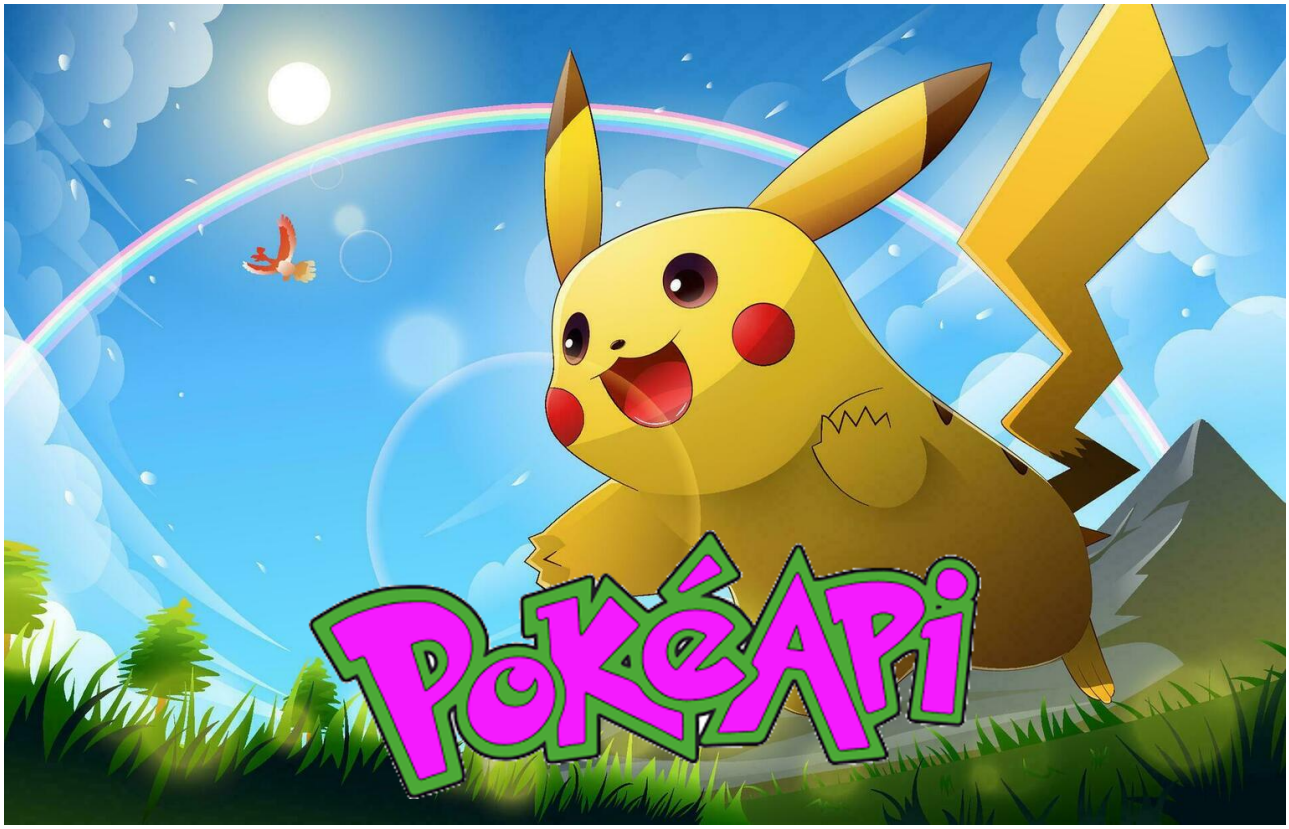

Explication sur les modes de connexions

Pour maintenir une connexion durable, il nous faut stocker des informations de connexion coté client. Il y a deux grandes méthodes pour cela:

1. La méthode **state-less**: Dans ce cas, cette information est une clé de sécurité sauvegarder dans le **localStorage** et passée manuellement dans l'en-tête de chaque requête.
2. La méthode **state-full**: Dans ce cas, cette information est sous la forme de cookie classique. Normalement les cookies ne fonctionne pas avec les appels **AJAX** mais **Axios** gère cela pour nous grâce à une certaine configuration expliquée ci-dessous.

Configuration de Axios

Modifier `CallerService` comme ceci:

```
const Axios = axios.create({
  baseURL: 'http://localhost:8000/api',
  headers: {
    Accept: 'application/json'
  },
  withCredentials: true, // précise à axios de mettre les cookies d'auth dans la
requête
```

```
  withXSRFToken: true,
  xsrfCookieName: 'XSRF-TOKEN',
  xsrfHeaderName: 'X-XSRF-TOKEN'
});
```

Le service d'authentification

Création d'un nouveau service `Account.service`:

```
import Axios from './CallerService';
import { useUserStore } from '@/stores/User';

export async function login(credentials: { email: string, password: string }):
Promise<void> {
  const res = await Axios.post('/authenticate', credentials, {
    headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
    baseURL: 'http://localhost:8000'
  });

  const userStore = useUserStore();
  userStore.setUser({
    email: res.data.user.email,
    role: res.data.user.role
  });
}

export function logout(): void {
  const userStore = useUserStore();
  userStore.clearUser();

  // partie js vanilla qui supprime tous les cookies du domaine
  const cookies = document.cookie.split(";");
  for (let i = 0; i < cookies.length; i++) {
    const cookie = cookies[i];
    const eqPos = cookie.indexOf('=');
    const name = eqPos > -1 ? cookie.substr(0, eqPos) : cookie;
    document.cookie = name + '=;expires=Thu, 01 Jan 1970 00:00:00 GMT;path=/';
  }
}

export function isLogged(): boolean {
  const userStore = useUserStore();
  return userStore.isLogged;
}
```

Le **store** user qui permet de pouvoir accéder aux infos de l'utilisateur partout dans l'application:

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

interface ConnectedUser {
  email: string | null;
  role: string | null;
};

export const useUserStore = defineStore('user', () => {
  const user = ref<ConnectedUser>({
    email: '',
    role: ''
  });

  user.value.email = localStorage.getItem('email'); // localStorage qui donne de la
persistance à nos données
  user.value.role = localStorage.getItem('role'); // localStorage qui donne de la
persistance à nos données

  const isLogged = computed(() => {
    return !!user.value.email;
  });

  function setUser(data: ConnectedUser) {
    user.value.email = data.email;
    user.value.role = data.role;
    localStorage.setItem('email', data.email ?? ''); // localStorage qui donne de la
persistance à nos données
    localStorage.setItem('role', data.role ?? ''); // localStorage qui donne de la
persistance à nos données
  }

  function clearUser() {
    setUser({
      email: '',
      role: ''
    });
  }

  return {
    user,
    isLogged,
    setUser,
```

```
    clearUser
  };
});
```

A présent ont peut lancer la connexion dans `Login.vue`:

```
AccountService.login(user.value).then(token => {
  router.push('/');
});
```

Ont rajoute au passage la déconnexion dans la barre de navigation. Ainsi qu'un petit test d'affichage/désaffichage des boutons de connexion/déconnexion. Si l'utilisateur se déconnecte, vous verrez l'affichage se mettre à jour automatique. C'est ce qu'ont appel la réactivité et c'est grâce au **store**.

```
<nav class="nav-auth">
  <div v-if="userStore.isLogged">
    <router-link to='/creatures-create' v-if="userStore.isLogged">Créer</router-link>
|
    <span>{{ userStore.user.email }}</span>
  </div>
  <router-link to='/login' v-if="!userStore.isLogged">Connexion</router-link> |
  <button @click="AccountService.logout()" v-
if="userStore.isLogged">Déconnexion</button> |
</nav>
```

Protection des routes

La dernière partie de ce tutoriel porte sur la sécurité. Il est vrai que la sécurité est très relative quand il s'agit du code coté client. Le minimum étant de faire le nécessaire pour que l'utilisateur ne puisse pas accéder aux vues qui lui sont interdit. Dans le router:

```
// router/index.ts
router.beforeEach((to, from, next) => {
  const userStore = useUserStore();
  const role = userStore.user.role;

  if (!to.meta.role) {
    return next();
  }
  else if (role == to.meta.role) {
    return next();
  }
```

```
  router.push('/login');
});
```

Nous utilisons ici les **meta** qui n'est ni plus ni moins qu'un moyen de "tagger" nos routes pour définir diverses stratégies applicative. Ici, la gestion des autorisations:

```
// Examples de protection user sur la route `/pokemons/create`
{ path: 'pokemons/create', name: 'pokemons-create', component: Public.PokemonCreate,
meta: { role: 'ROLE_USER' }}
```