Touring Vue Router

Titre
Réaliser une application web étape par étape afin de comprendre les notions de base d'une application réalisée avec **vue.js**.

Compétence
Implémenter, au moyen de la technologie définie, le front-end d'une application Web interactive permettant la gestion de données.

Table des matières

1. Introduction

Dans le cours Touring Vue Router, nous explorerons la plupart des fonctionnalités de la bibliothèque **Vue Router** qui nous permettent de créer une navigation avancée via nos applications à page unique (SPA) à l'aide de Vue.

La majeure partie de cette configuration sera effectuée dans notre fichier ***/router/index.js***, où nous définissons les routes de notre application. Plus précisément, quels chemins d'URL renvoient à quels composants et où ils sont définis sur notre écran.

Au cours de ce document, nous développerons l'application d'événements que vous avez commencé à créer précédemment dans le cours **Real World Vue 3**. Nous intégrerons la pagination, une gestion appropriée des erreurs lorsqu'une page n'existe pas ou que le réseau est en panne, une barre de progression pour compenser lorsqu'une page est lente à charger (probablement à cause d'un appel API lent) et un message flash pour donner à nos utilisateurs un message qui apparaît en haut de n'importe quelle page.

Nous donnerons également un aperçu d'une grande partie de la syntaxe de **Vue Router** dont vous aurez besoin pour créer de grandes applications Vue. Nous allons faire un voyage à travers l'univers de **Vue.js** pour explorer la technologie et construire une base solide de nouvelles compétences.

Tout au long de ce cours, nous apprendrons les bases de Vue.js et nous allons construire une application pour mettre ces concepts en pratique.

2. Paramètres d'URL de réception

Dans cette leçon, nous donnerons un aperçu de toutes les différentes manières dont nous pouvons recevoir et analyser les données de l'URL dans nos composants avec **Vue Router**. Cela garantira que nous disposons des outils dont nous avons besoin pour créer la pagination dans notre prochaine leçon.

2.1 Problème: Comment lire les paramètres de requête à partir de l'URL?

Par exemple, souvent, lorsque nous écrivons une pagination, nous pouvons avoir une URL qui ressemble à ceci : ***http://example.com/events?page=4***

Comment pouvons-nous accéder à l'intérieur de notre composant **page** ?

2.2 Solution: $route.query.page

Dans notre composant, pour lire le numéro de la page, tout ce que nous devons faire dans notre template est d'écrire :

```
<h1>You are on page {{ $route.query.page }}</h1>
```

Cela pourrait ressembler à ceci :

Pour y accéder depuis le code du composant, nous devrons ajouter cela :

```
import { computed } from "vue";
import { useRoute } from 'vue-router'
const route = useRoute()

const page = computed(() => parseInt(route.query.page) || 1)
```

2.3 Problème: Et si nous voulions que la page fasse partie de l'URL?

Dans certains cas, dans le développement Web, vous souhaiterez peut-être que le numéro de page fasse réellement partie de l'URL, plutôt que dans les paramètres de requête (qui viennent après un point d'interrogation).

2.4 Solution: paramètre de route

Dans le précédent cours **Real World Vue**, vous avez déjà vu la solution. Pour résoudre ce problème, nous aurions probablement une route qui ressemblerait à cela :

```
const routes = [
  ...
  { path: '/events/:page', component: Events },
]
```

Ensuite, dans notre composant d'événements, nous pourrions y accéder dans le template en tant que tel :

```
<h1>You are on page {{ $route.params.page }}</h1>
```

Notez que dans ce cas, nous utilisons **$route.params** à la place de **$route.query** ce que nous avons vu ci-dessus.

2.5 Bonus: passer des paramètres comme Props

Si nous voulons rendre notre composant plus réutilisable et testable, nous pouvons le découpler de la route en disant à notre routeur de transmettre notre paramètre **page** en tant que Props de composant. Pour ce faire, à l'intérieur de notre routeur nous écririons :

```
const routes = [
  ...
  { path: '/events/:page', component: Events, props: true },
]
```

Ensuite, à l'intérieur de notre composant, nous aurions :

```
<script setup>

defineProps(['page'])

</script>
<template>
  <h1>You are on page {{ page }}</h1>
</template>
```

Notez que nous déclarons page comme Props et que nous le rendons sur la page.

2.6 Problème: configuration au niveau de la route

Parfois, nous avons un composant que nous souhaitons pouvoir configurer au niveau de la route. Par exemple, s'il y a des informations supplémentaires que nous souhaitons afficher ou non.

2.7 Solution: mode objet des Props

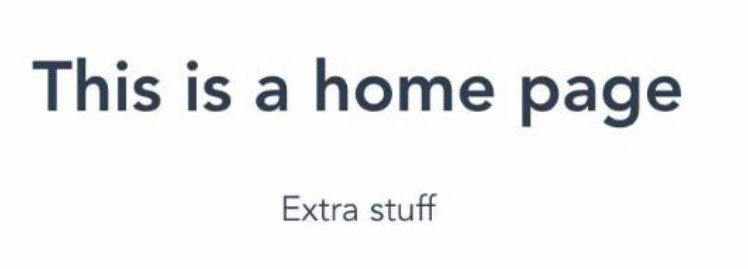
Lorsque nous voulons envoyer la configuration dans un composant depuis le routeur, cela peut ressembler à ceci :

```
const routes = [
  {
    path: "/",
    name: "Home",
    component: Home,
    props: { showExtra: true },
  },
```

Notez l'objet props statique avec showExtra. Nous pouvons ensuite le recevoir comme Props dans notre composant et en faire quelque chose :

```
<template>
  <div class="home">
    <h1>This is a home page</h1>
    <div v-if="showExtra">Extra stuff</div>
  </div>
</template>
<script setup>

defineProps(['showExtra'])

</script>
```

Et maintenant, lorsque je consulte ma page d'accueil, je vois :



2.8 Problème: Comment transformer les paramètres d'une requête ?

Parfois, vous pouvez rencontrer une situation dans laquelle les données envoyées dans vos paramètres de requête doivent être transformées avant d'atteindre votre composant. Par exemple, vous souhaiterez peut-être convertir des paramètres en d'autres types, les renommer ou combiner des valeurs.

Dans notre cas, supposons que notre URL envoie e=true en tant que paramètre de requête, alors que notre composant souhaite recevoir showExtra=true en utilisant le même composant dans l'exemple ci-dessus. Comment pourrions-nous réaliser cette transformation ?

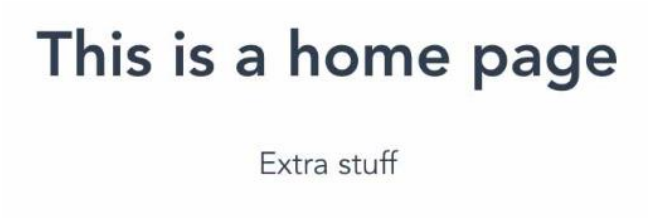
2.9 Solution: mode fonction des Props

Dans notre route pour résoudre ce problème, nous écririons :

```
const routes = [
  {
    path: "/",
    name: "Home",
    component: Home,
    props: (route) => ({ showExtra: route.query.e }),
  },
```

Notez que nous envoyons une fonction anonyme qui reçoit la route comme argument, puis extrait le paramètre de requête appelé e et le mappe au Props showExtra.

En utilisant le même code de composant que ci-dessus, nous obtenons le même résultat :



La fonction anonyme ci-dessus pourrait également s'écrire comme :

```
props: route => {
  return { showExtra: route.query.e }
}
```

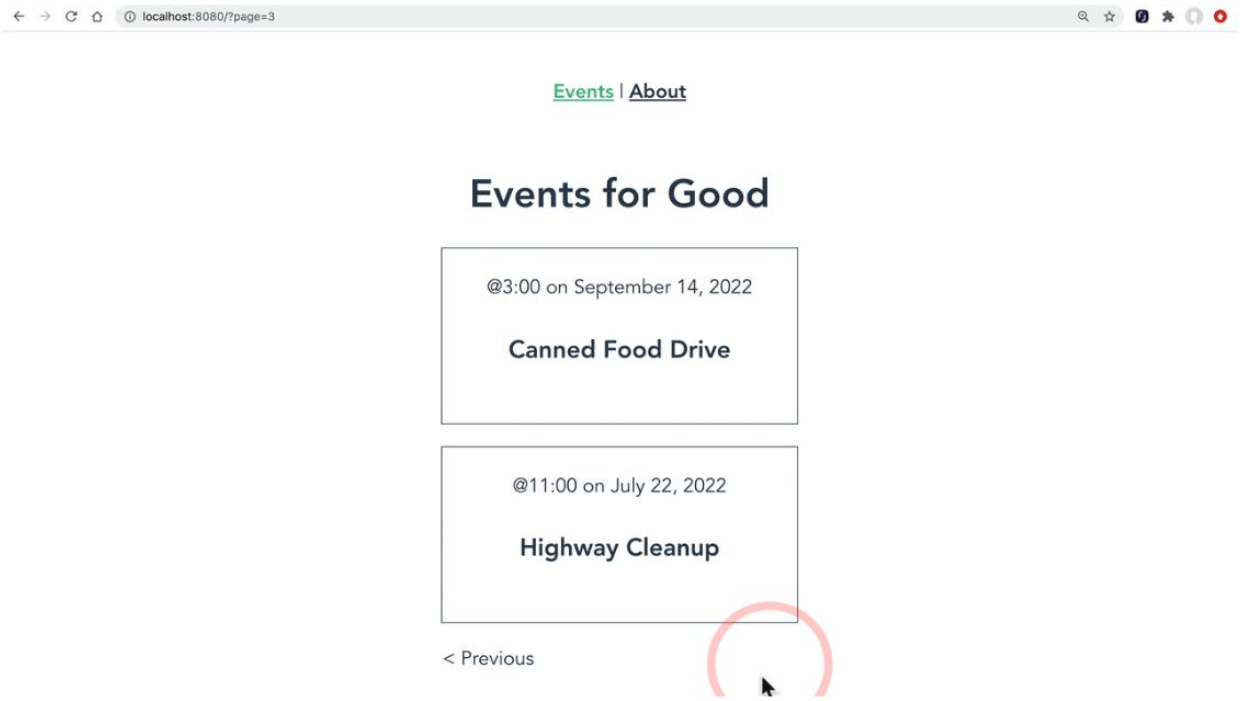
C'est un peu plus verbeux, mais je voulais vous montrer ceci pour vous rappeler que vous pouvez placer des transformations ou des validations complexes à l'intérieur de cette fonction.

Dans la section suivante, nous utiliserons certaines de ces techniques pour créer une pagination.

3. Construire la pagination

La pagination est une fonctionnalité très courante qui devra être ajoutée à n'importe quelle application. Dans cette leçon, nous allons implémenter la

pagination de base. Nous partirons du code écrit dans notre cours **Real World Vue 3**, qui effectue un appel API pour récupérer une liste d'événements. À la fin de la leçon, notre page ressemblera et fonctionnera comme ceci :



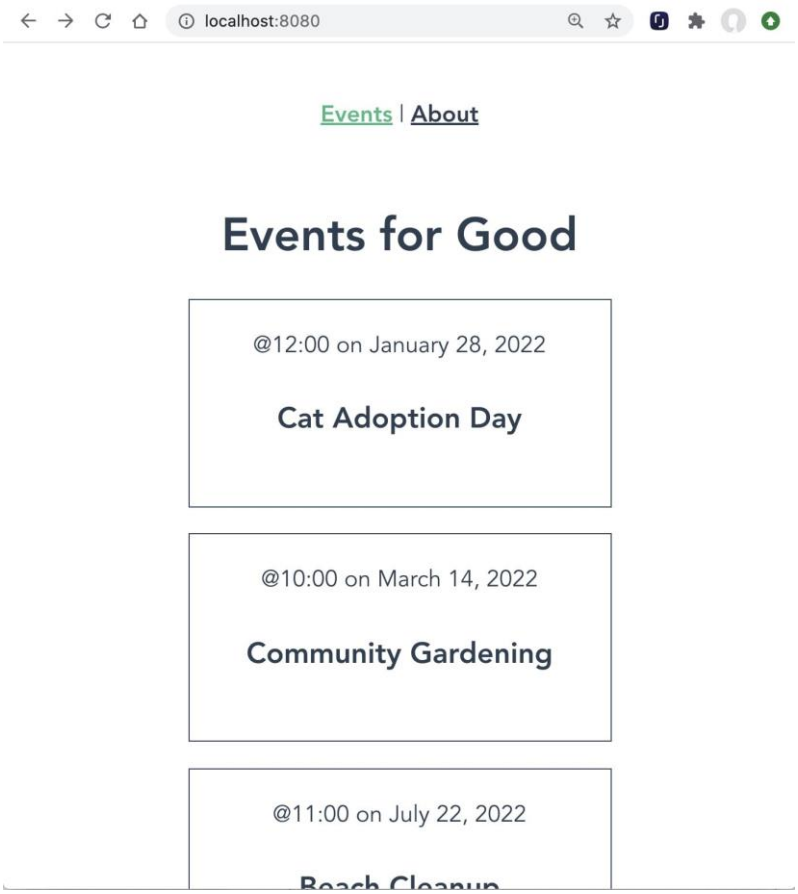
3.1 A vous de coder dans ce cours

Dans les chapitres suivants, nous intégrerons des fonctionnalités à notre application et je vous encourage à coder en même temps. Je vais même parfois vous donner des idées sur des choses à essayer de construire au-dessus de l'application. Vous pouvez soit consulter le référentiel, soit mieux encore, simplement télécharger le code de la branche L3-start (sans contrôle de version) et l'archiver dans votre propre dépôt git.

Au fur et à mesure que nous développons des fonctionnalités à chaque niveau, vous pouvez consulter le code de début de chaque niveau en regardant la branche L#-start et le code de fin dans la branche L#-end(# étant le numéro de la leçon). Ainsi, pour cette leçon, vous trouverez le code de début dans la branche L3-start et le code de fin dans la branche L3-end.

3.2 L'application d'événements

Notre application répertorie actuellement les événements d'une API distante et ressemble à ceci :



Nous utiliserons les étapes suivantes pour ajouter une pagination à cet exemple :

1. Modifier l'appel d'API **EventService** pour prendre "**perPage**" et "**page**"
2. Analyser et définir la page actuelle du routeur en utilisant le mode Fonction
3. Modifier la façon dont **EventService** est appelé à partir de **EventList.vue**
4. Ajouter des liens de pagination au template **EventList**
5. Afficher le lien Page suivante uniquement lorsqu'il existe une page suivante
6. Améliorer le style de pagination

3.3 Etape 1: Modification de l'appel API EventService

Heureusement pour nous, le service JSON Server que nous utilisons intègre cette fonctionnalité pour pouvoir paginer. Il existe deux paramètres de requête que nous pouvons envoyer au service :

- **_limit** - Combien d'articles à retourner par page.
- **_page** - Sur quelle page nous sommes.

Notre appel getEvents() ressemble actuellement à ceci :

/src/services/EventService.js

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

Nous allons ajouter deux paramètres à l'appel de fonction et les envoyer dans l'URL :

```
getEvents(perPage, page) {
  return apiClient.get('/events?_limit=' + perPage + '&_page=' + page)
},
```

C'est tout.

3.4 Etape 2. Analyser et définir la page actuelle du routeur

Lorsque nous serons à la page deux, notre URL ressemblera à http://localhost:8080/?page=2. Ainsi, nous analyserons ces données (si elles existent) et les enverrons comme Props depuis le routeur, comme nous l'avons appris dans la leçon précédente :

```
{
  path: '/',
  name: 'EventList',
  component: EventList,
  props: route => ({ page: parseInt(route.query.page) || 1 })
},
```

Comme vous pouvez le voir, si le paramètre de requête page existe, nous l'analysons à partir d'une chaîne en un entier, sinon `|| 1` nous le définissons sur la première page. Nous devrons accepter page comme Props dans notre EventList.vue, que nous modifierons ensuite :

3.5 Etape 3. Modifier le EventList.vue

Le composant sur lequel nous paginons est EventList.vue, qui ressemble actuellement à ceci :

/src/views/EventList.vue

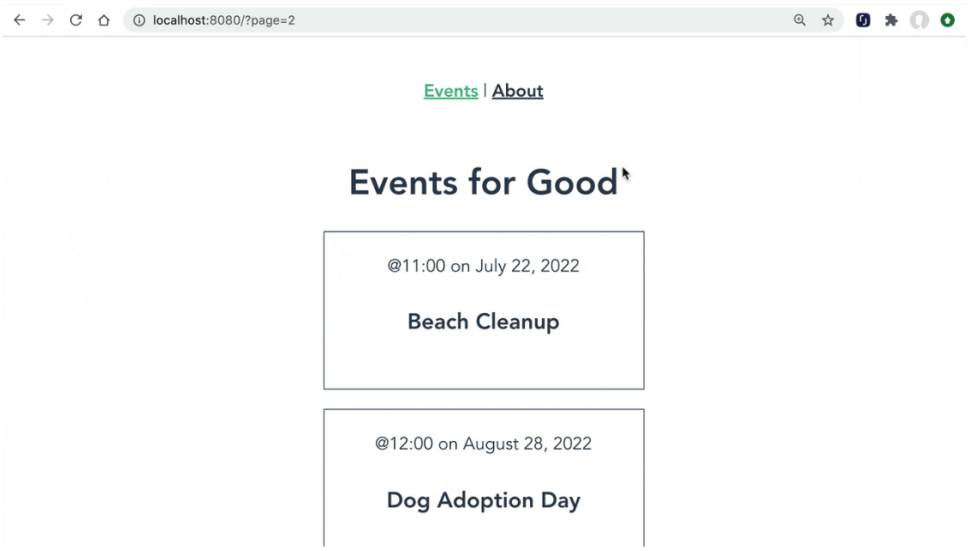
```
...
import { ref, onMounted } from 'vue'
import EventCard from '@/components/EventCard.vue'
import EventService from '@/services/EventService.js'

const events = ref(null)

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

Modifions ceci pour recevoir page comme Props pour l'instant et envoyons-le dans l'appel getEvents. Pour l'instant, nous allons coder en dur 2 comme nombre d'événements à renvoyer par page.

/src/views/EventList.vue

```
...
import { onMounted, ref, computed } from "vue";

const props = defineProps(['page'])

const events = ref("");

const page = computed(() => props.page)

onMounted(() => {
  EventService.getEvents(2, page.value)
    .then((response) => {
      events.value = response.data;
    })
    .catch((error) => {
      console.log(error);
    });
});
</script>
...
```

Si vous suivez, c'est à ce stade que vous souhaiterez exécuter npm install, puis npm run dev rendre le serveur opérationnel. Voici ce que nous voyons :



Ça a l'air génial, différentes pages chargent différents événements !

3.6 Etape 4. Ajout de liens de pagination

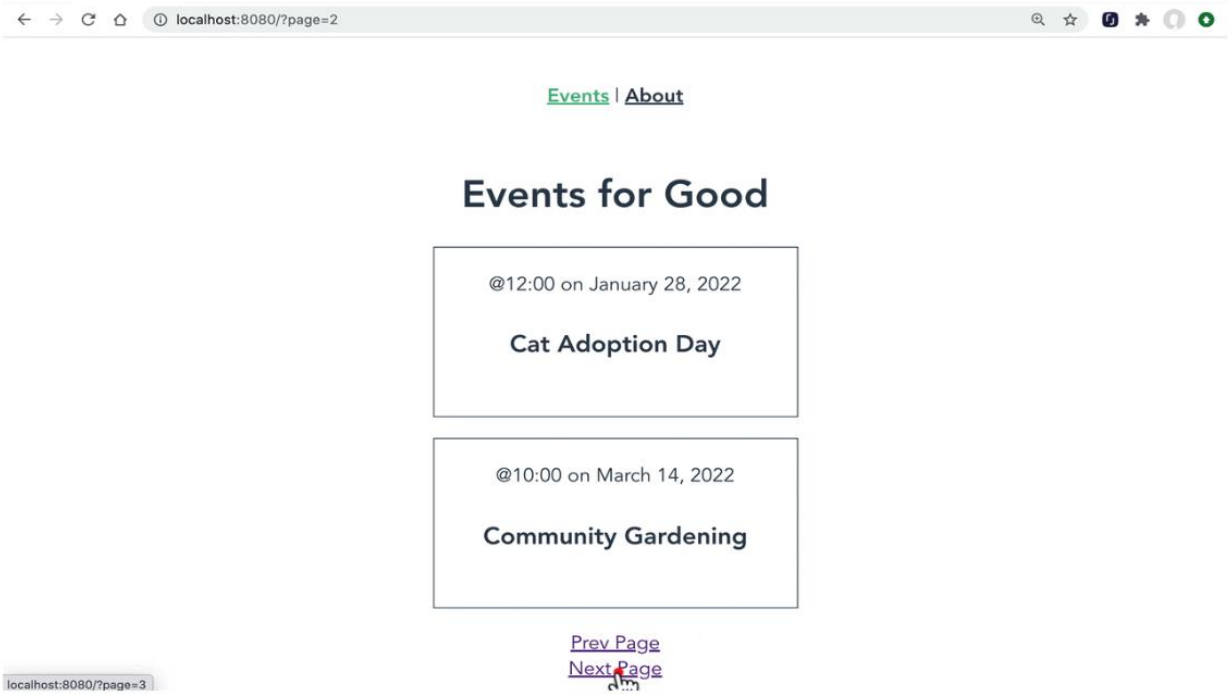
Ensuite, nous devons ajouter la pagination au bas de notre liste.

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

    <router-link
      :to="{ name: 'EventList', query: { page: page - 1 } }"
      rel="prev"
      v-if="page != 1"
      >Prev Page</router-link
    >

    <router-link
      :to="{ name: 'EventList', query: { page: page + 1 } }"
      rel="next"
      >Next Page</router-link
    >
  </div>
</template>
    ...
```

Quelques points à remarquer ici :

- Les nouvelles directives router-link, dans lesquelles j'utilise query: pour spécifier la page précédente et la page suivante en soustrayant et en ajoutant 1.
- rel="prev" et rel="next" n'ont rien à voir avec Vue, ce sont simplement de bonnes pratiques de référencement et de normes pour les pages Web.
- La page précédente que je n'affiche que si je ne suis pas sur la première page en utilisant un fichier v-if.

Jetons un coup d'œil à cela dans notre navigateur



Comme vous pouvez le voir sur ma première page, les données sont correctement paginées. Cependant, lorsque je clique pour passer à la page suivante, rien ne se passe.

3.7 Problème: la liste des événements n'est pas mise à jour

Ce qui se passe ici, c'est que notre routeur voit que nous chargeons la même route nommée EventList, il n'a donc pas besoin de recharger le composant (ou de réexécuter les hooks de cycle de vie où notre appel API est stocké). C'est comme cliquer deux fois sur un lien de navigation. Lorsque quelqu'un clique deux fois sur un lien de navigation et qu'il est déjà sur cette page, voulons-nous qu'il recharge le composant ? Non, c'est ce qui se passe. `onMounted()` n'est pas appelé à nouveau lorsque nous passons à la deuxième page, car il ne recharge pas le composant.

Inévitablement, vous rencontrerez cela en tant que développeur Vue lorsque vous souhaitez recharger un composant avec une modification des paramètres de requête.

3.8 2 solutions

Il existe deux façons de résoudre ce problème :

1. Dire à notre routeur de recharger les composants dans notre router-view lorsque l'URL complète change, y compris les paramètres de requête. Nous pouvons le faire en disant à notre App.vue router-view d'utiliser $route.fullPath pour valeur de l'attribut key.

```
<router-view :key="$route.fullPath" />
```

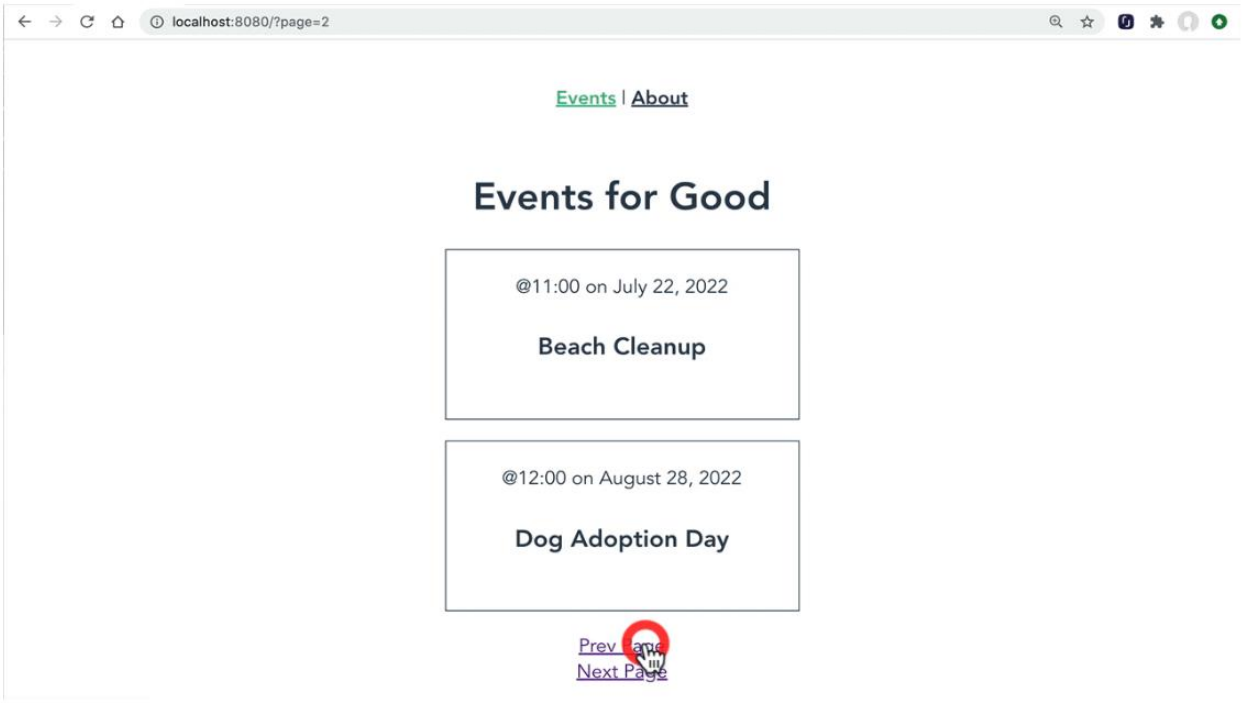
   Ce n'est pas la solution que l'on va retenir donc voyons le point 2.

2. Surveiller les propriétés réactives pour les modifications (qui incluent les paramètres de requête). Nous pouvons le faire en encapsulant simplement notre appel API dans une watchEffect méthode. Il s'agit d'une nouvelle méthode dans Vue 3 qui surveille les modifications de propriétés réactives et réexécute le code approprié si quelque chose change.

```
import { onMounted, ref, computed, watchEffect } from "vue";

const props = defineProps(['page'])

const events = ref("");

const page = computed(() => props.page)

onMounted(() => {
  watchEffect(() => {
    events.value = null
    EventService.getEvents(2, page.value)
      .then((response) => {
        events.value = response.data;
      })
      .catch((error) => {
        console.log(error);
      });
  })
});
  ...
```

Vous remarquerez que juste en dessous de watchEffect() je réinitialise le fichier events.value = null. C'est ainsi que lorsque nous chargeons une autre page, la

liste actuelle des événements est supprimée afin que l'utilisateur sache qu'elle est en cours de chargement. Nous pourrions également avoir une roulette animée si nous le voulions.



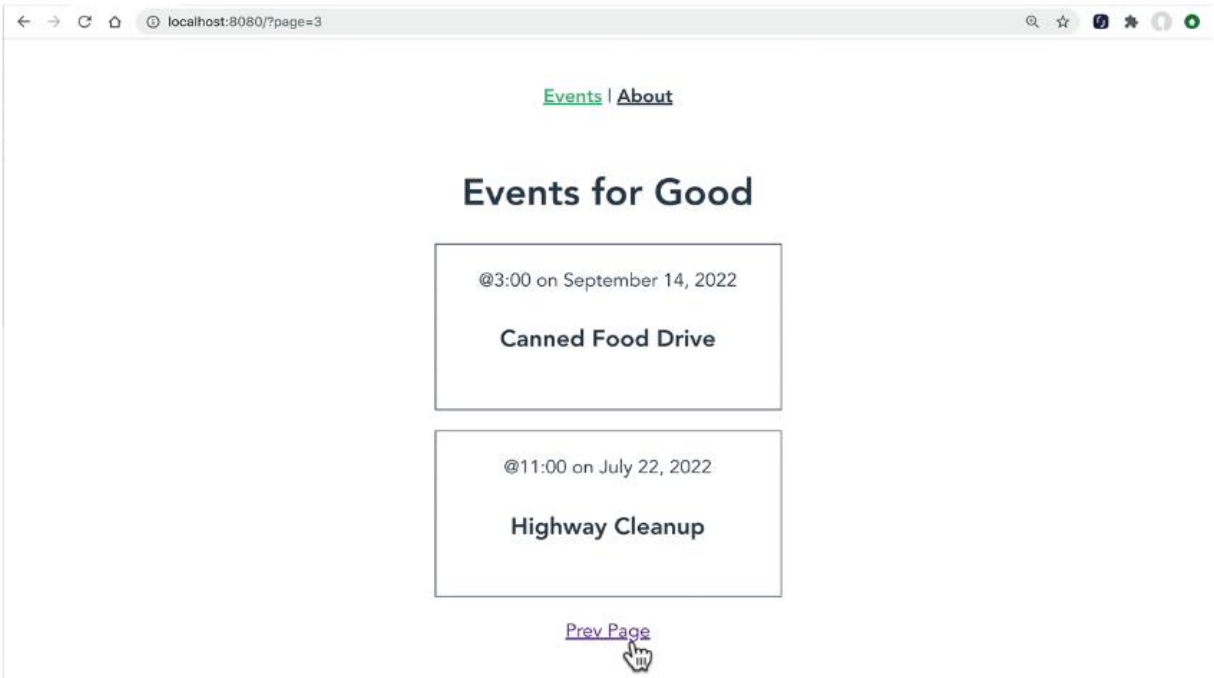
Maintenant, tout fonctionne, sauf que sur la dernière page, on ne devrait pas afficher le lien Next Page.

3.9 Etape 5. Vérification de la dernière page

Une façon de savoir si nous sommes sur la dernière page est de connaître le nombre total d'événements, afin de pouvoir calculer le nombre total de pages. Heureusement, JSON Server en tient compte, et sur les en-têtes qu'ils renvoient depuis l'appel à l'API, il y a un x-total-count en-tête qui nous envoie déjà le total des événements. Super ! Nous devons simplement extraire cela de notre appel API et créer une nouvelle propriété calculée qui calcule s'il y a isNextPage. Si c'est vrai, nous afficherons « Page suivante ».

```vue
<template>
  <div class="events">
    ...

    <router-link
      :to="{ name: 'EventList', query: { page: page + 1 } }"
      rel="next"
      v-if="hasNextPage"
      >Next Page</router-link
    >
  </div>
</template>

<script>
  ...
  const totalEvents = ref(0) // Store the total events
  ...

  const hasNextPage = computed(() => {
    // Calculate totalPages, based on 2 per page
    const totalPages = Math.ceil(totalEvents.value / 2)

    // If current page is less than total pages, return true
    return page.value < totalPages
  })

  onMounted(() => {
    watchEffect(() => {
      events.value = null
      EventService.getEvents(2, page.value)
        .then(response => {
          events.value = response.data
          // our response has total stored in the header.
          totalEvents.value = response.headers['x-total-count']
        })
        .catch(error => {
          console.log(error)
        })
    })
  })
  </script>
```

Assurez-vous de consulter les commentaires que j'ai laissés dans le code ci-dessus. Maintenant ça marche, il n'y a plus le lien Next Page sur la dernière page :



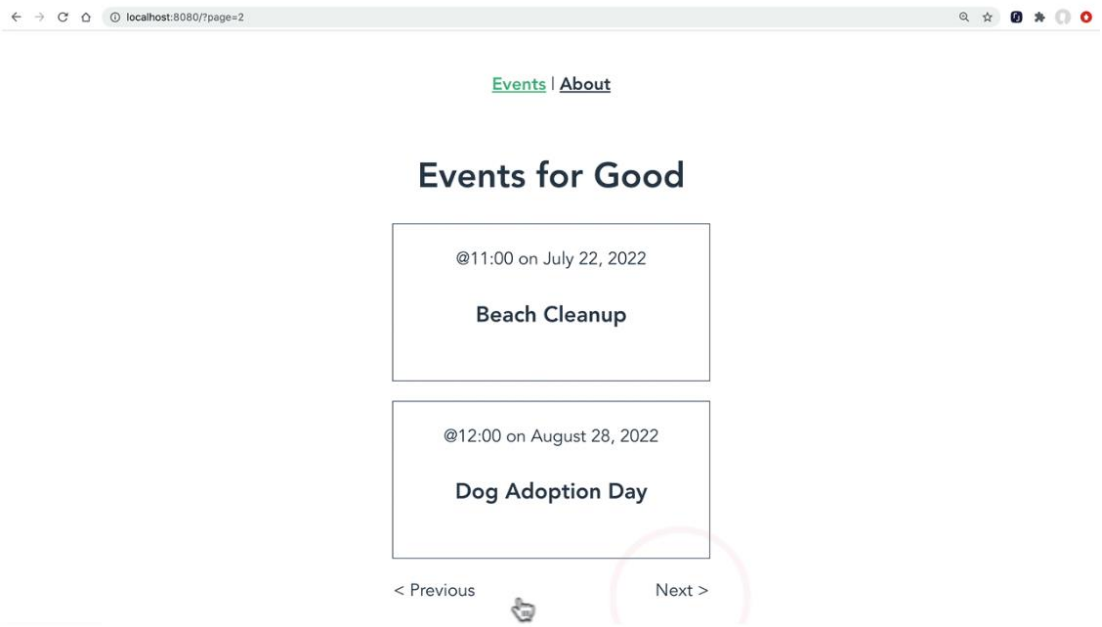
3.10 Etape 6. Améliorer le style de pagination

Nous avons fini de créer notre code Vue, mais ce serait bien si nos liens étaient un peu plus jolis. Allons-y et embellissons-les un peu en utilisant flexbox.

Ci-dessous, j'ai ajouté un div pour nos liens de pagination, puis j'ai donné à chacun un identifiant, modifié le texte du lien et ajouté du style plus bas dans ce composant Single File Vue.

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

    <div class="pagination">
      <router-link
        id="page-prev"
        :to="{ name: 'EventList', query: { page: page - 1 } }"
        rel="prev"
        v-if="page != 1"
        >&#60; Previous</router-link
      >

      <router-link
        id="page-next"
        :to="{ name: 'EventList', query: { page: page + 1 } }"
        rel="next"
        v-if="isNextPage"
        >Next &#62;</router-link
      >
    </div>
  </div>
</template>

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

.pagination {
  display: flex;
  width: 290px;
}
.pagination a {
  flex: 1;
  text-decoration: none;
  color: #2c3e50;
}

#page-prev {
  text-align: left;
}

#page-next {
  text-align: right;
}
</style>
```

Désormais, nos liens sont plus agréables à regarder :



Dans la prochaine leçon, nous verrons comment imbriquer nos routes lorsque notre application devient de plus en plus complexe.

4. Routes imbriqués

Souvent, lors de la création d'applications Web, nous devons être capables d'effectuer plusieurs actions (Afficher, Modifier, Enregistrer) pour une seule ressource (dans notre cas un événement). Chaque URL fournit des informations différentes sur cette ressource :

- /event/2 - Détails de l'événement (informations)
- /event/2/register – Pour vous inscrire à l'événement
- /event/2/edit - Pour modifier l'événement

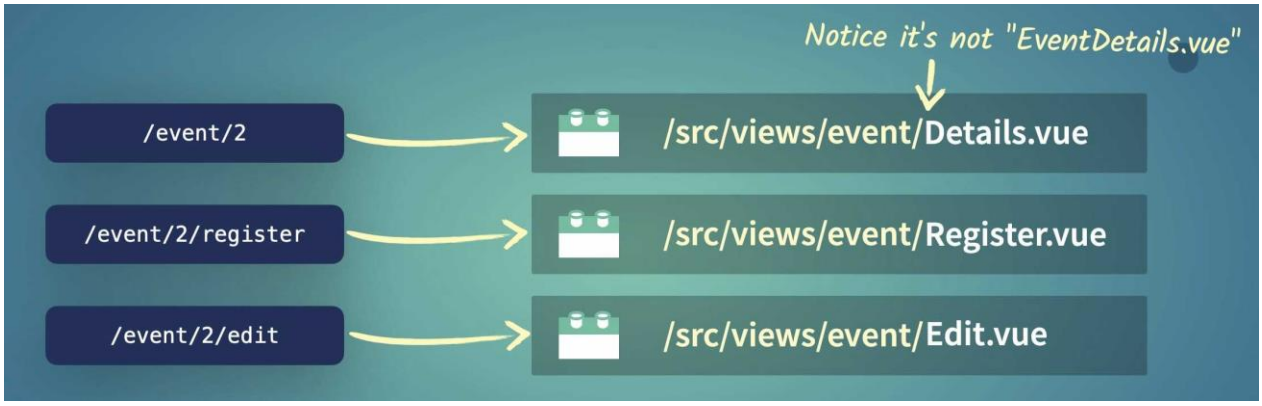
La mise en œuvre de cela entraîne quelques problèmes à résoudre.

4.1 Problème: Où place-t-on ces composants?

Il y a plusieurs solutions.

4.2 Solution: Dans leur propre dossier.

Une bonne pratique pour placer ces composants dans Vue serait de les placer dans leur propre dossier event, en les organisant efficacement par ressource :



Génial, cela résout un problème.

4.3 Problème: Comment créer et acheminer vers ces vues ?

Pour résoudre ce problème, je vais d'abord utiliser une solution simple, utilisant le routage de base. Il y aura du code en double dans cette solution, puis je vais vous montrer comment résoudre cela à l'aide des routes imbriquées de Vue Router.

4.4 Solution : Routage de base

Tout d'abord, nous devrons mapper ces composants dans le routeur. Notez que je les appelle EventDetails, EventRegister et EventEdit, afin de ne pas rencontrer accidentellement de conflits de noms à mesure que mon routeur grandit :

```
import { createRouter, createWebHistory } from 'vue-router'
import About from '@/views/About.vue'
import EventList from '@/views/EventList.vue'
import EventDetails from '@/views/event/Details.vue'
import EventRegister from '@/views/event/Register.vue'
import EventEdit from '@/views/event/Edit.vue'

const routes = [
  {
    path: '/',
    name: 'EventList',
    component: EventList
    props: route => ({ page: parseInt(route.query.page) || 1 })
  },
  {
    path: '/event/:id',
    name: 'EventDetails',
    props: true,
    component: EventDetails
  },
  {
    path: '/event/:id/register',
    name: 'EventRegister',
    props: true,
    component: EventRegister
  },
  {
    path: '/event/:id/edit',
    name: 'EventEdit',
    props: true,
    component: EventEdit
  },
  ...
```

Remarquez comment j'importe les composants, en utilisant une route dynamique et en spécifiant `props: true` d'envoyer la partie identifiant de l'URL en tant que Props dans les composants. Jetons un coup d'œil à l'une des vues.

/src/views/event/Details.vue

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

const { id } = defineProps(['id'])

const event = ref(null)
onMounted(() => {
  EventService.getEvent(id)
    .then(response => {
      event.value = response.data
    })
    .catch(error => {
      console.log(error)
    })
})
</script>
<template>
  <div v-if="event">
    <h1>{{ event.title }}</h1>
    <div id="nav">
      <router-link :to="{ name: 'EventDetails', params: { id } }"
        >Details</router-link
      >
      |
      <router-link :to="{ name: 'EventRegister', params: { id } }"
        >Register</router-link
      >
      |
      <router-link :to="{ name: 'EventEdit', params: { id } }"
        >Edit</router-link
      >
    </div>
    <p>{{ event.time }} on {{ event.date }} @ {{ event.location }}</p>
    <p>{{ event.description }}</p>
  </div>
</template>
```

Remarquez ici que j'affiche le titre de l'événement, puis je crée un lien vers tous les composants de l'événement. Créons les deux autres nouveaux composants.

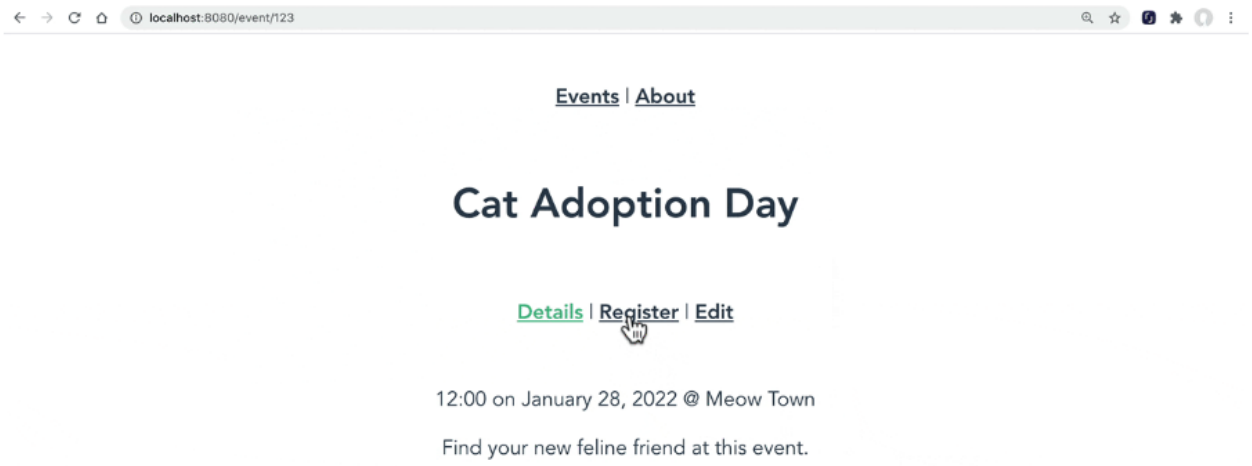
/src/views/event/Register.vue

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

const { id } = defineProps(['id'])

const event = ref(null)
onMounted(() => {
  EventService.getEvent(id)
    .then(response => {
      event.value = response.data
    })
    .catch(error => {
      console.log(error)
    })
})
</script>
<template>
  <div v-if="event">
    <h1>{{ event.title }}</h1>
    <div id="nav">
      <router-link :to="{ name: 'EventDetails', params: { id } }"
        >Details</router-link
      >
      |
      <router-link :to="{ name: 'EventRegister', params: { id } }"
        >Register</router-link
      >
      |
      <router-link :to="{ name: 'EventEdit', params: { id } }"
        >Edit</router-link
      >
    </div>
    <p>Regstration form here</p>
  </div>
</template>
```

/src/views/event/Edit.vue

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

const { id } = defineProps(['id'])

const event = ref(null)
onMounted(() => {
  EventService.getEvent(id)
    .then(response => {
      event.value = response.data
    })
    .catch(error => {
      console.log(error)
    })
})
</script>
<template>
  <div v-if="event">
    <h1>{{ event.title }}</h1>
    <div id="nav">
      <router-link :to="{ name: 'EventDetails', params: { id } }"
        >Details</router-link
      >
      |
      <router-link :to="{ name: 'EventRegister', params: { id } }"
        >Register</router-link
      >
      |
      <router-link :to="{ name: 'EventEdit', params: { id } }"
        >Edit</router-link
      >
    </div>
    <p>Edit the event here</p>
  </div>
</template>
```

Je n'ai pas encore déposé de formulaire d'inscription et la page d'édition n'est pas encore étoffée. Voici à quoi cela ressemble dans le navigateur :



4.5 Problème: Nous répétons l'en-tête et la navigation sur chaque page.

Vous remarquerez peut-être qu'il y a une certaine répétition. Plus précisément l'en-tête et la navigation. Ne serait-il pas bien si nous pouvions éliminer cette duplication ? Nous avons également le même code d'appel API dans chaque composant. Réparons cela aussi.

4.6 Solution: routes imbriquées

Dans cette application, nous avons déjà un niveau supérieur, <router-view> mais nous sommes tombés sur un cas où nous avons besoin d'une mise en page différente ou personnalisée pour tous les composants du profil d'événement. Allons-y et créons un nouveau composant appelé Layout.vue dans le répertoire /src/views/event/ pour la disposition des événements.

/src/views/event/Layout.vue

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

const { id } = defineProps(['id'])

const event = ref(null)
onMounted(() => {
  EventService.getEvent(id)
    .then(response => {
      event.value = response.data
    })
    .catch(error => {
      console.log(error)
    })
})
</script>
<template>
  <div v-if="event">
    <h1>{{ event.title }}</h1>
    <div id="nav">
      <router-link :to="{ name: 'EventDetails', params: { id } }"
        >Details</router-link
      >
      |
      <router-link :to="{ name: 'EventRegister', params: { id } }"
        >Register</router-link
      >
      |
      <router-link :to="{ name: 'EventEdit', params: { id } }"
        >Edit</router-link
      >
    </div>
    <router-view :event="event" />
  </div>
</template>
```

Notez que cette mise en page contient le code en double des 3 composants que nous avons répertoriés. Notez également que le router-view est affiché sous la forme <router-view :event="event" />, nous transmettons donc l'objet événement à partir de notre API afin de ne pas avoir à le récupérer à nouveau.

Maintenant, ces composants sont assez simples :

/src/views/event/Details.vue

```
<script setup>
defineProps(['event'])
</script>
<template>
  <p>{{ event.time }} on {{ event.date }} @ {{ event.location }}</p>
  <p>{{ event.description }}</p>
</template>
```

Notez que l'objet événement est transmis en tant que Props. Ensuite, il y a

/src/views/event/Register.vue

```
<script setup>
defineProps(['event'])
</script>
<template>
  <p>Register for the event here</p>
</template>
```

Et enfin

/src/views/event/Edit.vue

```
<script setup>
defineProps(['event'])
</script>
<template>
  <p>Edit the event here</p>
</template>
```

Il ne reste plus qu'à mapper ces routes imbriquées ensemble dans notre fichier de routeur.

Nous procédons ainsi :

/src/router/index.js

```
import { createRouter, createWebHistory } from 'vue-router'
import EventList from '../views/EventList.vue'
import EventLayout from '../views/event/Layout.vue'
import EventDetails from '../views/event/Details.vue'
import EventRegister from '../views/event/Register.vue'
import EventEdit from '../views/event/Edit.vue'
import About from '../views/About.vue'

const routes = [
  {
    path: '/',
    name: 'EventList',
    component: EventList,
    props: route => ({ page: parseInt(route.query.page) || 1 })
  },
  {
    path: '/event/:id',
    name: 'EventLayout',
    props: true,
    component: EventLayout,
    children: [  // <-----
      {
        path: '',
        name: 'EventDetails',
        component: EventDetails
      },
      {
        path: 'register',
        name: 'EventRegister',
        component: EventRegister
      },
      {
        path: 'edit',
        name: 'EventEdit',
        component: EventEdit
      }
    ]
  },
  ...
```

Il y a quelques points auxquels il faut prêter une attention particulière ici. La première consiste à remarquer notre route EventLayout avec l'option children, qui envoie un autre tableau de routes. Ensuite, remarquez comment les enfants héritent du chemin /event/:id de la route parent. Étant donné que EventDetails a un chemin vide, c'est ce qui est chargé par <router-view /> lors de la visite /event/:id. Ensuite, comme vous pouvez vous y attendre /user/:id/register, /user/:id/edit charge simplement les routes appropriés.

Comme on peut s'y attendre, tout fonctionne très bien :

4.7 Encore une optimisation

Après avoir montré ce code à Eduardo San Martin Morote (@posva) qui gère la bibliothèque Vue Router, il m'a suggéré de faire une optimisation supplémentaire du code, en ce qui concerne le layout.vue. Jetez un œil aux liens de navigation tels qu'ils se trouvent actuellement dans ce fichier :

/src/views/event/Layout.vue

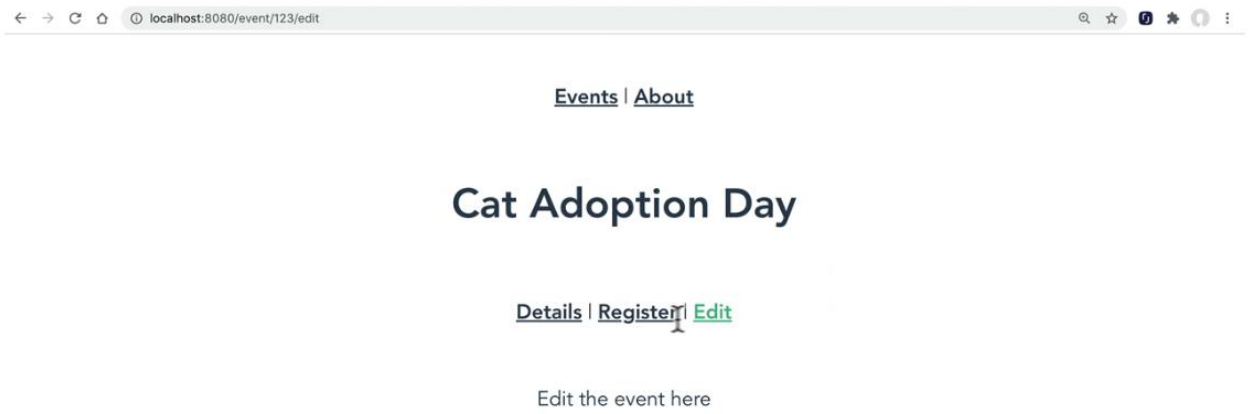
```
<template>
  <div v-if="event">
    <h1>{{ event.title }}</h1>
    <div id="nav">
      <router-link :to="{ name: 'EventDetails', params: { id } }"
        >Details</router-link
      >
      |
      <router-link :to="{ name: 'EventRegister', params: { id } }"
        >Register</router-link
      >
      |
      <router-link :to="{ name: 'EventEdit', params: { id } }"
        >Edit</router-link
      >
    </div>
    ...
```

Remarquez spécifiquement params: { id }. Il s'avère que nous pouvons supprimer cela, et les liens fonctionnent toujours parfaitement :

/src/views/event/Layout.vue

```
<template>
  <div v-if="event">
    <h1>{{ event.title }}</h1>
    <div id="nav">
      <router-link :to="{ name: 'EventDetails }"
        >Details</router-link
      >
      |
      <router-link :to="{ name: 'EventRegister' }"
        >Register</router-link
      >
      |
      <router-link :to="{ name: 'EventEdit' }"
        >Edit</router-link
      >
    </div>
    ...
```

Comment cela marche-t-il ? Eh bien, puisque ces liens nécessitent tous :id, lorsque le router-link est rendu dans le template (s'il n'est pas envoyé), il examinera les paramètres de l'URL, et s'il :id existe dans la route actuelle, il utilisera le :id dans toutes les URL des liens. Donc, de retour dans notre navigateur, tout fonctionne toujours :



5. Redirection et alias

À mesure que notre application évolue, nous devrons peut-être modifier les chemins d'URL de l'endroit où nos pages ont été initialement trouvées. Il existe deux méthodes pratiques pour cela :

5.1 Problème: Modification des routes

Que se passerait-il si nous devions modifier notre application pour passer /about de notre page À propos à /about-us. Comment pourrions-nous gérer cela ?

5.2 Solution n°1: Redirection

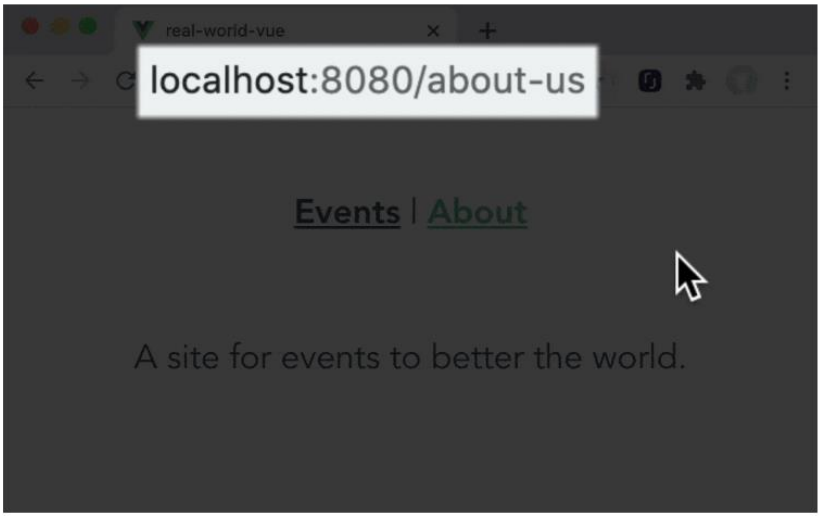
Évidemment, la première étape consiste à modifier notre route initiale :

```
const router = new VueRouter({
  routes: [
    ...
    {
      path: '/about-us',
      name: 'About',
      component: About
    }
  ]
})
```

Si nous utilisons des routes nommées, nous n'avons pas du tout besoin de modifier nos router-links. Sinon, nous devrions le faire. Ensuite, comme il peut y avoir des liens sur Internet vers notre /about page, nous souhaitons effectuer cette redirection de /about vers /about-us, avec la route supplémentaire suivante.

```
const router = new VueRouter({
  routes: [
    ...
    {
      path: '/about',
      redirect: { name: "About" }
    }
  ]
})
```

Notez que nous utilisons la route nommée pour la redirection. Nous aurions également pu utiliser `redirect: "/about-us"` la même fonctionnalité, mais il s'agit de coder en dur une URL à un endroit supplémentaire que nous aurions dû modifier si le chemin changeait. Voici à quoi cela ressemble :

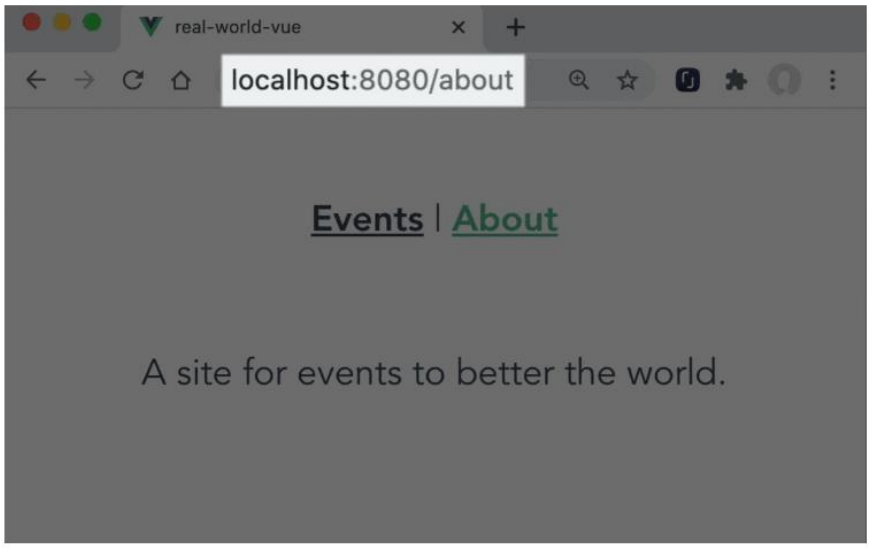


5.3 Solution n°2: Alias

Au lieu de rediriger l'ancien chemin, nous souhaiterions peut-être simplement lui donner un alias, ce qui signifie simplement fournir un chemin en double vers le même contenu. Nous pourrions mettre à jour ce chemin et fournir un alias à l'ancien chemin :

```
const router = new VueRouter({
  routes: [
    ...
    {
      path: '/about-us',
      name: 'About',
      component: About,
      alias: '/about' // <-----
    }
  ]
})
```

Désormais, l'utilisateur peut accéder à `/about` ou `/about-us` et il obtiendra le même contenu.



5.4 Problème: routes complexes

Dans l'application que nous avons créée dans notre cours Touring Vue Router pour visualiser un événement nous nous rendons à l'URL /event/123. Certains développeurs préféreront peut-être que cette URL soit /events/123 au pluriel. Supposons que nous souhaitions effectuer cette modification et veillons à ce que toutes nos anciennes URL soient correctement redirigées vers la nouvelle URL.

5.5 Solution: redirection des segments dynamiques

Pour rediriger un segment dynamique, nous devrons accéder aux paramètres lorsque nous créons le nouveau chemin. Pour ce faire, nous devrons envoyer une fonction anonyme dans la propriété redirect, comme ceci :

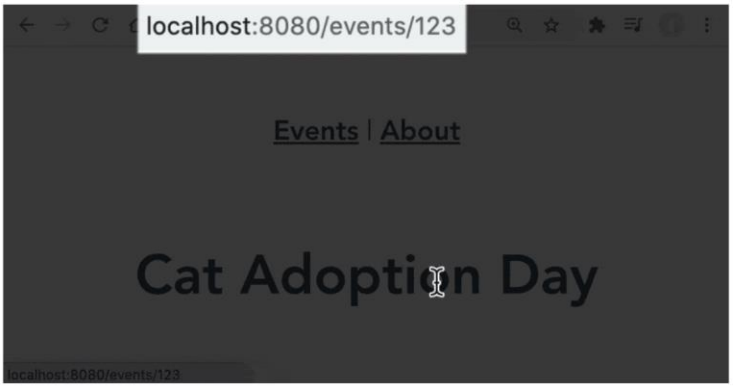
/src/router/index.js

```
...
const routes = [
  ...
  {
    path: '/events/:id', // <--- make plural 'events'
    name: 'EventLayout',
    ...
  },
  {
    path: '/event/:id',
    redirect: to => {
      return { name: 'EventDetails', params: { id: to.params.id } }
    }
  },
```

Remarquez comment, à l'intérieur de cette fonction anonyme, nous pourrions appliquer une logique complexe si nécessaire. Il s'avère que nous pouvons simplifier ce que nous avons écrit ci-dessus, car le paramètre id sera transmis automatiquement. Vue Router est intelligent donc nous pouvons simplifier comme ceci :

```
{
    path: '/event/:id',
    redirect: () => {
      return { name: 'EventDetails' }
    }
},
```

L'identifiant dans l'URL sera transmis lorsque nous redirigerons de /event/123 vers /events/123, comme vous pouvez le voir ci-dessous :



Cependant, si vous avez regardé la dernière leçon, vous savez que nous avons deux routes enfants imbriquées sous /events, en particulier les routes register et edit. La solution ci-dessus ne prend pas en compte ces routes enfants, que vous pouvez voir dans notre fichier de routeur ressemblant à ceci :

```
{
    path: '/events/:id',
    name: 'EventLayout',
    props: true,
    component: EventLayout,
    children: [
      {
        path: '',
        name: 'EventDetails',
        component: EventDetails
      },
      {
        path: 'register',  <---- How do we redirect for this?
        name: 'EventRegister',
        component: EventRegister
      },
      {
        path: 'edit',   <---- and this?
        name: 'EventEdit',
        component: EventEdit
      }
    ]
}
```

Eh bien, il existe deux solutions.

5.6 ✓ Rediriger avec les enfants

Il s'avère que la redirection a la capacité d'accepter des enfants. Nous pouvons donc faire ceci :

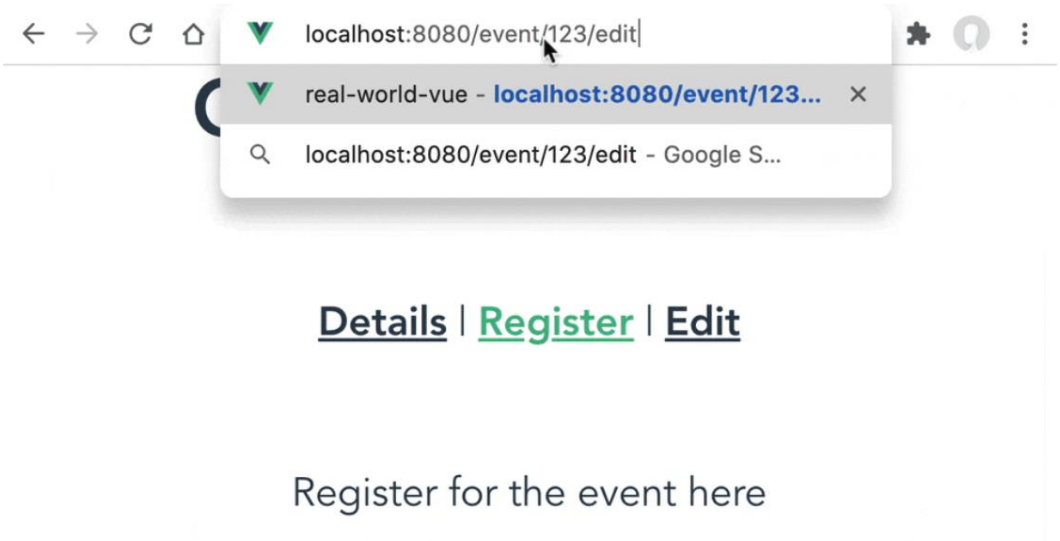
```
{
   path: '/event/:id',
   redirect: () => {
     return { name: 'EventDetails' }
   },
   children: [
     { path: 'register', redirect: () => ({ name: 'EventRegister' }) },
     { path: 'edit', redirect: () => ({ name: 'EventEdit' }) }
   ]
 },
```

5.7 ✓ Redirection avec Wildcard

Une autre façon de résoudre ce problème consiste à utiliser un caractère générique, comme ceci :

```
{
    path: '/event/:afterEvent(.*)',
    redirect: to => {
      return { path: '/events/' + to.params.afterEvent }
    }
  },
```

Cela consiste à prendre tout ce qui vient après le mot correspondant /event/ et à le placer après /events/. C'est moins de code et couvre tous les routes enfants. Voilà, ça marche :



6. Navigation programmatique

La navigation est déclenchée à l'aide de Vue Router lorsqu'un <router-link> est cliqué, mais elle peut également être déclenchée de manière programmatique à partir de notre code. Dans cette leçon, nous verrons plus en détail comment cela peut fonctionner. Laissez-moi vous montrer où cela serait nécessaire, avant de vous montrer la nouvelle syntaxe.

6.1 🛑 Problème: Soumission de formulaire

L'endroit le plus courant où vous voudrez naviguer par programme est lorsqu'un formulaire est soumis. Il est probable que vous ne souhaitiez modifier la navigation que si la création de la nouvelle ressource est réussie.

Examinons donc un exemple simple, dans notre fichier Register.vue :

```
<script setup>

defineProps(['event'])

const register = () => {
  // If registration API call is successful
  // Push back to the event details
}
</script>

<template>
  <p>Regstration form here</p>
  <button @click="register">Register Me!</button>
</template>
```

Par souci de simplicité, nous allons sauter l'appel à l'API lui-même, supposer qu'il a été couronné de succès et trouver comment naviguer vers la page des détails de l'événement.

6.2 ✓ Solution: router.push

La solution ici est un appel à router.push, avec les mêmes arguments que lorsque nous utilisons <router-link>. Donc dans notre cas :

```vue
<script setup>
import { useRouter } from 'vue-router'

defineProps(['event'])
const router = useRouter()

const register = () => {
  // If registered then redirect to event details
  router.push({
    name: 'EventDetails'
  })
}
</script>
<template>
  <p>Regstration form here</p>
  <button @click="register">Register Me!</button>
</template>
```

Voyons cela en action : browser-complete.gif

Remarque complémentaire : il serait intéressant d'envoyer un message à l'utilisateur sur cette page pour l'informer que son inscription a été effectuée avec succès. Nous verrons dans une prochaine leçon comment délivrer ce type de message de manière créative.

6.3 Autres exemples de push

Lorsque vous cliquez sur un <router-link>, vous appelez simplement la fonction router.push à l'intérieur du code. Comme vous pouvez l'imaginer, il existe toutes sortes de combinaisons que vous pouvez utiliser comme <router-link> :

```vue
// Directly to the path with a single string
router.push('/about')

// Directly to the path with an object
router.push({ path: '/about' })

// Directly to the named path
router.push({ name: 'About' })

// With a dynamic segment as I showed above
router.push({ name: 'EventDetails', params: { id: 3 } })

// With a query ... resulting in /?page=2 in our example app
router.push({ name: 'EventList', query: { page: 2 } })
```

Les paramètres et les valeurs de la requête peuvent être des variables, comme dans notre exemple ci-dessus.

6.4 Navigation et remplacement

Il arrive que vous souhaitiez faire naviguer l'utilisateur hors d'une page, sans pour autant créer une nouvelle entrée dans l'historique de son navigateur (ce qui rendrait le bouton "retour" inutile pour revenir à la page en cours). Peut-être que le formulaire que vous soumettez, une fois soumis, ne devrait pas être autorisé à être soumis à nouveau ? Attention : Cela pourrait vraiment perturber vos utilisateurs. 

```
router.replace({ name: 'EventDetails', params: { id: 3 } })
```

Cela remplacera la page sur laquelle je suis actuellement par la page EventDetails, de sorte que le bouton "back" ne renverra plus à la page actuelle. Les arguments sont les mêmes que ceux de router.push.

6.5 Naviguer dans la pile historique

Peut-être aimeriez-vous avoir un bouton personnalisé pour revenir en arrière et avancer dans votre interface. Cela peut être utile si vous construisez une application mobile native qui n'a pas les mêmes barres de navigation. Dans ce cas, vous pouvez utiliser go pour naviguer en avant et en arrière dans l'historique.

```
// Go forward a record
router.go(1)

// Go backward a record
router.go(-1)
```

7. Gestion des erreurs et 404

Il existe trois types d'erreurs que nous devons détecter dans notre application, et nous aurons des résultats différents pour chacune d'entre elles.

- Lorsqu'un utilisateur tente de naviguer vers une page qui n'existe pas.
- Lorsque la connectivité réseau d'un utilisateur est défaillante.
- Lorsqu'un utilisateur tente de se rendre à un événement qui n'existe pas.

Traitons chacun de ces problèmes un par un.

7.1 🛑 Problème: Le 404 générique non trouvé

Actuellement, lorsque je me rends à une URL qui n'existe pas, j'obtiens une page blanche sans aucune information sur ce qui se passe. 1.1611600265624.gif

7.2 ☑ Solution: Un composant "Non trouvé"

Créons un composant générique "Not Found" vers lequel nous redirigerons les utilisateurs qui accèdent à un chemin inexistant.

📄 **/src/views/NotFound.vue**

```
<template>
  <h1>Oops!</h1>
  <h3>The page you're looking for is not here.</h3>
  <router-link :to="{ name: 'EventList' }">Back to the home page</router-link>
</template>
```

Nous devons maintenant rendre ce composant lorsque nous passons à une route attrape-tout, en mettant à jour notre fichier router.js :

/src/router/index.js

```
...
import NotFound from '@/views/NotFound.vue'

const routes = [
  ...
  {
    path: '/:catchAll(.*)',
    name: 'NotFound',
    component: NotFound
  }
]
```

Comme vous pouvez le voir, nous créons un chemin qui capture tout ce qui ne correspond pas à une route actuelle de notre composant NotFound, puis nous redirigeons vers le chemin 404 lorsque nous atteignons notre nouvelle route attrape-tout.

Il y a une raison pour laquelle nous redirigeons vers notre page 404 (plutôt que de simplement rendre le composant), ce qui sera clair dans une minute. Maintenant, lorsque nous chargeons un chemin qui n'existe pas, comme /login, nous obtenons :2.1611600265625.gif

7.3 🛑 Problème : Que se passe-t-il lorsque nous essayons de visualiser un événement inexistant ?

Actuellement, lorsque nous allons sur un événement qui n'existe pas, comme /event/1233, nous obtenons une page blanche : 3.1611600271115.gif

Une meilleure solution serait d'ajouter notre composant NotFound et d'y faire naviguer l'utilisateur. Essayons-le !

7.4 ☑ Solution : Créer une page 404 pour un événement inexistant

Tout d'abord, dans le composant NotFound, je vais ajouter une propriété appelée resource qui prendra par défaut la valeur page. Ainsi, si ce composant est chargé sans la propriété, le texte sera "La page que vous cherchez n'est pas ici". Ainsi, lorsqu'un événement n'est pas trouvé, je peux envoyer event comme valeur de resource et le texte sera "L'événement que vous cherchez n'est pas là".

/src/views/NotFound.vue

```vue
<script setup>
defineProps({
  resource: {
    type: String,
    required: true,
    default: 'page'
  }
})
</script>

<template>
  <div>
    <h1>Oops!</h1>
    <h3>The {{ resource }} you're looking for is not here.</h3>
    <router-link :to="{ name: 'EventList' }">Back to the home page</router-link>
  </div>
</template>
```

Maintenant, je dois mettre à jour le routeur, donc cette route prend un prop :

/src/router/index.js

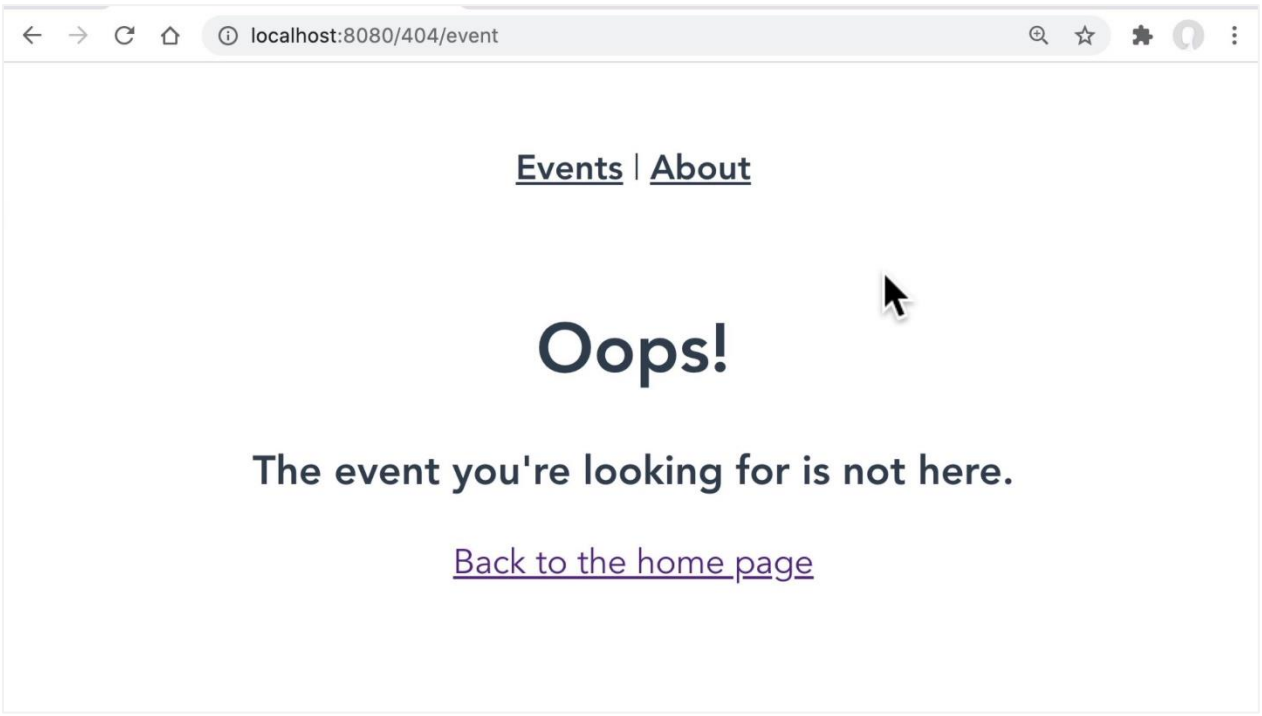
```
...
{
  path: '/404/:resource',
  name: '404Resource',
  component: NotFound,
  props: true
},
...
```

Et maintenant, je peux rediriger l'utilisateur ici si la requête API lors de l'obtention d'un événement renvoie une erreur :

/src/views/Layout.vue

```
<script setup>
import { onMounted, ref, defineProps, computed } from "vue";
import EventService from "@/services/EventService.js";
import { useRouter } from "vue-router"

const router = useRouter();

...

onMounted(() => {
  EventService.getEvent(id)
    .then(response => {
      event.value = response.data
    })
    .catch((error) => {
      router.push({
        name: '404Resource',
        params: { resource: 'event' }
      })
    });
 })
 ...
```

Maintenant, lorsque nous nous rendons sur le site http://localhost:8080/event/123123123, nous voyons :



C'est génial ! Désormais, lorsque des personnes créent un lien vers des événements qui n'existent pas, nous disposons d'une page 404 très attrayante. Il est inévitable que les utilisateurs tombent sur cette page puisque nos utilisateurs peuvent supprimer des événements.

7.5 🛑 Problème: Les erreurs ne seront pas toujours 404

Le seul problème avec cette solution est que nous supposons que toutes les erreurs de réseau que nous recevons sont des erreurs 404. Cependant, que se passe-t-il si l'Internet de l'utilisateur vient de tomber en panne ou s'il dispose d'une connexion très lente ? Nous ne voulons pas qu'il reçoive une erreur 404 si son Internet tombe en panne, mais c'est ce qui va se passer pour l'instant, alors corrigeons cela.

7.6 ☑ Solution: Un composant NetworkError

Nous allons commencer par créer un nouveau composant :

/src/views/NetworkError.vue

```
<template>
  <div class="networkError">
    <h1>Uh-Oh!</h1>

    <h3>
      It looks like you're experiencing some network issues, please take a breath and
      <a href="#" @click="$router.go(-1)">click here</a> to try again.
    </h3>
  </div>
</template>
```

Remarquez que j'utilise ici la méthode `$router.go(-1)` pour renvoyer l'utilisateur à la page précédente qui a déclenché l'erreur de réseau. Nous avons appris cela dans la leçon précédente.

Ensuite, nous ajouterons ceci à notre routeur :

/src/router/index.js

```
import NetworkError from "@/views/NetworkError.vue";

...
{
  path: '/network-error',
  name: 'NetworkError',
  component: NetworkError
},
..
```

Enfin, nous devons vérifier quel type d'erreur de réseau nous avons rencontré afin de pouvoir pousser vers le chemin approprié à l'intérieur de notre Layout.vue :

/src/views/event/Layout.vue

```
...
import { useRouter } from 'vue-router'
const router = useRouter()
...
onMounted(() => {
  EventService.getEvent(id)
    .then(response => {
      event.value = response.data
    })
    .catch(error => {
      if (error.response && error.response.status == 404) {
        router.push({
          name: '404Resource',
          params: { resource: 'event' }
        })
      } else {
        router.push({ name: 'NetworkError' })
      }
    })
})
...
```

Désormais, si notre serveur API tombe en panne ou met trop de temps à revenir, nos utilisateurs recevront une erreur de réseau au lieu d'obtenir une page web qui se charge indéfiniment ! Voir ci-joint :5.1611600276047.gif

Nous devrions également rediriger vers le composant NetworkError si l'appel à la liste des événements dans le fichier EventList.vue échoue.

/src/views/EventList.vue

```
...
import { useRouter } from "vue-router"

const router = useRouter();
...

     EventService.getEvents(2, page.value)
      .then((response) => {
        events.value = response.data;
        totalEvents.value = response.headers["x-total-count"];
      })
      .catch(() => {
        router.push({ name: 'NetworkError' }) // <<-----
      });
```

7.7 << ReVue

Dans cette leçon, nous avons appris à créer une page d'erreur fourre-tout lorsque l'URL ne correspond à aucune de nos routes. Nous avons ensuite montré comment créer une page 404 Not Found lorsque l'utilisateur navigue vers un événement qui n'existe pas, puis comment filtrer ces erreurs 404 des autres erreurs de réseau.

8. Messages Flash

Bien que cette leçon ne vous apprenne rien de nouveau avec Vue Router, elle couvrira une fonction de notification d'application web courante et utilisera certaines fonctionnalités de Vue 3 pour créer un stockage global. Mais d'abord, laissez-moi vous montrer le problème que nous essayons de résoudre.

8.1 Problème: Délivrer un message temporaire

Souvent, dans notre application, nous voudrons faire clignoter à l'écran un message pour l'utilisateur qui disparaît au bout de 5 secondes. Il existe différents noms pour cette fonctionnalité, mais j'ai appris à la connaître sous le nom de "Message Flash". Le premier endroit où il peut être intéressant d'avoir un message flash est après l'inscription d'un utilisateur à un événement, pour lui faire savoir qu'il est inscrit lorsque nous le renvoyons à la page de l'événement :

/src/views/event/Register.vue

```
<script setup>
import { useRouter } from 'vue-router'

defineProps(['event'])
const router = useRouter()

const register = () => {
  // Call to API
  // If registered then redirect to event details
  router.push( {
    name: 'EventDetails'
  })
}
</script>
<template>
  <p>Regstration form here</p>
  <button @click="register">Register Me!</button>
</template>
```

Comme vous pouvez le voir ci-dessus, avant de repousser notre navigation vers la page des détails de l'événement, nous aimerions faire apparaître un message flash sur cette page pour indiquer que l'utilisateur s'est enregistré avec succès. Ce type de message flash peut également être utile lorsqu'un utilisateur est connecté ou déconnecté avec succès.

8.2 ☑ Solution: Stockage global et fourniture / injection Global Storage & Provide /Inject

Il y a plusieurs façons de résoudre ce problème, mais comme toujours je veux vous montrer la solution la plus simple et la plus propre. Ma solution comporte trois étapes :

1. Créer un mécanisme de stockage global pour stocker notre message flash.
2. Définir le message flash pour l'enregistrement dans notre Register.vue.
3. Créer un endroit où le message flash est affiché dans notre App.vue.

8.3 Étape 1: Stockage global

Vous vous dites peut-être : **"N'est-ce pas la raison d'être de Vuex ? Disposer d'un mécanisme de stockage global auquel on peut accéder à partir de différentes**

parties de mon application Vue ? " La réponse est oui, c'est exactement le problème que Vuex résout. Cependant, Vuex peut être complexe à comprendre (et vous n'avez pas besoin de le connaître si vous êtes en train d'apprendre) et nous pouvons utiliser une solution plus simple (et toujours élégante) pour l'instant grâce à Vue 3.

Vue 3 a découplé la réactivité des composants. Cela nous permet de déclarer un objet réactif indépendant de nos composants, puis d'injecter cet objet réactif dans les composants qui en ont besoin. De retour dans notre main.js où notre instance Vue est créée, nous pouvons créer un Global Store, que nous appellerons GStore.

/src/main.js

```
import { createApp, reactive } from "vue";
import App from "./App.vue";
import router from "./router";

const app = createApp(App);

app.use(router);

// Create a reactive object
const GStore = reactive({ flashMessage: '' })
app.provide('GStore', GStore)  // provide this object so others can inject it

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

Veillez à lire les commentaires dans le code ci-dessus. Vous pouvez considérer GStore comme un objet JavaScript qui a une réactivité, ce qui signifie que lorsque l'une des propriétés est modifiée, elle est mise à jour sur notre page web. Vous pouvez en apprendre davantage sur la syntaxe réactive dans mon cours Vue 3 Composition API, je n'entrerai donc pas dans les détails ici.

Provide existe depuis Vue 2 et nous permet de spécifier les données que nous voulons **injecter** dans d'autres composants. Vous pouvez le considérer comme une alternative à l'utilisation de props et d'événements pour la communication entre composants. Nous utiliserons ensuite inject pour y accéder.

8.4 Étape 2 : Création du message flash

De retour dans notre composant Register, nous pouvons maintenant injecter notre GStore et l'utiliser pour stocker notre flashMessage :

/src/views/event/Register.vue

```
<script setup>
import { useRouter } from 'vue-router'
import { defineProps, inject } from "vue";

const { event } = defineProps(['event'])
const router = useRouter()
const GStore = inject('GStore')

const register = () => {
  // Call to API
  // If registered then redirect to event details
  GStore.flashMessage = 'You are successfully registered for ' + event.title
  setTimeout(() => {
    GStore.flashMessage = ''
  }, 3000)
  router.push({
    name: 'EventDetails'
  })
}
</script>
```

Deux choses à noter ci-dessus. Tout d'abord, l'option inject component, qui donne à notre composant l'accès à notre GStore. Ensuite, nous définissons notre message flash, et nous fixons un Timeout pour que le flashMessage soit effacé après 4 secondes.3 plutôt?

8.5 Étape 3: Affichage du message Flash

De nombreux composants de notre application créeront ces messages d'utilisateur, et nous devons donc les afficher dans notre présentation principale, qui dans notre application est notre App.vue :

/src/views/App.vue

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

const GStore = inject('GStore') // <----
</script>

<template>
  <div id="app">
    <div id="flashMessage" v-if="GStore.flashMessage">
      {{ GStore.flashMessage }}
    </div>
    ...
</template>

<style>
@keyframes yellowfade {
  from {
    background: yellow;
  }
  to {
    background: transparent;
  }
}

#flashMessage {
  animation-name: yellowfade;
  animation-duration: 3s;
}

...
</style>
```

Remarquez que nous injectons et affichons le flashMessage s'il existe, et j'ai ajouté une animation pour qu'il soit surligné en jaune et qu'il disparaisse (voir le style) lorsque la div apparaît à l'écran, de sorte qu'il attire les yeux de l'utilisateur. Dans le navigateur, cela ressemble à ceci :finished-1920.gif

8.6 Dans ReVue

Nous disposons maintenant d'un message Flash que nous pouvons utiliser dans notre application. Nous disposons également d'un magasin global dont nous aurons besoin dans les prochaines leçons pour stocker les données récupérées par notre API. Pourquoi ? Parce que nous voudrons parfois nous assurer que notre appel à l'API a abouti avant de charger notre composant. Pour l'instant, nous chargeons notre composant, qu'il soit réussi ou non, et notre appel à l'API

est couplé à notre composant lui-même. Ce n'est pas forcément ce que nous voulons, et nous expliquerons pourquoi dans la prochaine leçon.

9. Récapitulation

Nous avons couvert certaines des principales fonctionnalités de Vue Router dans ce cours, et c'est un excellent travail que d'être arrivé jusqu'ici. Dans cette leçon, nous allons jeter un coup d'œil rapide à certaines fonctionnalités avancées et aux raisons pour lesquelles vous pourriez les utiliser, telles que les champs méta de route, les routes à chargement paresseux et le comportement de défilement.

9.1 Champs méta de route

Il arrive que vous souhaitiez associer des informations à une série de routes et les définir au niveau du routeur. Le meilleur exemple est celui de l'autorisation, lorsque vous souhaitez que seules certaines personnes puissent suivre certaines routes. Il se peut que seuls les utilisateurs connectés puissent voir certaines routes, ou que seuls les utilisateurs puissent modifier le contenu qu'ils ont créé (et non celui des autres utilisateurs). Dans ce cas, il est utile d'intégrer l'autorisation au niveau du routeur, plutôt qu'au niveau du composant.

Jouons avec cela dans notre routeur existant du cours. Pour info, je ne fais que jouer, donc vous ne trouverez pas ce code dans github, mais vous pouvez le copier et le coller.

/src/router/index.js

```
...
{
  path: 'edit',
  name: 'EventEdit',
  component: EventEdit,
  meta: { requireAuth: true }
}
```

Comme vous pouvez le voir, l'option meta prend simplement un hachage, et vous pouvez placer n'importe quelle valeur dans ce hachage. Cette information peut ensuite être utilisée partout où nous avons accès à un objet route (généralement dans les arguments to ou from).

Voici un exemple d'utilisation de ces informations dans le fichier du routeur lui-même :

```
router.beforeEach((to, from) => {
  NProgress.start()

  const notAuthorized = true
  if (to.meta.requireAuth && notAuthorized) {
    GStore.flashMessage = 'Sorry, you are not authorized to view this page'

    setTimeout(() => {
      GStore.flashMessage = ''
    }, 3000)

    return false
  }
})
```

J'utilise notAuthorized comme une constante pour l'instant, mais cela pourrait facilement être l'appel d'une autre méthode d'une bibliothèque qui a tout le code d'autorisation. J'utilise le flashMessage que nous avons créé pour la première fois dans la leçon bla-bla, et enfin return false indique à Vue Router d'arrêter simplement la navigation. Voici à quoi cela ressemble :1.1622835295500.gif

Cette solution pose toutefois un problème. Lorsque je navigue directement vers l'URL pour modifier la page, j'obtiens une page blanche qui ressemble à ceci :2.1622835295501.gif

Pour contourner ce problème, nous pouvons vérifier si la route from est associée à un href. Si ce n'est pas le cas, il doit s'agir d'une personne qui se rend directement à cette URL, auquel cas nous la redirigeons vers le chemin racine : la liste des événements :

```
router.beforeEach((to, from) => {
  NProgress.start()

  const notAuthorized = true
  if (to.meta.requireAuth && notAuthorized) {
    GStore.flashMessage = 'Sorry, you are not authorized to view this page'

    setTimeout(() => {
      GStore.flashMessage = ''
    }, 3000)

    if (from.href) { // <--- If this navigation came from a previous page.
      return false
    } else {  // <--- Must be navigating directly
      return { path: '/' }  // <--- Push navigation to the root route.
    }
  }
})
```

Nous pouvons maintenant constater que la navigation directe vers la page donne lieu à un message d'erreur et à une redirection vers la page d'accueil.
3.1622835300541.gif

Il est intéressant de mentionner que lorsque l'on utilise des routes enfants, comme nous le faisons avec notre EventLayout dans notre application d'exemple, nous pouvons placer des options méta sur notre route parent, et elles seront automatiquement héritées par les routes enfants. Cela peut être utile lorsque nous avons des pages d'administration qui requièrent toutes une autorisation.

9.2 Chargement à la volée

Lors de la création d'applications contenant de grandes quantités de JavaScript, le chargement initial de la page (où le JavaScript est téléchargé) peut commencer à devenir important et ralentir la vitesse de chargement de l'application. Il se peut également que certaines parties de votre application ne soient chargées que par certaines personnes, par exemple les créateurs de contenu par rapport aux consommateurs de contenu. Sur YouTube, il y a beaucoup plus de gens qui regardent des vidéos que de gens qui en créent. Les personnes qui se contentent de regarder des vidéos n'ont pas besoin de télécharger tout le JavaScript nécessaire au chargement des pages web utilisées par les créateurs de vidéos.

Le Lazy Loading nous permet de séparer notre application en bundles JavaScript distincts, de sorte qu'un bundle n'est téléchargé qu'en cas de besoin.

Vous avez peut-être déjà vu la version de base du lazy loading lorsque vous avez créé votre première application Vue. Le routeur Vue contient par défaut ceci :

/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(/* webpackChunkName: "about" */ '../views/About.vue')
  }
```

Si nous chargeons la page web de base et que nous regardons dans l'onglet Réseau de Chrome Dev Tools, nous pouvons voir que ce n'est que lorsque nous cliquons sur le lien "à propos" que le JavaScript de la page à propos est chargé. 4.1622835303324.gif

Remarquez que le JavaScript chargé s'appelle "about.js" ; c'est le nom de la page web que nous avons spécifié dans le commentaire.

Une autre façon de voir le code ci-dessus écrit en Vue est la suivante :

```
const About = () => import(/* webpackChunkName: "about" */ '../views/About.vue')

const routes = [
  ...
  {
    path: '/about',
    name: 'About',
    component: About
  }
```

Comme vous pouvez l'imaginer, si vous avez un tas de pages de créateurs YouTube, vous utiliserez le même webpackChunkName, comme ceci :

```
const Uploader = () => import(/* webpackChunkName: "creator" */
'../views/Uploader.vue')
const Editor = () => import(/* webpackChunkName: "creator" */ '../views/Editor.vue')
const Publisher = () => import(/* webpackChunkName: "creator" */
'../views/Publisher.vue')
```

Désormais, lorsque l'on navigue vers l'une de ces routes, le paquet JavaScript creator.js est chargé sur la page.

9.3 Comportement de défilement

Remarquez que lorsque nous sommes sur Google et que nous recherchons Vue Mastery, lorsque nous faisons défiler la page vers le bas et que nous cliquons sur le bouton suivant, nous apparaissons en haut de la deuxième page de résultats : 5.1622835306037.gif

Un effet secondaire involontaire d'une application à page unique est que lorsque vous faites défiler une page vers le bas et que vous cliquez sur un bouton, vous n'êtes pas (par défaut) ramené en haut de la page. Voir notre pagination des événements : 6.1622835310806.gif

Heureusement, nous pouvons facilement doter notre application de cette fonctionnalité en ajoutant un peu de code à notre routeur :

/src/router/index.js

```
...
const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes,
  scrollBehavior() {  // <---
    // always scroll to top
    return { top: 0 }
  }
})
...
```

Désormais, lorsque nous naviguerons, nous serons ramenés en haut de la page vers laquelle nous avons navigué : 7.1622835314307.gif

C'est très bien, mais il y a un autre comportement que nous pourrions souhaiter. Sur Google, lorsque nous faisons défiler une page jusqu'en bas, que nous cliquons pour passer à la page suivante, puis que nous utilisons le bouton "Précédent", nous sommes ramenés à l'endroit où nous nous trouvions (en faisant défiler la page jusqu'en bas). 8.gif

C'est un comportement auquel nous ne pensons pas vraiment, mais auquel nous nous attendons. Lorsque nous appuyons sur le bouton Précédent, nous nous attendons à être ramenés à l'endroit où nous venons de partir. Cependant, nous venons de dire à notre application Vue d'aller en haut de la page à chaque navigation (même en arrière).

Pour revenir à la même partie de la page que nous venons de quitter, il s'agit d'une petite modification :

```
...
const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes,
  scrollBehavior(to, from, savedPosition) {
    if (savedPosition) { // <----
      return savedPosition
    } else {
      return { top: 0 }
    }
  }
})
...
```

S'il existe une position sauvegardée pour cette page, elle reviendra correctement à l'endroit où nous venons de nous trouver. Nous pouvons voir cela dans notre exemple d'événement : 9.gif

9.4 Conclusion

Comme vous pouvez le voir, Vue Router fournit des outils très puissants pour créer notre application Vue. Il y a quelques sujets supplémentaires que vous pouvez lire dans la documentation de Vue Router, mais j'espère vraiment que ce cours vous a donné une base solide pour continuer à construire des applications Vue.