivants :

- Qu'est-ce que le routage côté client ?
- Qu'est-ce qu'une application à page unique ?
- Comment le routeur Vue est-il configuré dans une application Vue ?
- Nous personnaliserons ensuite les routeurs dans notre application d'exemple.

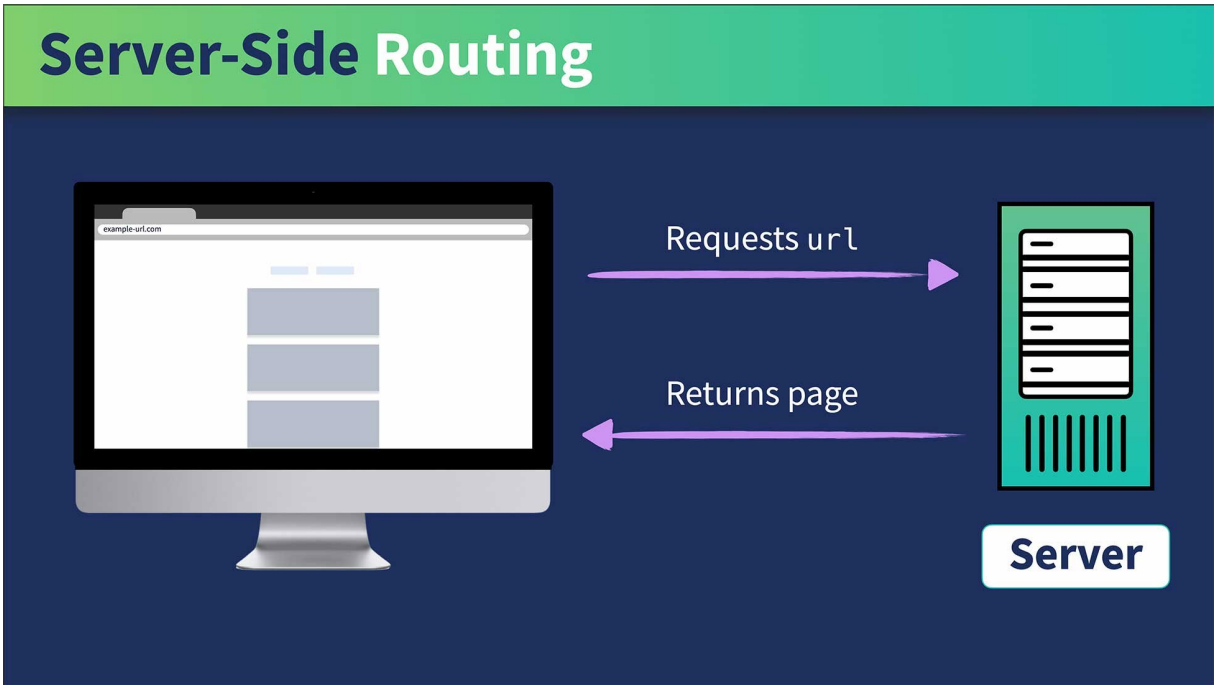
Ressources pour cette leçon

Code source :

- ✓ Code de départ
- ✓ Code final
- ✓ Mode historique

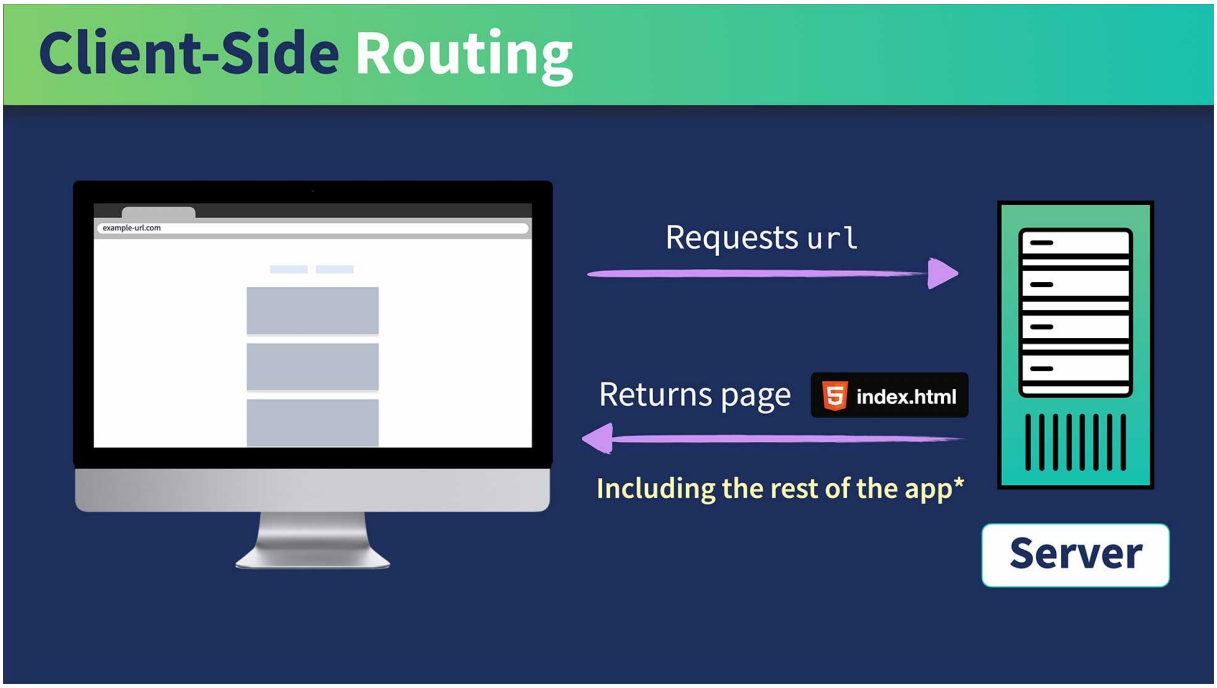
4.1 Routage côté serveur ou côté client

Lorsqu'il s'agit de sites web, nous connectons généralement nos pages à l'aide de liens. Un lien est cliqué, il appelle le serveur pour la page suivante, et cette page est chargée.



On parle de "routage côté serveur" parce que le client adresse une demande au serveur à chaque changement d'URL.

Lorsqu'il s'agit de Vue, beaucoup choisissent le routage côté client, ce qui signifie que le routage s'effectue dans le navigateur lui-même à l'aide de JavaScript.



Dans de nombreux cas, la vue de notre application que nous devons afficher a déjà été chargée dans le navigateur, nous n'avons donc pas besoin d'aller la chercher sur le serveur. Vue Router met simplement à jour la partie de l'application qui est actuellement affichée.

https://firebasestorage.googleapis.com/v0/b/vue-mastery.appspot.com/o/flamelink%2Fmedia%2F4.opt.gif?alt=media&token=d8f79fe9-dc02-4b1d-8ee5-91fd87e953db

En fait, avec un tel routage, notre application fonctionne comme une application à page unique. Qu'est-ce que cela signifie exactement ?

4.2 Les applications à page unique (Single Page Application)

Une application à page unique (SPA) est une application web qui se charge à partir d'une seule page et qui met à jour cette page de manière dynamique au fur et à mesure que l'utilisateur interagit avec l'application. Dans notre cas, tout est chargé à partir du fichier index.html de notre projet. Dans la leçon 2 (Create-Vue - créer le projet), nous avons examiné ce fichier et vu qu'il contenait cette `div` avec l'identifiant `#app`.

index.html

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

Nous avons également jeté un coup d'œil à **main.js** et appris que lorsque notre application est créée, elle est montée dans cette `div` avec l'identifiant `#app`.

src/main.js

```
const app = createApp(App)

app.use(createPinia())
app.use(router)

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

En d'autres termes, le fichier **index.html** est la "page unique" de notre application à page unique, où tout le code de l'application est monté. Vue Router permet le routage côté client afin que nous puissions naviguer et afficher différentes "vues" de notre application.

4.3 package.json

Toutes les dépendances de notre application sont suivies dans notre fichier **package.json**. Si nous jetons un coup d'œil rapide à l'intérieur, nous voyons que le CLI de Vue a déjà inséré Vue Router en tant que dépendance parce que nous avons choisi de l'ajouter lorsque nous avons configuré notre projet.

package.json

```
...
"dependencies": {
  "axios": "^1.2.1",
  "pinia": "^2.0.21",
  "vue": "^3.2.38",
  "vue-router": "^4.1.5"
},
..
```

Ceci indique à notre application d'utiliser une version de vue-router compatible avec la version 4.1.5 de vue-router. (Votre numéro de version peut être différent selon la date à laquelle vous avez suivi ce cours).

Plus tôt dans la leçon 2, nous avons exécuté `npm install`, qui a été envoyé à NPM, et a installé la bibliothèque vue-router dans le répertoire **node_modules** de notre application.

Jetons maintenant un coup d'œil dans le répertoire **router** pour voir comment Vue Router fonctionne.

4.4 Comment Vue Router est configuré

Dans le répertoire router, nous trouvons le fichier index.js de notre routeur. Au début de ce fichier, nous importons la bibliothèque vue-router.

src/router/index.js

```
import { createRouter, createWebHistory } from 'vue-router'
```

Nous importons ensuite un composant que nous utiliserons dans nos routes :

```
import HomeView from '../views/HomeView.vue'
```

Ensuite, nous utilisons cette route :

```
routes: [
  {
    path: '/',
    name: 'home',
    component: HomeView
  },
  {
    path: '/about',
    name: 'about',
    ...// Skipping this part, which we will come back later
  }
]
```

Le `path` indique l'itinéraire réel, en termes d'URL, vers lequel l'utilisateur sera dirigé. Dans cette première route, il n'y a que le `/`, ce qui signifie qu'il s'agit de la racine, de la page d'accueil de notre application et de ce que les gens voient lorsqu'ils se rendent sur notre domaine à l'adresse example.com.

Le `name` nous permet de donner un nom à cette route afin que nous puissions utiliser ce nom dans toute notre application pour faire référence à cette route (nous y reviendrons plus tard dans le cours).

Le `component` nous permet de spécifier le composant à rendre sur cette route. Notez que `HomeView` a été importé en haut du fichier. Ainsi, le composant `HomeView` sera rendu chaque fois que l'URL du navigateur se terminera par un `/` sans rien après.

En jetant un coup d'œil au deuxième objet route, nous pouvons voir qu'il a un chemin différent :

📁 src/router/index.js

```
{
  path: '/about',
  name: 'about',
  // route level code-splitting
  // this generates a separate chunk (About.[hash].js) for this route
  // which is lazy-loaded when the route is visited.
  component: () => import('../views/AboutView.vue')
}
```

Lorsque l'URL du navigateur se termine par `/about`, le composant "`About`" est affiché.

Vous avez probablement remarqué que le composant est importé différemment. Plutôt que de l'importer au début du fichier comme nous l'avons fait avec `HomeView`, nous l'importons seulement lorsque la route est appelée. Comme indiqué dans les commentaires, cela va générer un fichier `about.js` séparé, qui ne sera chargé dans le navigateur de quelqu'un que lorsqu'il naviguera vers `/about`. Il s'agit d'une optimisation des performances qui n'est pas nécessaire dans notre petite application simple. Mais lorsqu'une application se développe, il peut être utile de répartir la façon dont elle est chargée dans différents fichiers JavaScript, qui ne sont chargés que lorsqu'ils sont nécessaires.

Nous utilisons `createRouter` pour créer le routeur, en lui indiquant d'utiliser l'API Historique du navigateur et en envoyant les `routes`, avant de l'exporter à partir de ce fichier.

📁 src/router/index.js

```
const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes: [
    ...
  ]
})

export default router
```

Nous avons donc défini les deux vues différentes entre lesquelles notre application va pouvoir naviguer, mais nous n'avons pas encore chargé ce routeur dans notre instance Vue. Rappelez-vous, notre application entière est chargée à partir de notre **main.js**, et si nous regardons à l'intérieur de ce fichier, nous pouvons voir que nous importons notre fichier **./router/index.js**, qui apporte ce que nous avons exporté à partir de **router.js**.

📄 src/main.js

```
import router from './router' // <-- This imports index.js from the /router directory
```

Et dans **main.js**, vous remarquerez que nous disons à notre instance Vue d'utiliser le routeur que nous avons importé :

📄 src/main.js

```
app.use(router)
```

Jusqu'à présent, tout va bien. Notre routeur est maintenant configuré. Mais où se trouve la fonctionnalité ajoutée pour permettre à l'utilisateur de naviguer vers différentes parties de l'application ?

4.5 Composants intégrés du routeur Vue

En regardant dans **App.vue**, nous trouverons une `div` avec l'id de `#nav`. À l'intérieur de celle-ci se trouvent des `router-links`, qui sont des composants globaux spécifiques à Vue Router auxquels nous avons accès.

src/App.vue

```
<header>
  <div class="wrapper">
    <nav>
      <RouterLink to="/">Home</RouterLink> |
      <RouterLink to="/about">About</RouterLink>
    </nav>
  </div>
</header>
```

Et en dessous, il y a cet autre composant Vue Router :

src/App.vue

```
<RouterView />
```

Que se passe-t-il ici ?

`<RouterLink>` est un composant (de la bibliothèque vue-router) dont le rôle est de créer un lien vers une route spécifique. Vous pouvez les considérer comme une balise d'ancrage améliorée.

`<RouterView/>` est essentiellement un espace réservé où le contenu de notre composant "view" sera rendu sur la page.

Lorsqu'un utilisateur clique sur le lien Home, où est-il dirigé ? La réponse se trouve dans l'attribut `to` : `<RouterLink to="/">`

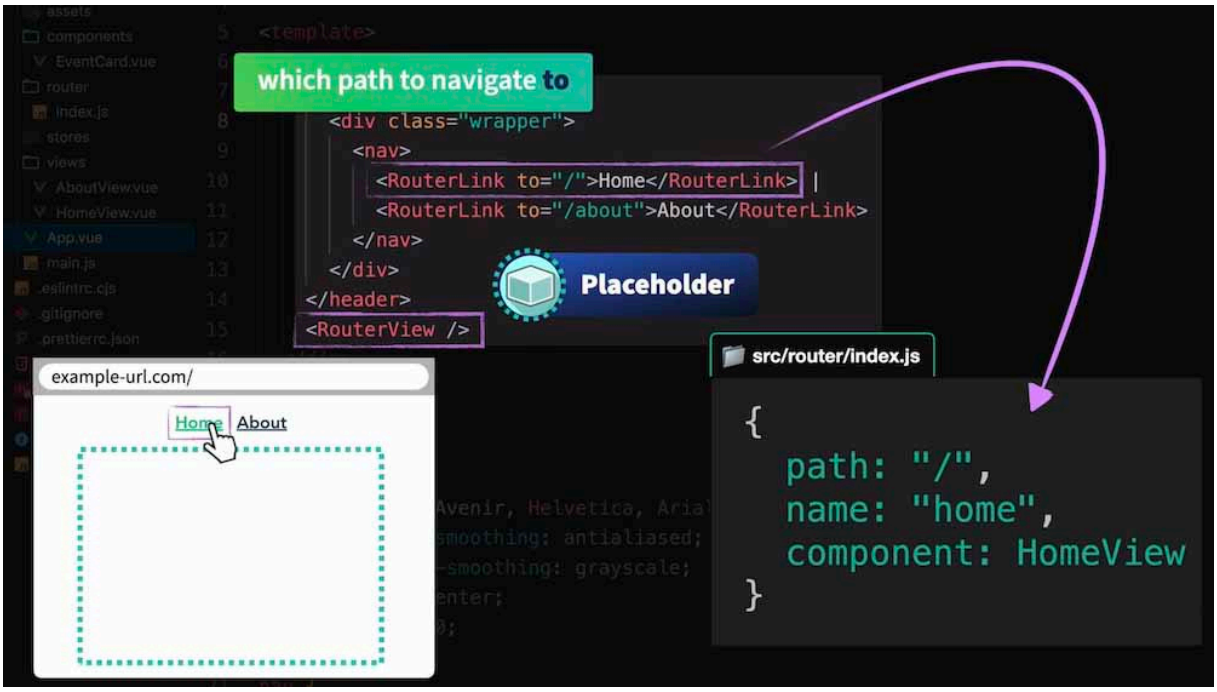
Il est dirigé vers `/`, ce qui signifie qu'en fonction de la route définie dans **router.js**, le composant Home sera chargé.

src/router/index.js

```
{
  path: '/',
  name: 'home',
  component: HomeView
},
```

Mais où sera-t-il chargé exactement ? La réponse est : dans le `<RouterView/>`

Encore une fois, il s'agit simplement d'un espace réservé qui est remplacé par le composant "vue" vers lequel nous nous dirigeons, tel que **HomeView** ou **AboutView**.



Personnalisation de notre **application d'exemple**

Maintenant que nous comprenons les fondements de Vue Router, nous sommes prêts à personnaliser les routes de notre application d'exemple. Notre liste de tâches comprend

1. Renommer **HomeView.vue** en **EventListView.vue**
2. Personnaliser la route pour **EventListView**
3. Mettre à jour **AboutView.vue**
4. Reconfigurer la route **About**

4.6 Renommer HomeView.vue en EventListView.vue

Parce que **HomeView** est en fait une liste d'événements, renommons-le en **EventListView**.

src/views/EventListView.vue

```
<script setup>
import EventCard from '@/components/EventCard.vue'
import { ref } from 'vue'

const events = ref([
...
```

4.7 Personnaliser la route pour EventListView

Maintenant que le fichier a été renommé, nous devons modifier notre déclaration d'importation dans notre fichier routeur et modifier l'objet route lui-même.

📁 **src/router/index.js**

```
import { createRouter, createWebHistory } from 'vue-router'
import EventListView from '../views/EventListView.vue' // imported renamed SFC

const routes = [
  {
    path: '/',
    name: 'event-list',
    component: EventListView
  },
  ...
]
```

4.8 Mise à jour de AboutView.vue

Ajoutons maintenant un peu de personnalisation à la page About pour qu'elle corresponde à notre exemple d'application, en ajoutant ce texte de description.

```
<template>
  <div class="about">
    <h1>A site for events to better the world.</h1>
  </div>
</template>
```

4.9 Reconfigurer la route About

Puisque nous n'avons pas besoin d'utiliser la séparation de code au niveau de la route pour notre application, nous allons simplifier l'objet de la route About, comme suit :

📁 **src/router/index.js**

```
{
  path: '/about',
  name: 'about',
  component: AboutView
}
```

(N'oubliez pas d'importer **AboutView.vue**)

À ce stade, notre fichier routeur nouvellement personnalisé ressemble à ceci :

📂 src/router/index.js

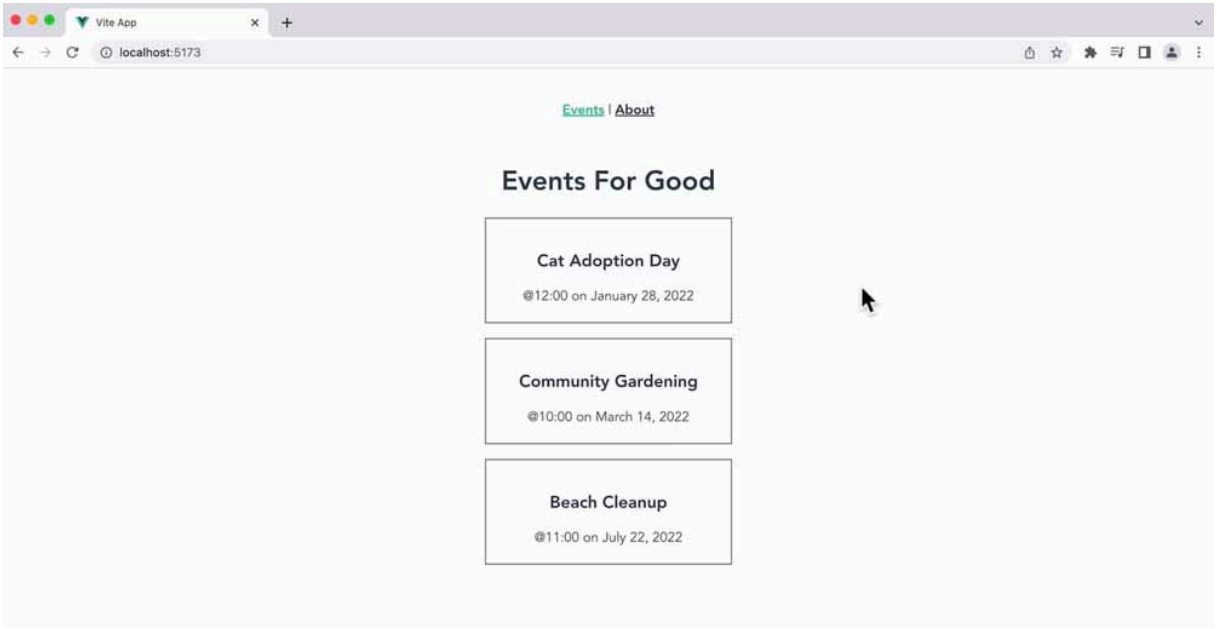
```
import { createRouter, createWebHistory } from 'vue-router'
import EventListView from '../views/EventListView.vue'
import AboutView from '../views/AboutView.vue'

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: '/',
      name: 'event-list',
      component: EventListView,
    },
    {
      path: '/about',
      name: 'about',
      component: AboutView,
    },
  ],
})

export default router
```

4.10 Dernière étape

Puisque nous affichons maintenant cette liste d'événements sur ce qui était auparavant notre page d'accueil, mettons à jour le code HTML interne du RouterLink pour cette vue.

📄 App.vue

```
<template>
  <div id="layout">
    <header>
      <div class="wrapper">
        <nav>
          <RouterLink to="/">Events</RouterLink> |
          <RouterLink to="/about">About</RouterLink>
        </nav>
      </div>
    </header>
    <RouterView />
  </div>
</template>
```

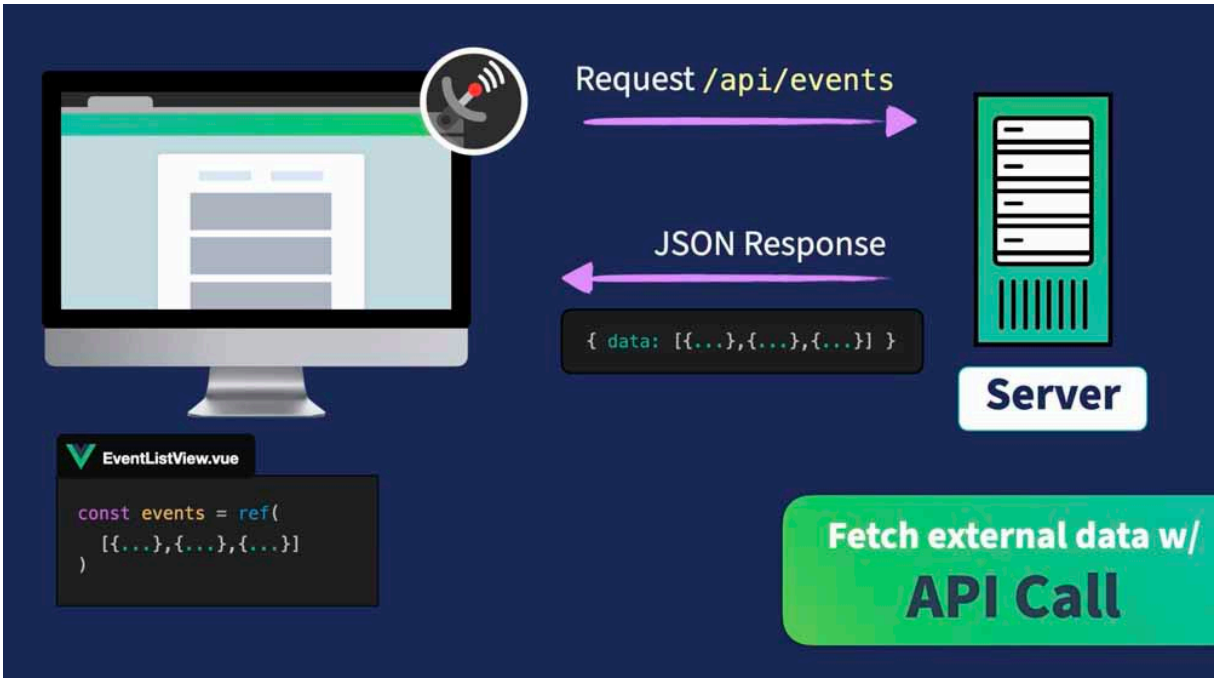


4.11 Prochaine leçon

Dans la prochaine leçon, nous apprendrons à récupérer nos événements en tant que données externes que nous tirons d'un appel API à l'aide d'Axios.

5 Appels API avec Axios

Dans l'état actuel de notre application, les événements que nous affichons sont simplement codés en dur dans les données du composant **EventListView.vue**. Dans une application réelle, il y aurait probablement une sorte de base de données d'événements que nous pourrions extraire. Notre application ferait une requête pour les événements, le serveur répondrait avec ces événements (sous forme de JSON), et nous prendrions ces événements et les définirions comme les données de notre composant (ref), que nous afficherions ensuite dans la vue.



Nos tâches dans cette leçon sont donc les suivantes :

- Créer une base de données fictive pour héberger nos événements
- Installer une bibliothèque (Axios) pour faire des appels API
- Implémenter un appel API `getEvents()`
- Refondre notre code API en une couche de service

> **Ressources pour cette leçon**
>
> **Code source :**
>
> ✓ Code de départ
> ✓ Code final

5.1 Notre base de données fictive

Pour créer notre base de données fictive, nous allons utiliser My JSON Server, une solution simple qui ne nécessite aucune installation. Nous avons juste besoin d'un repo Github avec un fichier **db.json**. Si vous avez suivi le codage, vous avez peut-être déjà remarqué que j'ai ajouté un fichier **db.json** au repo du cours :

db.json


```json
{
  "events": [
    {
      "id": 123,
      "category": "animal welfare",
      "title": "Cat Adoption Day",
      "description": "Find your new feline friend at this event.",
      "location": "Meow Town",
      "date": "January 28, 2022",
      "time": "12:00",
      "organizer": "Kat Laydee"
    },
    {
      "id": 456,
      "category": "food",
      "title": "Community Gardening",
      "description": "Join us as we tend to the community edible plants.",
      "location": "Flora City",
      "date": "March 14, 2022",
      "time": "10:00",
      "organizer": "Fern Pollin"
    },
    {
      "id": 789,
      "category": "sustainability",
      "title": "Beach Cleanup",
      "description": "Help pick up trash along the shore.",
      "location": "Playa Del Carmen",
      "date": "July 22, 2022",
      "time": "11:00",
      "organizer": "Carey Wales"
    }
  ]
}
```


Ce code devrait vous sembler très familier, puisqu'il s'agit d'une version JSON des données d'événements ou events qui se trouvent actuellement dans les données locales de notre composant **EventListView.vue**. Ce sont les données que nous allons bientôt récupérer avec notre nouvel appel API.

Afin d'accéder à notre serveur fictif, nous irons à l'url :

my-json-server.typicode.com/{GithubUserName}/{RepoName}

(Évidemment, si vous créez votre propre fichier **db.json** dans le repo de votre compte Github, vous voudrez remplir les blancs pour votre **UserName** et **RepoName** ici).

L'ajout de '**/events**' à la fin de l'URL nous permet de cibler spécifiquement les données d'événements, de sorte que my-json-server.typicode.com/{GithubUserName}/{RepoName}/events est l'URL que nous utiliserons bientôt pour effectuer notre appel.

5.2 Axios pour les appels d'API

Maintenant que nous avons notre base de données fictive et que nous savons quelle URL appeler, nous sommes prêts à installer une bibliothèque pour nous aider à faire des appels d'API. Nous utiliserons la bibliothèque Axios.

Depuis le terminal, après avoir accédé à la racine de votre projet, exécutez :

```
npm install axios
```

Pourquoi utiliser Axios ? Elle est très populaire et inclut de nombreuses fonctionnalités, notamment :

- Requêtes GET, POST, PUT et DELETE
- Ajouter l'authentification à chaque requête
- Définir des délais d'attente si les requêtes prennent trop de temps
- Configuration des valeurs par défaut pour chaque requête
- Intercepter les requêtes pour créer un logiciel intermédiaire
- Gérer correctement les erreurs et les demandes d'annulation
- Sérialiser et désérialiser correctement les requêtes et les réponses
- Et bien d'autres choses encore...

Maintenant que nous l'avons installé, nous pouvons commencer à l'utiliser et écrire notre premier appel d'API.

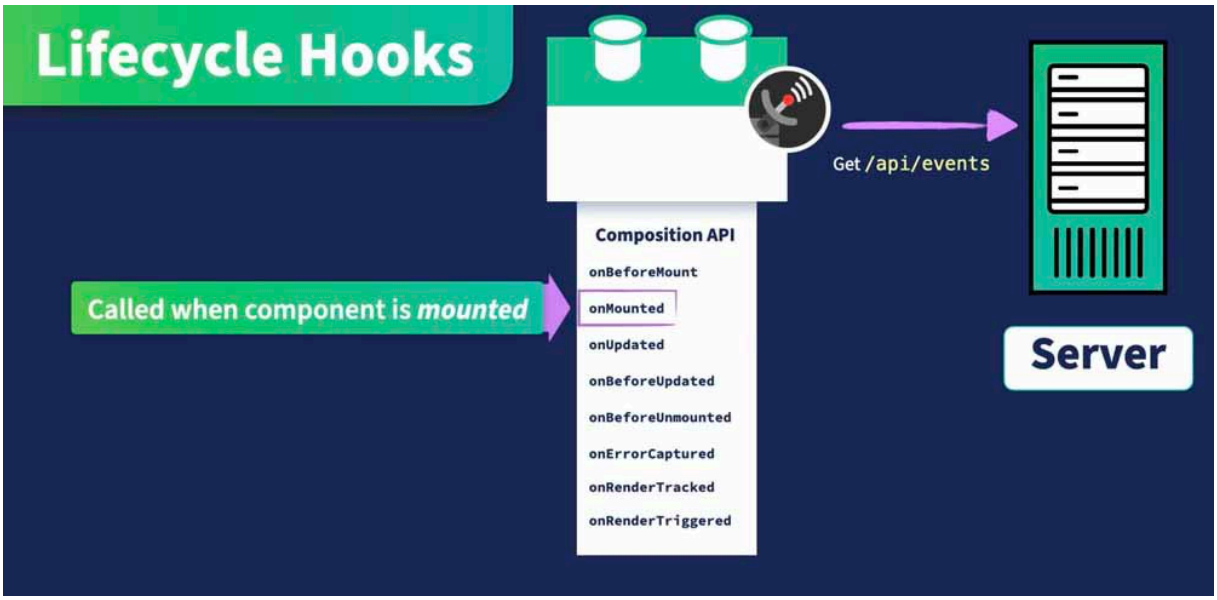
5.3 Implémentation d'Axios pour obtenir des événements

Pour écrire notre appel API, nous allons nous rendre dans le composant **EventListView.vue**, supprimer les données d'événements codées en dur (remplacer par null), importer Axios, puis importer/ajouter le hook de cycle de vie onMounted

📂 **src/views/EventListView.vue**

```
<script setup>
import { ref, onMounted } from 'vue'
import EventCard from '@/components/EventCard.vue'
import axios from 'axios'

const events = ref(null)

onMounted(() => {
  // get events from mock db when component is created
})
</script>
```

Si les hooks de cycle de vie sont un concept nouveau pour vous, il vous suffit de comprendre qu'un composant a un cycle de vie et que différents hooks (ou méthodes) sont exécutés à ces différentes étapes de son cycle de vie. Par exemple, avant qu'il ne soit créé, lorsqu'il est créé, avant qu'il ne soit monté, lorsqu'il est monté, etc.



Dans notre cas, nous voulons faire notre appel à l'API et obtenir nos événements lorsque le composant est onMounted, nous allons donc exécuter la méthode get disponible sur axios, en passant l'url my-json-server comme argument (l'endroit d'où nous voulons l'obtenir).

📂 **src/views/EventListView.vue**

```
import { ref, onMounted } from 'vue'
import EventCard from '@/components/EventCard.vue'
import axios from 'axios'

const events = ref(null)

onMounted(() => {
  axios
    .get('https://my-json-server.typicode.com/Code-Pop/Real-World_Vue-3/events')
    .then((response) => {
      events.value = response.data
    })
    .catch((error) => {
      console.log(error)
    })
})
```

Axios étant une bibliothèque basée sur les promesses et fonctionnant de manière asynchrone, nous devons attendre que la promesse renvoyée par la requête get soit résolue avant de continuer. C'est pourquoi nous avons ajouté le .then, qui nous permet d'attendre la réponse et de mettre notre ref d'événement events local égal à celle-ci.

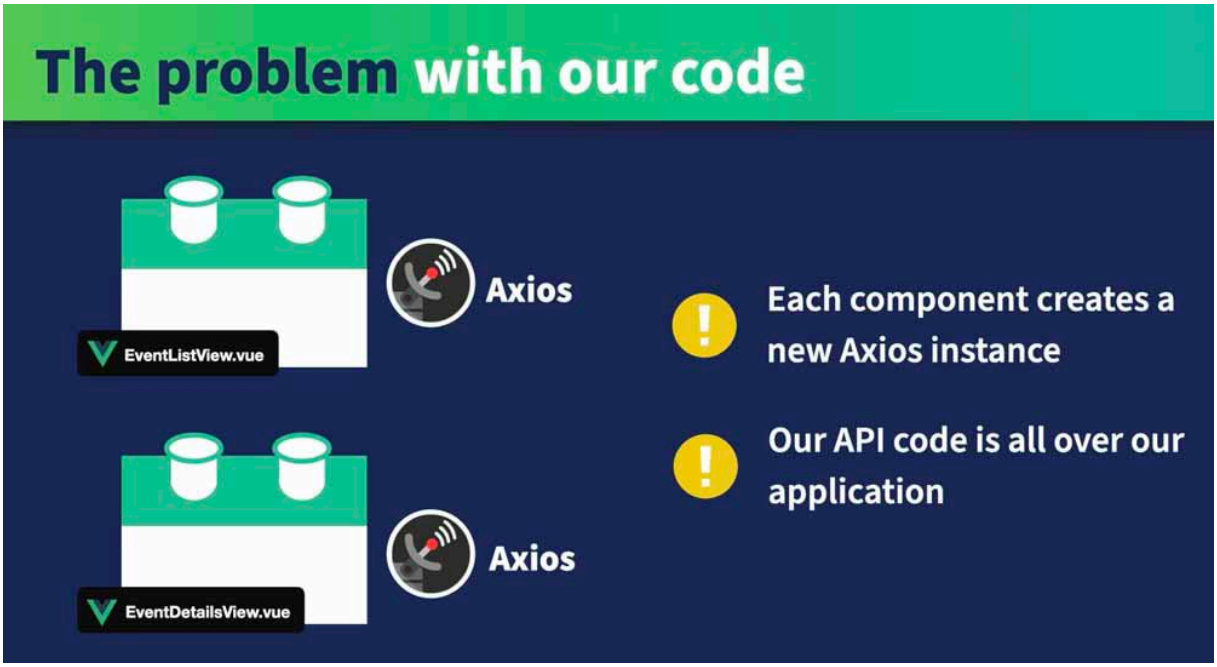
Parce que nous voulons saisir toutes les erreurs qui se produisent, nous avons également ajouté .catch et nous enregistrons simplement l'erreur dans la console. Bien qu'il existe des solutions au niveau de la production pour la gestion des erreurs, nous ne nous y attarderons pas dans ce cours. Cette solution répond à nos besoins pour cette implémentation simple.

(Au cas où vous vous poseriez la question : nous aurions pu utiliser la syntaxe alternative async / await au lieu de .then. J'ai choisi cette syntaxe car je suppose que plus de gens sont déjà familiers avec elle, et je la trouve un peu moins abstraite pour les nouveaux venus. Les deux fonctionnent très bien, et je vous encourage à écrire vos appels asynchrones comme vous et votre équipe le préférez).

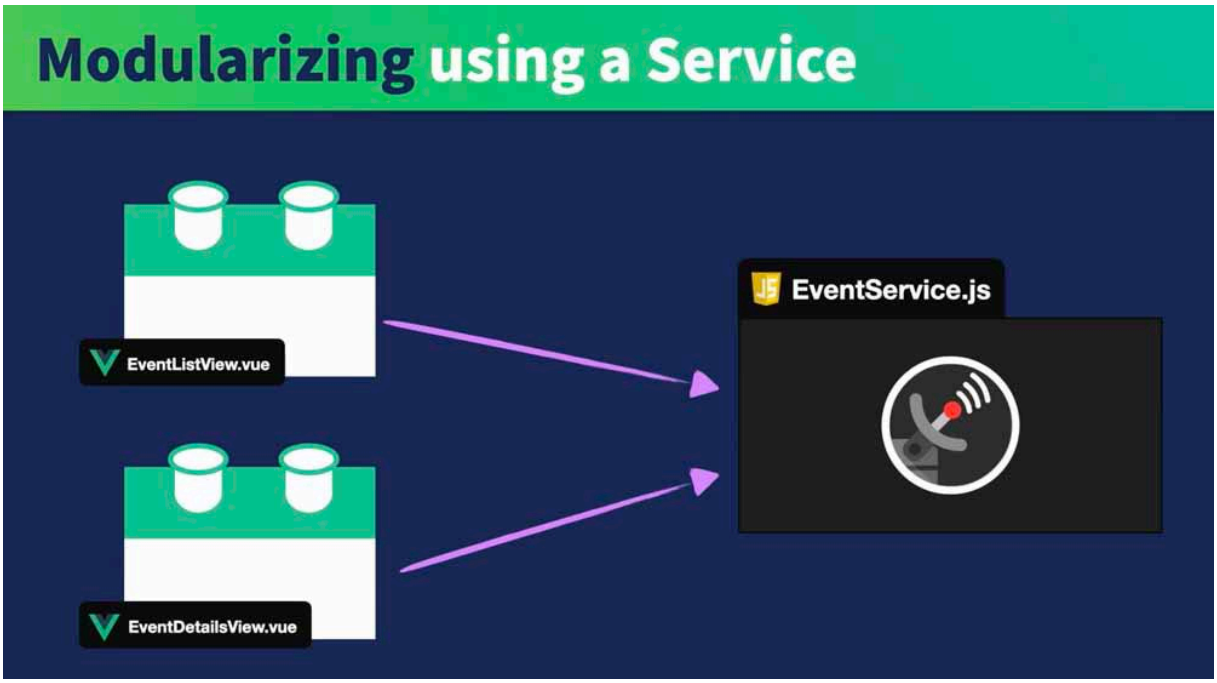
Si nous vérifions ceci dans le navigateur, nous devrions maintenant voir nos événements s'afficher, tirés en douceur depuis notre serveur fictif nouvellement implémenté.

5.4 Réorganiser notre code en une couche de service

Bien que nous ayons fait de grands progrès, il y a un problème avec notre code. Actuellement, nous importons Axios dans le composant **EventListView.vue**. Mais dans la prochaine leçon, nous allons créer un nouveau composant qui affichera les détails de notre événement. Ce nouveau composant devra également faire un appel à l'API. Si nous importons Axios dans chaque composant qui en a besoin, nous créons inutilement une nouvelle instance d'Axios à chaque fois. Le code de l'API étant omniprésent dans notre application, cela devient désordonné et rend notre application plus difficile à déboguer.



Une solution plus propre et plus évolutive consiste à modulariser le code de notre API dans une couche de service. Pour ce faire, nous allons créer un dossier **services** dans notre répertoire src et créer un nouveau fichier **EventService.js**.



📁 src/services/EventService.js

```
import axios from 'axios'

const apiClient = axios.create({
  baseURL: 'https://my-json-server.typicode.com/Code-Pop/Real-World_Vue-3',
  withCredentials: false,
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json'
  }
})

export default {
  getEvents() {
    return apiClient.get('/events')
  }
}
```

En haut, nous importons Axios. En dessous, nous avons ajouté une constante `apiClient`, qui contient notre instance singulière d'Axios. Comme vous pouvez le voir, nous avons défini une `baseURL` et d'autres configurations qu'Axios doit utiliser pour communiquer avec notre serveur.

Maintenant que nous avons mis cela en place, nous pouvons exporter une méthode qui récupère nos événements, en utilisant notre nouveau `apiClient` Axios.

📁 src/services/EventService.js

```
...

export default {
  getEvents() {
    return apiClient.get('/events')
  }
}
```

Comme vous pouvez le voir, nous avons toujours accès à la méthode `get` d'Axios, et nous passons `'/events'` comme argument lors de cet appel. Cette chaîne sera ajoutée à notre `baseURL`, de sorte que la requête sera faite à : 'https://my-json-server.typicode.com/Code-Pop/Real-World_Vue-3/events'.

Il ne nous reste plus qu'à utiliser ce nouvel **EventService** dans notre composant **EventListView.vue**, en supprimant l'importation d'Axios, en important l'EventService et en lançant son appel `getEvents()`.

📁 src/views/EventListView.vue

```
<script setup>
import { ref, onMounted } from 'vue'
import EventCard from '@/components/EventCard.vue'
~~import axios from 'axios'~~
import EventService from '@/services/EventService.js'

const events = ref(null)

onMounted(() => {
  EventService.getEvents()
    .then((response) => {
      events.value = response.data
    })
    .catch((error) => {
      console.log(error)
    })
})
</script>

<template>
  <h1>Events For Good</h1>
  <div class="events">
    <EventCard v-for="event in events" :key="event.id" :event="event" />
  </div>
</template>

<style scoped>
.events {
  display: flex;
  flex-direction: column;
  align-items: center;
}
</style>
```

Ainsi, nous avons remanié le code de notre API pour en faire une couche de service modulaire.

5.5 À suivre

Lorsque nous visualisons nos événements dans le navigateur, les cartes d'événement sont cliquables. Ne serait-il pas agréable de pouvoir cliquer sur ces cartes pour obtenir plus de détails sur

l'événement en question ? Dans la prochaine leçon, nous apprendrons comment y parvenir grâce aux capacités de routage dynamique de Vue Router.

6 Routage dynamique

Dans cette leçon, nous allons ajouter une fonctionnalité permettant à un utilisateur de cliquer sur l'une des cartes d'événement affichées sur notre page d'accueil et d'être dirigé vers une vue qui présente plus de détails sur cet événement. En d'autres termes, nous allons mettre en œuvre un comportement de routage dynamique. Nous aborderons cette nouvelle fonctionnalité en deux parties.

6.1 Partie 1 : Ce que nous allons réaliser

- Créer un nouveau composant **EventDetailsView** pour afficher les détails de l'événement.
- Ajouter un nouvel appel API pour récupérer un événement unique par son identifiant (c'est l'événement dont nous afficherons les détails)
- Ajouter une route pour le nouveau composant **EventDetailsView**
- Rendre **EventCard** cliquable afin que nous puissions accéder à cette nouvelle route **EventDetailsView**

Ressources pour cette leçon

Code source :

- ✓ Code de départ
- ✓ Code final

6.1.1 Création du composant EventDetailsView

Tout d'abord, nous allons créer le composant pour afficher les détails de l'événement, en l'ajoutant à notre répertoire de vues.

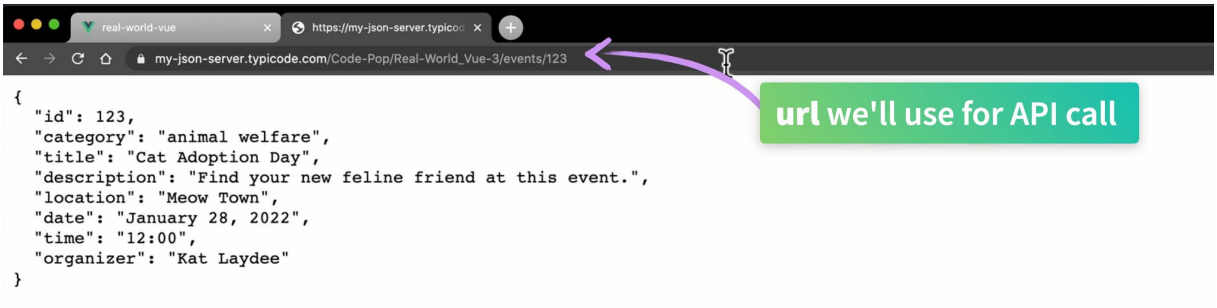
src/views/EventDetailsView.vue

```vue
<script setup>
import { ref, onMounted } from 'vue'

const event = ref(null)

onMounted(() => {
  // fetch event (by id) and set local event data
})
</script>

<template>
  <div>
    <h1>{{ event.title }}</h1>
    <p>{{ event.time }} on {{ event.date }} @ {{ event.location }}</p>
    <p>{{ event.description }}</p>
  </div>
</template>
```

Il restitue les détails de la référence de l'événement. Cet événement est récupéré à partir d'un appel à l'API qui le récupère, par son identifiant. Reprenons notre base de données fictive pour voir comment la récupérer.

6.1.2 Ajout d'un appel à l'API pour récupérer un événement par son identifiant

Remarquez ce qui se passe lorsque nous appelons notre url my-json-server, cette fois avec un identifiant à la fin *(.../events/123*). Cela cible un seul événement, dont l'identifiant correspond à la fin de notre url : 123.



C'est le type d'url que nous utiliserons pour récupérer un événement unique, qui se termine par l'identifiant de l'événement. Allons dans notre fichier **EventService** et ajoutons cet appel API maintenant.

src/services/EventService.js

```
import axios from 'axios'

const apiClient = axios.create({
  baseURL: 'https://my-json-server.typicode.com/Code-Pop/Real-World_Vue-3',
  withCredentials: false,
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json'
  }
})

export default {
  getEvents() {
    return apiClient.get('/events')
  },
  //Added new call
  getEvent(id) {
    return apiClient.get('/events/' + id)
  }
}
```

L'appel `getEvent` est très similaire à l'appel `getEvents` de la dernière leçon. Cependant, il prend un `id` comme argument, qui est ajouté à la fin de l'url à laquelle nous faisons une demande d'obtention.

Maintenant que l'appel `getEvent` est prêt à l'emploi, utilisons-le dans notre nouveau composant **EventDetailsView**.

📁 **src/views/EventDetailsView.vue**

```
import { ref, onMounted } from 'vue'
import EventService from '@/services/EventService.js'

const event = ref(null)
const id = ref(123)

onMounted(() => {
  EventService.getEvent(id.value)
    .then((response) => {
      event.value = response.data
    })
    .catch((error) => {
      console.log(error)
    })
})
```

Quelques points à noter ici :

Nous appelons `getEvent` à partir de l'**EventService**, que nous avons maintenant importé dans le composant.

Nous transmettons `id.value` - Cet `id` n'est actuellement qu'une valeur de données codée en dur. (Nous rendrons cette valeur dynamique dans la deuxième partie de cette leçon. Ce n'est pas notre solution ultime).

Nous définissons notre `événement` local ref égal à la réponse de notre requête `getEvent`.

Maintenant que ce composant demande l'affichage d'un seul événement, nous pouvons l'ajouter à nos routes.

6.1.3 Ajouter EventDetailsView en tant que route

Nous allons aller dans notre fichier routeur, importer **EventDetailsView**, et l'ajouter à notre tableau de `routes` :

📁 **src/router/index.js**

```
...
import EventDetailsView from '../views/EventDetailsView.vue'
import AboutView from '../views/AboutView.vue'

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    ...
    {
      path: '/event/123',
      name: 'event-details',
      component: EventDetailsView,
    },
    ...
  ],
})
```

Pour l'instant, nous nous contenterons de coder en dur le chemin : `"/event/123"`. Par la suite, la partie finale (`123`) sera dynamique et mise à jour avec l'id~~entifiant~~ de l'événement en cours d'affichage.

Maintenant que nous avons cette nouvelle route, nous devons pouvoir y accéder. Encore une fois, nous voulons accéder à cette route chaque fois que nous cliquons sur l'une des **EventCards** ~~cartes d'événement~~ de notre page d'accueil.

6.1.4 Rendre EventCard cliquable avec un RouterLink

En ce qui concerne le composant **EventCard**, enveloppons le code de notre modèle dans un `RouterLink`.

src/components/EventCard.vue

```
<template>
  <RouterLink to="event/123">
    <div class="event-card">
      <h2>{{ event.title }}</h2>
      <span>@{{ event.time }} on {{ event.date }}</span>
    </div>
  </RouterLink>
</template>
```

Désormais, lorsque nous cliquerons sur l'une de nos cartes d'événement, nous serons dirigés vers le nouveau chemin `event/123`.

Si nous vérifions cela dans le navigateur, nous verrons une erreur dans la console si nous cliquons sur un événement :

Ce qui se passe ici, c'est que **EventDetailsView** essaie d'afficher les détails de l'événement avant d'avoir reçu l'événement en retour de l'appel API. Nous devons dire à notre composant d'attendre de recevoir l'événement avant d'essayer d'en afficher les détails. Heureusement, il s'agit d'une solution très simple.

Dans **EventsDetailsView.vue**, ajoutez un `v-if` sur la div :

src/views/EventDetailsView.vue

```
<template>
  <div v-if="event">
    <h1>{{ event.title }}</h1>
    <p>{{ event.time }} on {{ event.date }} @ {{ event.location }}</p>
    <p>{{ event.description }}</p>
  </div>
</template>
```

En ajoutant un simple `v-if="event"` à notre div, nous pouvons nous assurer qu'elle ne s'affiche que lorsque l'`événement` existe dans nos données.

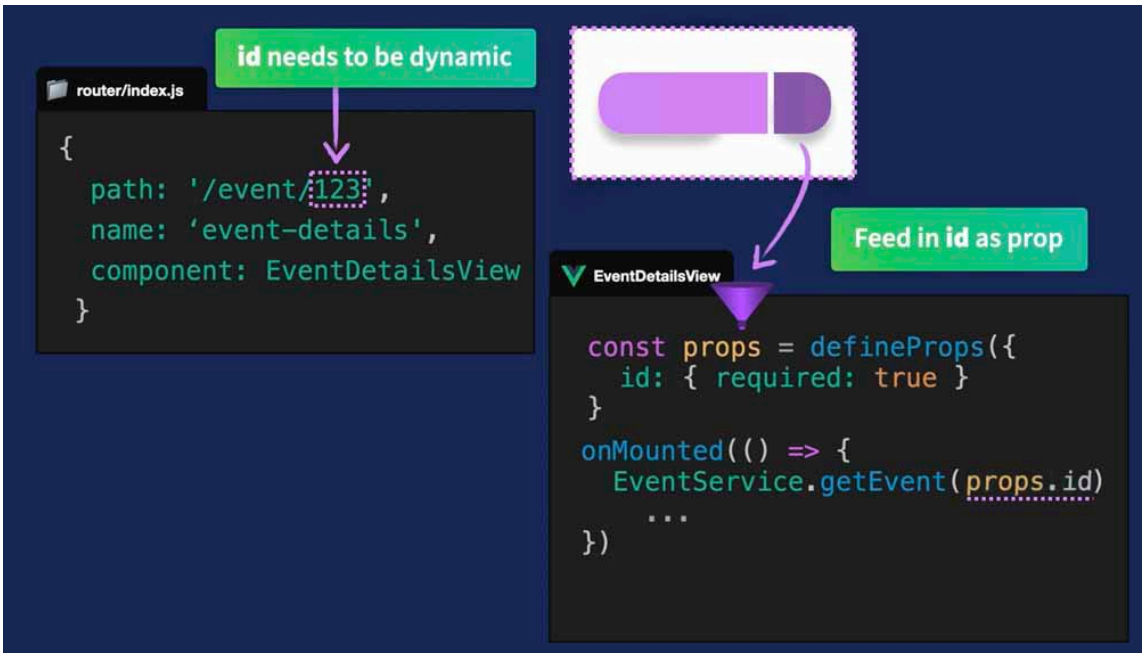
Si nous vérifions à nouveau dans le navigateur, nous verrons que cela fonctionne jusqu'à présent... Lorsque nous cliquons sur la ~~**carte d'événement**~~ **EventCard** de la Journée de l'adoption des chats, nous accédons à une vue qui affiche les détails de cet événement.

Cependant, si nous cliquons sur n'importe quelle autre **EventCard**, nous obtenons toujours les mêmes détails sur la Journée de l'adoption des chats, et l'id~~entifiant~~ à la fin de notre URL est le même : 123. C'est normal, puisque nous avons codé en dur l'id~~entifiant~~ que nous passons dans l'appel `getEvent`, et dans le chemin de la route EventDetails.

Ceci nous amène à la fin de la partie 1 et au début de la partie 2, où nous rendons ce comportement de routage dynamique afin de pouvoir accéder aux détails de n'importe quelle **EventCard** sur laquelle nous cliquons.

6.2 Partie 2 : Rendre le comportement dynamique

Pour rendre notre comportement de routage dynamique, nous devons remplacer l'id~~entifiant~~ codé en dur dans notre chemin (`/123`) par un **segment dynamique**. Il s'agit en fait d'un *paramètre* variable pour le chemin d'accès à l'URL, qui est mis à jour avec l'id~~entifiant~~ de l'événement actuellement affiché sur cette route.



Nous voudrons ensuite être en mesure d'introduire ce segment dynamique dans le composant **EventDetailsView** en tant que paramètre à utiliser lors de l'appel `getEvent`.

6.2.1 Ajouter un segment dynamique à la route EventDetailsView

Commençons par ajouter un segment dynamique au chemin de la route **EventDetailsView**.

src/router/index.js

```
{
  path: '/event/:id',
  name: 'event-details',
  props: true,
  component: EventDetailsView,
},
```

Remarquez que la syntaxe d'un segment dynamique commence par deux points `:` et est suivie par le nom que vous souhaitez donner au segment. Dans ce cas, il s'agit de `:id` puisqu'il est remplacé par l'id~~entifiant~~ de notre événement. Dans un autre cas d'utilisation, cela pourrait être `:username` ou `:orderNumber`.

Nous avons également ajouté `props : true` ici pour donner au composant **EventDetailsView** l'accès à ce paramètre de segment dynamique en tant que prop.

Puisque nous avons mis à jour le chemin dans cette route, le chemin dans l'attribut `to` de l'attribut de notre **EventCard** doit maintenant être mis à jour. Rappelez-vous qu'il est actuellement codé en dur à `to="event/123"`

📂 **src/components/EventCard.vue**

```
<template>
  <RouterLink to="event/123">
    <div class="event-card">
      <h2>{{ event.title }}</h2>
      <span>@{{ event.time }} on {{ event.date }}</span>
    </div>
  </RouterLink>
</template>
```

Une solution plus simple consisterait à utiliser une **route nommée**, où nous lions :to à un objet qui spécifie la route vers laquelle ce lien renvoie.

💡 Tangente pertinente : Nous avons également rendu notre application un peu plus évolutive. Dans une application plus grande avec des RouterLinks partout, il devient inutilement difficile de maintenir les chemins dans chaque RouterLink chaque fois qu'ils doivent changer. D'un autre côté, si vos RouterLinks utilisent des **routes nommées**, et que le chemin de votre route doit changer, vous pouvez simplement le changer une fois dans le fichier du routeur, et aucun de vos RouterLinks n'a besoin d'être mis à jour puisqu'ils ne dépendent pas du chemin lui-même.

6.2.2 Ajouter l'id~~entifiant~~ de l'événement aux paramètres du routeur

À ce stade, vous vous demandez peut-être comment indiquer à notre segment dynamique :id la valeur par laquelle il doit être remplacé. Nous pouvons le faire en ajoutant la propriété params à notre objet dans l'attribut to :

📂 **src/components/EventCard.vue**

```
<script setup>
defineProps({
  event: {
    type: Object,
    required: true,
  },
})
</script>

<template>
  <RouterLink :to="{ name: 'event-details', params: { id: event.id } }">
    <div class="event-card">
      <h2>{{ event.title }}</h2>
      <span>@{{ event.time }} on {{ event.date }}</span>
    </div>
  </RouterLink>
</template>
```

Rappelez-vous de ce qui a été dit il y a quelques leçons, ce composant a l'événement en tant que props, donc nous pouvons récupérer event.id à partir de lui et mettre l'id de params égal à lui.

```
<RouterLink :to="{ name: 'event-details', params: { id: event.id } }">
```

Désormais, lorsque nous cliquons sur ce RouterLink, nous sommes dirigés vers **EventDetailsView** et le chemin de la route est complété par l'id~~entifiant~~ de l'événement.

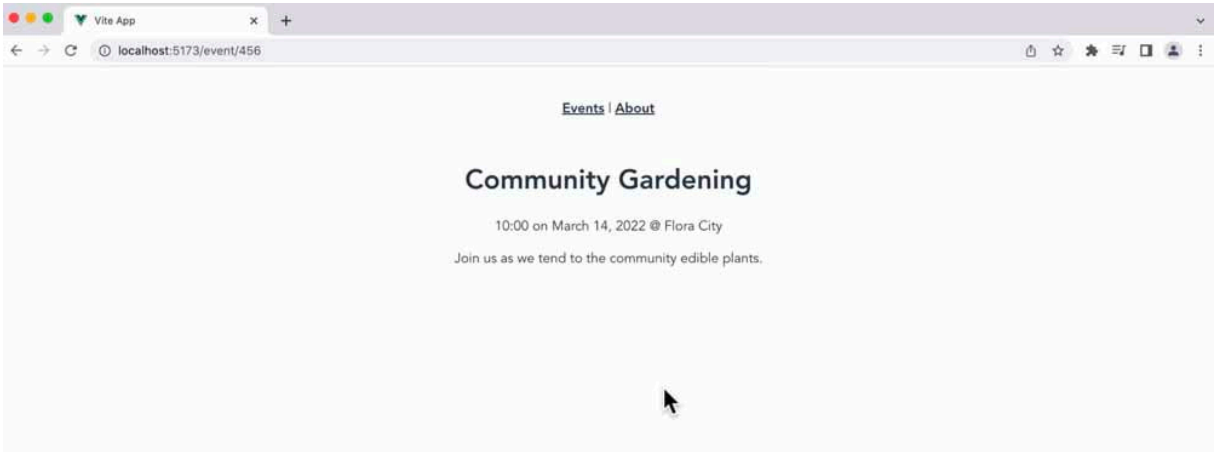
Nous pouvons enfin introduire ce paramètre id dans le composant **EventDetailsView** sous la forme d'une propriété.

src/views/EventDetailsView.vue

```
<script setup>
import { ref, onMounted } from 'vue'
import EventService from '@/services/EventService.js'

const props = defineProps({
  id: {
    required: true,
  },
})

const event = ref(null)

onMounted(() => {
  EventService.getEvent(props.id)
    .then((response) => {
      event.value = response.data
    })
    .catch((error) => {
      console.log(error)
    })
})
</script>
```

Maintenant, lorsque nous disons getEvent(this.id), nous nous référons à l'id props nouvellement ajouté. Lorsque **EventDetailsView** est routé vers et donc *monté*, il fait maintenant une demande pour l'événement avec l'id qui est trouvé dans le paramètre dynamique du chemin de la route.

Si nous vérifions cela dans le navigateur, nous sommes en mesure de cliquer sur une **EventCard** et d'afficher les détails appropriés pour cet événement.



6.2.3 Nettoyage de notre code

Avec cela, nous avons terminé notre comportement de routage dynamique. Ouf ! C'était beaucoup d'étapes. Maintenant, je veux juste nettoyer quelques points avant de terminer.

Tout d'abord, nos **EventCards** ne sont plus aussi belles maintenant qu'elles sont entourées d'un RouterLink

Ajoutons une classe event-link au <RouterLink> pour lui donner un aspect plus agréable :

src/components/EventCard.vue

```
<template>
  <RouterLink
    class="event-link"
    :to="{ name: 'event-details', params: { id: event.id } }"
  >
    <div class="event-card">
      <h2>{{ event.title }}</h2>
      <span>@{{ event.time }} on {{ event.date }}</span>
    </div>
  </RouterLink>
</template>

<style scoped>
...
.event-link {
  color: #2c3e50;
  text-decoration: none;
}
</style>
```

Par souci de cohérence, nous pouvons également mettre à jour le fichier **App.vue** pour utiliser des routes nommées au lieu de chemins codés en dur.

src/App.vue

```
<div class="wrapper">
  <nav>
    <RouterLink :to="{ name: 'event-list' }">Events</RouterLink> |
    <RouterLink :to="{ name: 'about' }">About</RouterLink>
  </nav>
</div>
```

Encore une fois, cela permet d'intégrer l'évolutivité à la maintenance des itinéraires de notre application.

6.3 Prochaines étapes

Pour continuer à apprendre des concepts tels que les paramètres de route et d'autres sujets relatifs à Vue Router, vous pouvez consulter l'ensemble de notre cours Touring Vue Router.

Dans la prochaine leçon, nous allons apprendre à prendre notre application et à la déployer en production, en utilisant Render.