Rapport Web - Client Side

LABROUSSE Nicolas

Identifiant Github : NicolasLabrousse

Tâches effectuées :

Au début du projet, nous avons décidé de nous séparer en deux équipes : une s'occupant de la partie client et l'autre de la partie serveur. J'ai décidé de participer à la réalisation du client.

Avant de démarrer, le développement de l'application à proprement parler, nous avons déterminé quelles seraient les fonctionnalités que nous voulions proposer à la démo.
Nous avons ensuite commencé à prendre en main React avec mon équipier.

Voici le détail des composants React que j'ai réalisé dans ce projet, je reviendrais sur chacun d'entre eux pour entrer plus dans le détail :

- SearchView
- Search
- Stations Details
- Line Graph
- Redux

Search View :

Ce composant est le premier que j'ai réalisé en collaboration avec Nathan Haro. C'était notre premier contact avec React et nous avons donc décidé de le créer ensemble pour comprendre comment un composant était construit ainsi que son comportement. Il s'agit de la page principale de l'application dans laquelle on retrouve le "hub" des fonctionnalités présentes.
Search View embarque les composants :

- AppBarCustom
- Search
- Map
- List

La version actuelle du composant ne contient presque plus de logique. Avant l'utilisation de Redux, il avait pour responsabilités de faire le lien entre les différents composants pour faire circuler les informations émanant par exemple de Search et consommer par Map ou List. Avec la mise en place de Redux, nous avons pu simplifier ce composant pour qu'il n'est plus qu'à gérer l'affichage.

Search :

Search est un composant dont la responsabilité est de permettre à l'utilisateur de rechercher des stations essences. Il permet à l'utilisateur de rechercher par adresse grâce à un champ dédié, à filtrer les carburants disponibles dans les stations ainsi que de régler un rayon de recherche autour du point géolocalisé.
La barre de recherche est de type "auto-complete", c'est-à-dire qu'elle propose des résultats à l'utilisateur en quasi temps-réel. Pour ce faire, j'ai utilisé une API gouvernementale permettant de récupérer les adresses correspondant à ce que l'utilisateur a tapé. Pour éviter de faire une requête à chaque lettre entrée par l'utilisateur, j'ai mis en place un système de timeout qui permet d'attendre que l'utilisateur ait fini de taper avant d'effectuer une requête.
Lorsque celui-ci sélectionne une adresse dans les résultats proposés, on vient enregistrer cette nouvelle adresse sélectionnée dans le state global à l'application gérée par Redux.
C'est le même principe qui est utilisé pour le rayon de recherche ainsi que pour les carburants.
Search met donc à disposition dans Redux les choix faits par l'utilisateur qui seront utilisés par d'autres composants dans l'application.

Station Details :

Stations Details est le composant qui permet d'avoir plus de détails sur une station donnée. Il est composé d'une liste contenant le prix des carburants disponible dans la station, une carte affichant sa position ainsi qu'un graphique permettant de visualiser l'historique des prix sur une semaine ainsi qu'un lien vers Google Maps permettant la navigation.
Le composant récupère l'ID de la station à afficher dans l'URL et initie une requête au back end pour récupérer toutes les informations sur celle-ci. Elle initie une requête supplémentaire pour récupérer l'historique des prix, adapter les résultats pour l'affichage puis les afficher dans un graphique.

Line Graph :

Line Graph est un module générique qui permet d'afficher un graphique de type courbe. Il se base sur la bibliothèque Recharts pour se faire. Il permet d'avoir plusieurs courbes au sein du même graphique à partir du moment où les données ont la même structure. Dans notre cas d'usage, il permet d'afficher des prix dans le temps pour différents carburants.
J'ai décidé d'utiliser Recharts car c'est l'une des bibliothèques React dédiées à l'affichage de graphiques les plus utilisés. On trouve une documentation bien faite qui permet de comprendre rapidement son fonctionnement et comment la mettre en œuvre dans son projet. Il en existe d'autres tels que React Chartjs 2 ou React Vis mais elles étaient moins connues et utilisées.

Redux :

Aux environs de la moitié du projet, nous nous sommes rendu compte que nous avions besoin d'un moyen plus simple et générique de transmettre les informations entre nos composants. Le problème provenait du composant Search et des données qu'il générait. Au fur et à mesure que l'application grossissait, de plus en plus de pages et de composants avaient besoin des informations que l'utilisateur avait rentré pour réaliser leur tâche. Sans Redux, nous commencions à avoir beaucoup de remontée d'information dans l'arbre des composants pour les redescendre vers une autre feuille. Cela est vite devenu ingérable.

Nous avons alors décidé d'intégrer Redux à notre application. Celui-ci permet de gérer un état global à l'application où tous les composants peuvent avoir accès. On y stocke tous les filtres sur les carburants qu'un utilisateur à sélectionné.

Il existe des alternatives à Redux tel que Recoil. Notre choix s'est arrêté sur Redux car c'est une bibliothèque encore largement utilisée et il s'avère que je vais être amené à l'utiliser dans le cadre de mon travail. Cela m'a permis de comprendre ses principes dans le cadre d'un projet scolaire.

Difficultés Rencontrées :

Je n'ai pas rencontré de difficulté majeure lors de ce projet. Ayant travaillé 2 ans sur un gros projet Angular lors de mes 2 premières années d'apprentissage, le passage à React s'est fait de façon plutôt simple. Nous avons passé un moment à réfléchir à l'architecture du projet pour répondre à la question : Presque tous les composants ont besoin des filtres sélectionnés par l'utilisateur. Comment rendre disponibles ces informations de la façon la plus simple ?
Nous avons passé un certain temps à comprendre Redux, à le mettre en place et revoir le code pour l'utiliser. Mais ce travail a payé puisque la suite des fonctionnalités a été bien plus simple à implémenter.

Certaines choses, qui ne paraissent pas de prime abord, prennent du temps. Pour avoir un rendu visuel cohérent, il faut souvent itérer de nombreuses fois pour atteindre l'objectif choisi.

J'ai aussi passé du temps à discuter avec l'équipe qui s'occupait du backend pour définir ce dont nous avions besoin, les endpoints ainsi que le format des données.

Stratégie employée pour la gestion des versions avec Git :

Pour chaque fonctionnalité, nous avons créé des issues Github puis nous nous les assignons en fonction de leur priorité et de notre envie de réaliser la tâche en question. Chaque feature faisait l'objet d'une branche spécifique que nous mergions lorsque le développement était terminé.

Commentaire de code :

```
/**
 * Method trigger when user start typing in autocomplete
 * Perform API call to retrieve city proposition
 * @param e
 */
onUserCityInput(e:any){

    this.setState({ city_loading: true });

    if (this.auto_complete_timeout != null) clearTimeout(this.auto_complete_timeout);

    this.auto_complete_timeout = setTimeout(() => {
        this.searchTerms = e.target.value;
        if(this.searchTerms) {
            this.adressesApi.getAdresses(this.searchTerms).then((adresses: Adress[])=>{
                this.setState({adresses : adresses});
                this.setState({ city_loading: false });
            }).catch((e)=>{
                console.log(e);
                this.setState({ city_loading: false });
            });
        } else {
            this.setState({ city_loading: false });
        }
    }, 500);

}
```

Je trouve cette méthode efficace dans ce qu'elle réalise. La première version ne comprenait pas de temporisation lorsqu'un utilisateur tapait une lettre. On effectuait une requête à chaque fois ce qui avait tendance à vite surcharger le réseau et à ralentir l'application avec plusieurs requêtes ouvertes en même temps. L'implémentation d'une temporisation de 500ms, qui est réinitialisée à chaque entrée de l'utilisateur, permet d'effectuer la requête quand celui-ci a fini de taper sa recherche.

```
/**
 * Format the station data to be displayed by the graph widget
 * @param res
 * @returns
 */
private formatStationData(res: GasDataPrice[]): GasDataPrice[] {

    res.forEach(element => {
        for(let i = 0; i<element.data.length; i++){
            if(parseInt(element.data[i].date)!==i){
                element.data.splice(i, 0, {date: i.toString(), price: null});
            }
        }
        element.data.push(element.data[0]);
        element.data.shift();
    });

    // eslint-disable-next-line array-callback-return
    res.map((priceHistory) => {
        priceHistory.data.forEach((element) => {
          switch (parseInt(element.date)) {
            case 0:
              element.date = 'Dimanche';
              break;
            case 1:
              element.date = 'Lundi';
              break;
            case 2:
              element.date = 'Mardi';
              break;
            case 3:
              element.date = 'Mercredi';
              break;
            case 4:
              element.date = 'Jeudi';
              break;
            case 5:
              element.date = 'Vendredi';
              break;
            case 6:
              element.date = 'Samedi';
              break;
            default:
              element.date = 'Inconnu';
          }
        });
      });
    return res;
}
```

Cette méthode permet de mettre en forme les données d'historique des prix des carburants d'une station pour affichage dans le graphique. Bien qu'elle fasse le travail demandé, elle mériterait d'être revu pour plusieurs raisons. La première étant qu'elle permet d'adapter seulement un historique sur une semaine. Si on veut un historique sur plusieurs mois ou années, cette méthode ne le permet pas.
La deuxième est que l'on retrouve "en dur" la traduction vers le nom de la journée. Si on veut gérer plusieurs langues par exemple, cela devient compliqué.