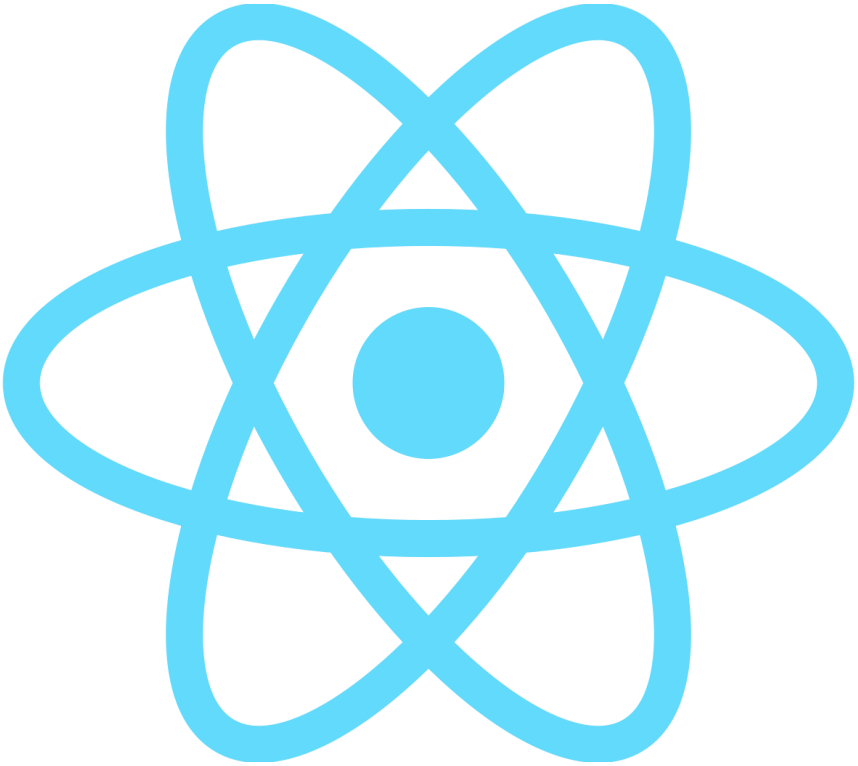
Programmable Web – Rapport individuel

Client Side

Semestre 9 – 2022



MAZURIER Alexandre (SI5-IHM)
Github ID : Alexandre-MAZURIER

Université Côte d'Azur
Polytech Nice Sophia
ProgWeb ClientSide

Table des matières

Tâches effectuées

Ci-dessous les principales tâches effectuées côté client sur ce projet. A cela s'ajoute une session de pair-coding de 3h pour initialiser le projet et mettre en place une première version de la carte, ainsi qu'un travail d'1h permettant d'actualiser les stations sur la carte lorsque l'utilisateur se déplace sur celle-ci. Le lien entre le front est le back se fait à travers notre fichier api.js regroupant les différents appels à l'api. J'ai notamment écrit la fonction permettant de récupérer les stations d'une ville en particulier. Les données sont échangées en format json, ce qui permet de les lire très facilement en JavaScript.

Page d'affichage des stations-service sous forme de tableau (~7h)

Cette page permet à l'utilisateur de consulter toutes les stations à partir d'un nom de ville qu'il entrera dans un champ de texte. Les notions de feedback sur le chargement, avertissement sur le champ non remplir avant envoie de la requête, etc ont été prises en compte. Lorsqu'il initie sa requête, l'utilisateur choisit le carburant qui l'intéresse, ainsi, seules les stations proposant ce carburant à la vente seront affichées dans la table.

Les données issues de la base de données peuvent être filtrées selon différents critères jugés intéressants. Les stations peuvent aussi être triés selon le prix de chaque carburant d'un simple clic sur la colonne souhaitée. Le fonctionnement des filtres est le suivant :

- Choix de la ville : côté serveur
- Choix du carburant : côté client
- Station de lavage : côté client
- Station de gonflage : côté client
- Automate 24/24 : côté client

J'ai décidé d'utiliser le filtre de plus grande envergure côté serveur afin de ne pas charger toutes les stations, ce qui amènerait des lenteurs dû au réseau et au traitement de la masse de données. En revanche, le tri selon les autres critères se fait côté client afin d'offrir un filtre qui paraît instantané, et parce que les données à filtrer sont de taille raisonnable. Aucune lenteur ne sera perceptible pour l'utilisateur peut importe le dispositif sur lequel il consulte la page.

Affichage différencié des icones de station-service en fonction du prix (~3h)

Afin de voir facilement quelle est la station la moins chère sur la carte, l'icone de la station la moins chère est de couleur verte, tandis que les autres sont de la couleur classique. Afin de rendre cette visualisation plus perceptible, j'ai aussi offert la possibilité à l'utilisateur d'activer/désactiver le mode clusterisation. En effet, si la station la moins chère de la zone est dans un cluster, elle n'aurait pas été visible. Pour activer/désactiver le mode clusturisation proprement, j'ai créé un composant *ConditionnalWrapper* qui permet d'englober ou non du code JSX avec une balise parente, en fonction d'une condition. Ainsi, lorsque l'utilisateur souhaite afficher des clusters de stations, les marqueurs sont englobés par *MarkerClusterGroup,* alors que dans l'autre cas, ils ne seront pas enveloppé dans un tag parent.

Stratégie employée pour la gestion des versions avec Git

Notre git est découpé en deux branches principales : *main* et *develop*. La branche *main* correspondrait à une version de production, c'est-à-dire une version stable, que l'on pourrait checkout notamment si l'on avait à faire une démonstration du projet à un tout instant alors que du développement est en cours. De son côté, la branche *develop* est la branche sur laquelle nous développons et envoyons les commits. Lorsque l'on implémentait une fonctionnalité complète conséquente, nous quittions la branche develop pour créer une branche ayant le nom de la fonctionnalité que nous développions. Dans mon cas, j'ai créé la branche *list-view*, que j'ai par la suite mergé sur develop.

Afin de se répartir les tâches et définir les contours du projet, nous avons aussi créé des issues, en assignant les personnes à ces dernières et en les catégorisant entre *front, back*, *feature, bug* …

Solutions choisies

Pour ce qui est de l'affichage des stations essences sur une carte, nous avons choisi au début du projet d'utiliser la librairie React Leaflet. En effet, cette librairie permet d'intégrer simplement une carte, des marqueurs, et de la clusterisation, sans avoir besoin de clés d'API, les données utilisées étant issues d'OpenStreetMap.

Pour l'affichage en liste des stations, j'ai décidé d'utiliser une librairie de DataTable. Je pensais dans un premier temps utiliser le composant Table de la librairie Mantine déjà ajouté au projet, mais elle n'offrait pas suffisamment de fonctionnalités. Je cherchais une librairie qui permettait de simplement trier les données, comportent une pagination, et d'un autre côté personnalisable. La librairie *react-data-table-component* répondait à tous ces critères. Sa documentation en ligne est globalement assez fournie, cependant, il peut être difficile de trouver une réponse à une question sur StackOverflow, si je compare à des librairies complètes comme Angular Material ou PrimeNg pour Angular que j'utilise dans d'autres projets.

Enfin, pour afficher des icones explicites dans la colonne *services* de l'affichage des stations en liste, j'ai utilisé la librairie React icons. Elle propose une large gamme d'icônes,là où *radix-icon* qui était déjà ajouté au projet ne proposait que les icones basiques.

Difficultés rencontrées

L'apprentissage de React s'est globalement bien déroulé, le TD iTunes ayant permis de découvrir pas mal d'aspects de React. De plus, mes connaissances sur d'autres Framework JavaScript comme Angular (avec lequel je développe une application lors de mon alternance), et VueJS, m'ont permis d'apprendre plus vite le fonctionnement général de React, n'étant pas bloqué sur les aspects du JavaScript, etc. Les difficultés que j'ai rencontrées sont principalement :

- L'utilisation du JavaScript au lieu du TypeScript. En effet, je trouve que le typescript permet de développer globalement, plus rapidement, avec une auto-complétion plus fiable, et beaucoup moins d'erreurs lors de l'exécution. Par exemple, le modèle de données que le serveur nous envoyait avait ses attributs en français, mais le modèle d'objet n'étant pas défini en JS, je faisais souvent des $station.name$ au lieu de $station.nom$, sans qu'il y ait d'erreur

dans l'éditeur, ce qui ne me serait pas arrivé avec du TypeScript. Nous n'avons su que trop tard que le TypeScript n'était pas interdit, mais uniquement déconseillé aux personnes n'en ayant jamais fait, le cours portant sur JavaScript.

- Bien que répété lors des cours, le fait que les changements d'états soient asynchrones m'a fait perdre pas mal de temps lors du développement. Il m'arrivait d'être face à des comportements aléatoires, sans comprendre ce qu'il se passait alors qu'il n'y avait aucune erreur dans la console. En réalité, c'était juste que j'utiulisais une valeur d'état juste après l'avoir modifié et que certaines fois, la valeur avait eu le temps de se modifier, alors que d'autres fois, non. J'ai perdu suffisamment de temps sur cette difficulté pour la re-rencontrer dans le futur.

Code

Lorsque j'ai implémenté la notion de filtres sur les services proposés par les stations côté front, j'ai écrit une fonction qui ne me semblait pas optimale (à droite). Ce filtre consiste à parcourir les stations et garder uniquement celles qui proposent le service coché par l'utilisateur. Lorsque plusieurs services sont cochés, on considère que c'est un « et logique » entre les éléments et que l'on souhaite afficher les stations comportant tous les services sélectionnés.

Le problème dans le code initialement écrit à droite est un problème de réutilisation de code, d'une part, mais pas seulement. L'effort pour ajouter un nouveau filtre est bien supérieur à la version corrigée juste en dessous. En plus d'éviter la dupplication, j'ai amélioré la lisibilité de la méthode en utilisant la méthode *find(),* plutôt que de boucler sur les services.

```
const filterStations = (filters) => {
  let stations = [...allStations];
  for (let i in filters) {
    if (filters[i]) {
      stations = stations.filter((s) => {
        return s.services.find(s => s.toLowerCase().includes(i))
      });
    }
  }
  setFilteredStations(stations);
};
```

Figure 2. Version corrigée du code

```
function filterStations(filters) {
    let stations = [...allStations];
    if(filters.cb) {
        stations = stations.filter(s => {
            let res = false
            s.services.forEach(service => {
                if(service.toLowerCase().includes("24/24")) {
                    res = true;
                }
            })
            return res;
     })
    }
    if(filters.gonflage) {
        stations = stations.filter(s => {
            let res = false
            s.services.forEach(service => {
                if(service.toLowerCase().includes("gonflage")) {
                    res = true;
                }
            })
            return res;
        })
    }
    if(filters.lavage) {
        stations = stations.filter(s => {
            let res = false
            s.services.forEach(service => {
                if(service.toLowerCase().includes("lavage")) {
                    res = true;
                }
            })
            return res;
        })
    }
    setFilteredStations(stations);
```

Figure 1. Ancienne version du code