Identifiant github

Nathanha

Tâches effectuées

Développement de la carte (temps de développement ~10h)

Cette carte a été développée dans un composant permettant son utilisation à plusieurs endroits dans l'application. En effet, elle est utilisée sur la page principale pour afficher un ensemble de stations, mais elle est également utilisée sur le détail d'une station pour afficher sa position.
Cette carte utilise la librairie LeafletJs, qui est opensource et permet un nombre de fonctionnalités importantes. Il existe sur le marché d'autres librairies comme GoogleMap, qui est payante, ou encore Mapbox. J'ai choisi d'utiliser LeafletJs, car c'est une librairie que j'ai déjà utilisée de nombreuses fois durant mon alternance.
Durant ce développement, la plus grosse difficulté a été l'optimisation du temps d'affichage des stations. En effet, le rendu de beaucoup de stations, avec leur pop-up, sur la carte était assez long. Pour remédier à cela, j'ai optimisé les requêtes envoyées au back pour ne récupérer que les coordonnées GPS et l'identifiant de chaque station. Cela a permis d'avoir un chargement presque instantané sur la carte. Pour afficher les informations détaillées d'une station, elles sont chargées lorsque la pop-up d'une station apparaît.

Développement de la liste (temps de développement ~8h)

Cette liste est une alternative d'affichage des stations. Elle ajoute cependant des fonctionnalités non présentes sur la carte. Il est par exemple possible d'ordonner cette liste par prix croissant ou décroissant d'un carburant, ou encore par la distance des stations par rapport à l'utilisateur. Pour cette liste, j'ai utilisé la librairie MUI qui fournit déjà un composant gérant une liste ordonnable. Il existe d'autres librairies permettant cela comme react-data-grid, mais c'est MUI que j'ai choisi, car elle reprend le design material design et est assez simple d'utilisation.
Les mêmes problèmes de performances d'affichage ont été retrouvés avec cette liste. Pour les régler, je récupère toutes les stations filtrées, avec seulement leurs coordonnées GPS et leur identifiant. Le détail de chaque station est ensuite chargé lors du changement de page sur la liste. Cela permet d'avoir de très bonnes performances et d'avoir une liste fonctionnant sans problèmes.

Page de recherche (temps de développement ~3h)

Après avoir développé le composant affichant la carte, le composant affichant la liste et le composant permettant de rentrer les filtres de recherche, il a fallu tous les regrouper sur une même page. Ce développement a été réalisé par Nicolas et moi, étant donné qu'il s'est occupé du développement du composant permettant de choisir des filtres de recherche.

Page d'accueil (temps de développement ~2h)

Cette page est la première page affichée sur l'application. C'est une page que nous avons décidé d'ajouter après plusieurs jours de développement. Au départ, quand l'utilisateur arrivait sur notre application, s'il n'avait pas sa location activée, nous affichions toutes les stations de France sur la carte. Le temps de chargement pour cela était très élevé. Pour remédier à cela, j'ai ajouté cette page qui va permettre à l'utilisateur de saisir plusieurs filtres avant d'être redirigé sur la page contenant la carte et la liste.

Implémentation de Redux (temps de développement ~3h)

Après avoir développé la page d'accueil et la page de recherche, il a fallu relier les filtres sélectionnés par l'utilisateur avec la carte et la liste pour pouvoir filtrer et afficher les stations. Pour cela, nous avons décidé d'utiliser Redux qui permet d'avoir un state global. Nous stockons dans ce state les différents filtres de recherche, pour que chaque composant puisse les utiliser.
Il existe d'autres alternatives à Redux comme Recoil, mais nous l'avons choisi, car il est simple d'implémentation grâce à la librairie reduxjs/toolkit que nous avons utilisée. J'avais déjà utilisé brièvement cette librairie durant mon alternance, et je voulais approfondir ces connaissances. Nous avons fait ce développement Nicolas et moi pour découvrir tous les deux les fonctionnalités apportées par cette librairie.
Pour améliorer l'utilisation de Redux, j'ai ajouté le stockage de ce state global dans le localstorage de React. Cela permet de garder les filtres de recherches si l'utilisateur rafraîchi sa page par exemple.

Formulaire de connexion (temps de développement ~1h30)

J'ai développé le design de ce formulaire pour que l'utilisateur puisse se connecter à notre application et avoir accès à des fonctionnalités supplémentaires. La liaison avec le back a été réalisée par Damien.
Pour le design, j'ai utilisé la librairie MUI, déjà utilisée pour afficher la liste des stations. Cela a permis de garder un design cohérent sur toute l'application. Ce développement aurait pu être réalisé en HTML natif, ou encore avec une autre librairie comme Formik.

AppBar (temps de développement ~1h)

C'est la barre affichée en permanence en haut de l'application. Elle permet à un utilisateur de se connecter, d'ajouter un plein d'essence ou encore de changer le thème de l'application. Elle a été réalisée avec la librairie MUI pour garder le même style dans toute notre application. Étant un composant simple, ce développement aurait pu être fait facilement en HTML, CSS natif.

Stratégie employée pour la gestion des versions avec Git

L'équipe avec laquelle j'ai travaillé est une équipe avec laquelle je travaille depuis la troisième année à Polytech. Nous avons donc l'habitude de travailler ensemble et d'utiliser des outils de versioning comme Git.

Durant ce projet, nous avons utilisé le système d'issues disponible sur Github pour nous séparer les tâches. Nous avons défini ces issues au début du projet lorsque nous avons décidé quelles fonctionnalités nous allions implémenter. Chaque issue représente donc une fonctionnalité et contient un commentaire permettant de donner plus d'indications sur cette dernière et éventuellement définir plusieurs tâches internes à cette fonctionnalité. Chaque issue était attribuée à un ou plusieurs membres de l'équipe en fonction de sa difficulté.

Pour travailler en autonomie sans problèmes, nous avons créé plusieurs branches. Chaque branche était généralement utilisée pour une fonctionnalité précise.

Durant le développement, il est arrivé que nous ajoutions de nouvelles issues auxquelles nous n'avions pas pensé au moment du démarrage du projet.

Fonction optimale

```
public getAdresses(keyword : String, limit: number = 10): Promise<Adress[]>{

    const promise : Promise<Adress[]> = new Promise<Adress[]>((resolve, reject)=>{

        axios.get('https://api-adresse.data.gouv.fr/search/?q='+keyword+'&limit='+limit).then((response)=>{
            const result = response.data.features;
            let adresses : Adress[] = [];
            result.forEach((adress:any) => {
                const temp = new Adress(adress.geometry.coordinates[1], adress.geometry.coordinates[0], adress.properties.label);
                adresses.push(temp);
            });
            resolve(adresses);
        }).catch((error)=>{
            reject(error);
        })

    })

    return promise;
}
```

Comme le nom de cette fonction l'indique, elle permet de récupérer des adresses. Pour cela, elle prend un mot-clé en paramètre et une limite de réponses à renvoyer. Elle va ensuite récupérer un ensemble

d'adresses correspondantes au mot-clé passé en paramètres auprès d'une API du gouvernement. Ce qui est intéressant est qu'elle renvoie un ensemble d'adresses avec le format de l'application. Elle joue donc également le rôle d'adaptateur. Elle permet donc d'abstraire l'appel à l'API du gouvernement aux composants qui vont s'en servir.

Fonction à améliorer

```
public getStationPopup(station: Station) {
    let daySchedule = undefined;
    if (station.schedules) {
        daySchedule = this.mapService.getCurrentDateSchedule(station.schedules);
    }
    return (
        <div className="infobulle" style={{ height: '320px'}}>
            {(station.address) ? <p onClick={this.navigateToStationDetail(station.id)} style={{ textAlign: 'center', fontWeight: 'bold', fontSize: '18px', cursor: 'pointer' }}> {station.address} </p> : ''}
            { (station.schedules) ?
                (daySchedule) ?

                    <p>
                        <span className={this.mapService.isStationOpened(station)?'opened-text':'closed-text'}>
                            {this.mapService.isStationOpened(station)?'Open':'Close'}
                        </span>
                        {this.mapService.getPopupIsOpenText(station)}
                    </p>

                : '' : ''
            }
            { (station.prices) ?
                    station.prices.map((price, i) => {
                        return (<p key={i} ><span className="gas-name"> {price.gas_name} </span> : {price.price} €</p>)
                    })
                : ''
            }
        </div>
    );
}
```

Cette fonction permet de récupérer le code de la pop-up à afficher lors du clic sur un marker représentant une station sur la carte. Je trouve qu'elle est un peu compliquée à lire du fait de sa longueur et du nombre de conditions.

Pour l'améliorer, je couperai sûrement l'affichage de cette pop-up en plusieurs petits composants indépendants. De plus, le style a été mis inline ici. Il faudrait le déplacer dans un fichier CSS avec des classes pour améliorer la visibilité.