Rapport Front

Bastien ACHARD

Github username : BastianACHARD
Github id : 43060957

Tâches effectuées :

J'ai principalement travaillé sur le front de notre application. J'ai commencé par init le front puis j'ai créé les différents dossiers pour que le front soit bien structuré et facile à comprendre pour les personnes extérieures au projet.
Pour le développement pure, la première chose que j'ai fait est de créer un fichier API qui me sert à mettre toutes mes requêtes qui communique avec le back. Ensuite, j'ai créé le premier context (nommé "stationContext") qui sert à stocker les données que je reçois du back. Je sais qu'il ne faut pas trop abuser des contexts mais pour ce type d'application qui est d'une taille assez petite, je trouve que c'est beaucoup plus pratique pour la gestion des données et aussi dans la structure de l'application.
Ainsi après avoir fini le contexte, j'ai pu me consacrer au développement des premiers Components (Map, ListStation, StationCard et FormFilter).
Ensuite, j'ai créé deux thèmes (Light et Dark) qui se sélectionne en fonction du thème du navigateur et l'enregistre dans le localStorage afin de garder le choix que l'utilisateur à fait.
Enfin, j'ai créé une requête pour récupérer la liste des prix moyens de chaque carburant sur les 10 derniers jours. J'ai dû créer un autre context pour stocker ces données.

Solutions choisies :

Comme je l'ai expliqué plus tôt, j'ai choisi d'utiliser les contexts afin de réunir la gestion des appels API et du stockage des données directement dans un context au lieu de les faire dans les components Parents. Je trouve que ça rend le code beaucoup plus lisible et ça limite l'utilisation des props.

Pour les appels API, j'utilise une lib nommée "Axios" tout simplement parce que j'ai déjà fait du react auparavant dans une entreprise. De même pour le fait que notre front soit en TypeScript. J'ai pris l'habitude de coder en TypeScript et maintenant je suis incapable de faire du front sans.

Pour les thèmes Light et Dark, j'ai suivi un tutoriel sur youtube car c'était la première fois que j'en faisais. J'ai utilisé la lib nommée "styled-components" pour ça. J'ai vu qu'il y avait plein de manière différente de faire ces thèmes mais j'ai choisi celle-ci car je la trouvais plus propre.

Pour la Map, j'ai utilisé LeafLet qui je trouve est une map très simple à utiliser et avec beaucoup de documentation. On aurait pu utiliser la map de google ou autres, mais ayant déjà fait un projet avec Leaflet, j'ai préféré choisir une map que je connais bien afin de ne pas perdre de temps, surtout qu'il est limité.

Pour le graph, Lucas a utilisé "Chart.js" et "react-chartjs-2". Je l'avais déjà utilisé en entreprise, c'est pour ça que je lui ai conseillé. De plus, Chart.js est super bien documenté et avec beaucoup d'exemples sur internet, ce qui le rend très simple d'utilisation.

Pour le formulaire (filtre), j'ai utilisé useForm avec la lib "react-hook-form", c'était la première fois que je l'utilisais mais je trouve ça vraiment trop bien pour la gestion des formulaires, c'est beaucoup plus simple et propre.

Difficultés rencontrées :

Je n'ai pas rencontré de réel problèmes durant le développement. Le seul point qui m'a bloqué est la gestion de la localisation de l'utilisateur en passant par le navigateur. J'ai voulu rendre cette gestion automatique mais j'ai des problèmes sur le rafraichissement des données dans mon context. C'est pour cela que j'ai dû passer par un bouton physique pour gérer ça.

Temps de développement / tâche :

- API => 4h
- Contexts => 8h
- Map => 3h
- Liste => 8h
- Filtre => 7h
- Thèmes => 3h
- Localisation => 4h

Code :

```tsx
// interface pour le context
interface contextType {
    stations: Station[],
    filter: Filter,
    updateStations: () => void,
    updateFilter: (filter: Filter) => void
}

// création du context
export const StationCtx = createContext<contextType | null>(null);

// Component qui sert de context provider
export const StationProvider: FC = ({ children }) => {
    // Hooks
    const [stations, setStations] = useState<Station[]>([]);
    const filt = {
        latitude: 48.856614,
        longitude: 2.3522219,
        radiusInMeter: 5000,
        fuels: [],
        services: [],
        sortByPrice: false
    }
    const [filter, setFilter] = useState<Filter>(filt);

    // update les données (la liste des stations)
    const updateStations = () => {
        console.log(filter);
        getFilterData(
            filter.latitude, filter.longitude, filter.radiusInMeter, filter.fuels,
            filter.services, filter.sortByPrice
        ).then(res => {
            setStations(res ? res : []);
        });
    }

    // update les données (les filtres)
    const updateFilter = (filter: Filter) => {
        setFilter(filter);
    }

    // les données qui sont passé dans le context
    const value: contextType = { stations, filter, updateStations, updateFilter };

    return (
        <StationCtx.Provider value={value}>
            {children}
        </StationCtx.Provider>
    );
};
```

Le premier code que je vous montre est celui du context qui gère les données des stations et des filtres. Il marche de la manière suivant :

- le component qui gère la liste des stations possède un useEffect qui lance le updateStations
- Le updateStations fait un appel API pour récupérer les stations avec le filtre par défaut, récupère les données et les stocks dans un useState nommé “stations”

- Quand on sélectionne des filtres différents dans le formFilter et que l'on valide cela déclenche le updateFilter qui set le filter avec les nouvelles données
- Par la suite, avec le useEffect cela va reload les données et ainsi appliqué le filtre

La gestion des données et des filtres dans un seul et même context apporte des avantages et inconvénients. Je trouve que ça simplifie la gestion des données en séparant cette gestion de l'utilisation de ces données dans les différents components. Les données se retrouvent au même endroit, ce qui simplifie les futures mises à jour.
Mais, je me rend compte de certains inconvénients, par exemple :

- En réalisant les filtres sur la même requête que la récupération des données, cela met beaucoup de pression sur le back qui fait énormément de calcul, ce qui rend les appels API assez longs. Il faudrait peut-être simplifier les filtres ou optimiser les calculs dans le back.

```
6
7   const ListStationsComponent: FC = () => {
8       // Hooks
9       const context = useContext(StationCtx);
10      useEffect(() => context!.updateStations());
11
12      // Variables
13      const data = context!.stations;
14      const stations = data.map(station => station);
15
16      return (
17          <div>
18              <FormFilterComponent />
19              <table>
20                  <tbody>
21                      {stations.map((station, index) => {
22                          return (
23                              <tr>
24                                  <td><StationComponent key={index}
25                                      latitude={station.latitude}
26                                      longitude={station.longitude}
27                                      adresse={station.adresse}
28                                      ville={station.ville}
29                                      listeDePrix={station.listeDePrix} />
30                                  </td>
31                              </tr>
32                          )
33                      })}
34                  </tbody>
35              </table>
36          </div>
37      );
38  }
39
40  export default ListStationsComponent;
```

Pour ce deuxième code, j'ai décidé de vous montrer le component ListStation qui n'est à mon sens pas assez développé. Tout simplement car il affiche juste la liste des stations qui sont affichées sur la map. Alors qu'on pourrait par exemple ajouter un onClick sur le component StationComponent (Card) qui en cliquant dessus change l'icône sur la map afin de savoir où la station que l'on a cliqué se situe.