Vue.js

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Le Router

Un routeur ?

Les application Vue.js sont par défaut uniquement disponible à une seule adresse de par l'envoi d'un seul fichier index.html modifié par la suite via le code métier.

En cas de volonté de permettre une navigation entre les pages de notre application et ainsi offrir aux utilisateurs la possibilité d'avoir des liens en favoris ou de partager leurs pages préférées avec leurs amis, il nous faut un moyen d'accéder directement à une page donnée. Pour cela, on va se baser sur la logique du routing avec la barre de navigation de notre navigateur. Chaque url sera analysée et amènera à une page donnée. On parle alors de routing.

Installer le routing

Pour ajouter le routing à notre application, il suffit d'installer la chose via npm:

```
npm install vue-router@latest
```

Plusieurs éléments sont par la suite disponible via le routing:

- `router-link`: Permet de rediriger l'utilisateur vers une page donnée
- `router-link`: Indique à notre page / composant qu'on va injecter ici des composants provenant du mécanisme de routing.

Utiliser le router

Pour demander à Vue de faire usage du router comme élément d'entrée dans l'application, on va devoir l'ajouter à la chaine des middlewares de notre application:

```
const router = VueRouter.createRouter({
  history: VueRouter.createWebHashHistory(),
  routes,
});
```

On demande à la création du router de baser le mécanisme d'historique sur celui présent nativement dans le navigateur. On remarque cependant le passage d'un attribut `routes`.

Le tableau de routes

Dans notre application, il va falloir ensuite définir un objet routes, qui se constitue purement et simplement en tant que tableaux d'objet définissant, dans leur syntaxe la plus simple, des routes et le composant à afficher en cas de présence dans notre barre de navigation de cette route:

```
const routes = [
  { path: "/", component: Home },
  { path: "/about", component: About },
];
```

Naviguer entre les routes

Dans le cas où l'on cherche à permettre à un utilisateur d'aller sur une autre page, il va falloir éviter la balise traditionnelle `<a>` qui causerait un rechargement de notre application et une perte potentielle de tout son état. A la place, on va plutôt utiliser la balise fournie par le Vue Router:

```
<RouterLink to="/route/a/atteindre" />
```

Cette balise peut bien entendu gérer les liaison de données, ce qui permettrait par exemple une route peuplée par une variable:

```
<RouterLink :to="`/route/${variable}`" />
```

Naviguer par le code

Dans le cas où l'on aurait besoin, suite à des actions utilisateur, de provoquer du code en amont d'une navigation, il va falloir nous lier à un évènement levé par l'utilisateur. A la fin du code lié à l'évènement, nous devront provoquer la navigation. Pour cela, il nous faut avoir accès au routeur dans le code métier:

```
const router = useRouter();
```

Pour ensuite provoquer une navigation, il suffit d'utiliser la méthode dédié du routeur tel que:

```
router.push("/route/destination");
```

Les paramètres de route

Dans le cas où l'on veut avoir les détails d'un élément, il va falloir pouvoir rendre notre route dynamique, en passant par exemple dans la chaine de caractère l'ID de l'élément à un moment donné. Pour ce faire, on peut indiquer dans un objet de route l'utilisation de variable sous la forme:

```
{ path: '/product/:productId', component: ProductDetails }
```

Ici, la variable se nommera donc `productId` et permettra des routes telles que `/product/123` ou `/product/dsdsqdqd-sdqsdqsdq-sdqdq`

Extraire les paramètres de route

Au sein des pages / composants, la récupération des paramètres de la route est assez aisée. Il nous suffit de demander l'accès à la route actuelle via `useRoute()` et d'aller y piocher ce qui nous intéresse.

```
const route = useRoute();
```

On peut ainsi aller y chercher les paramètres de la route, ou les paramètres de requête via la syntaxe suivante:

```
const params = route.params;
const { paramName } = params;

const query = route.query;
const { qParamName } = query;
```

Page 404

En cas d'utilisation d'une route n'existant pas par l'utilisateur, on aimerait l'amener à une page d'erreur de type 404 - Not found. Pour cela, on peut, dans un objet route, utiliser une syntaxe particulière pour indiquer que toutes les routes atteindront cet élément:

```
{ path: '/:pathMatch(.*)*', component: NotFoundPage }
```

On peut également utiliser cette logique similaire à une regexp afin de rendre la suite d'une url non limité à une seule syntaxe:

```
{ path: '/static-value-:everyAfter(.*)*', component: ElementName }
```

La Redirection et les Alias

Dans le cas où l'on veut avoir plus facile de naviguer entre nos pages, il peut être utile de nommer les routes. Pour cela, on peut, en objet du tableau de route, avoir:

```
{ path: '/chemin/de/route', name: 'nomRoute', component: ComponentName },
```

Ensuite, en valeur d'une méthode de navigation

```
router.push({
  name: "nomRoute",
  params: { keyA: "valueA", keyB: "valueB" },
});
```

Ce qui donne la route: `/chemin/de/route?keyA="valueA"&keyB="valueB"`

Routes enfants

Dans le cas où l'on aimerait avoir un affichage commun pour plusieurs routes, on peut spécifier dans le composant que l'on veut avoir en commun une balise `<RouterView />` de sorte à indiquer l'injection de routes enfants. Au niveau du tableau de routes, on va par la suite avoir une structure de la sorte:

```
const routes = [
  {
    path: "/product",
    component: ProductList,
    children: [
      { path: "details/:prodId", component: ProductDetails },
      { path: "add", component: ProductAdd },
    ]
```

Un peu de sécurité

Dans le cas où l'on veut protéger l'accès à nos routes, il est possible de faire appel au mécanisme des `Guards`. Ces éléments sont disponibles via la méthode `beforeEach()` de l'objet router défini de base à la racine de l'application:

```
router.beforeEach(async (to, from) => {
  if (to.path.startsWith("/private")) {
    const isLoggedIn = await getUserLogStatus();
    if (!isLoggedIn) return "/login";
  }
  return true;
});
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Store Pinia

Pinia ?

Si l'on a besoin, au sein d'une application front-end, de stocker des données utiles à notre application, ou si l'on cherche à stocker son état, il peut être intéressant de faire ceci de façon centralisée afin d'offrir à nos composants un point d'entrée et de sortie de ces données. De la sorte, on peut se passer de l'utilisation et du transit intempestifs de multiples props et simplement récupérer ou modifier depuis les composants en ayant besoin les données voulues. Pour cela, on peut utiliser le mécanisme de **store** que nous offre Pinia.

```
npm install pinia
```

Définir un store

Dans un premier temps, il va nous falloir créer l'endroit de centralisation des données, le store via `defineStore`:

```
import { ref, computed } from "vue";
import { defineStore } from "pinia";

export const useProductStore = defineStore("storeName", () => {
  const datas = ref([]);

  const dataForId = computed((id) => datas.value.find((x) => x.id === id));

  function add(data) {
    datas.value.push(data);
  }
  function removeById(dataId) {
    datas.value = datas.value.filter((x) => x.id !== dataId);
```

Exploiter le store

Pour aller piocher les données du store, au sein de nos composants, on va utiliser simplement notre store comme s'il s'agissait d'un composable classique, c'est à dire via la syntaxe suivante:

```
const productStore = useProductStore();
```

Une fois l'accès au store obtenu, on peut aller chercher les données brutes via les variables réactives:

```
<li v-for="p of productStore.datas">{p.name}</li>
```

Ou passer par les variables de type **computed**:

```
const product102 = productStore.dataForId(102);
```

Provoquer des changement

Dans le cas où l'on veut faire des modification des valeurs stockées dans le store, on va alors utiliser les fonction de mutation définies précédemment dans notre store:

```
const addProduct = productStore.add;

function submitHandler() {
  // Récupération des champs d'un formulaire présent dans le template

  addProduct(newProduct);
}
```

Actions asynchrones (1/2)

Pinia est capable de gérer les actions asynchrone facilement. En effet, une action de modification d'un store n'est ni plus ni moins qu'une fonction exportée. Celle-ci peut alors sans soucis être synchrone ou non.

Dans le cas où l'on est amené à réaliser des actions asynchrone, il est de bon ton de créer des variables réactives permettant de connaitre l'état de l'application (si la requête est terminée ou non, si elle a échouée ou non) en plus de celles relatives au données:

```
const pendingState = ref(false);
const error = ref(null);
```

Actions asynchrones (2/2)

```
async function addAsync(data) {
  error.value = null;
  pendingState.value = true;

  try {
    const isSuccess = await sendDataAsync(data);
    if (isSuccess) {
      datas.value.push(data);
      pendingState.value = false;
    } else throw new Error("Can't send data!");
  } catch (e) {
    error.value = e;
    pendingState.value = false;
  }
}
```