# Rapport Programmation Web : Client Side

Léo BURETTE - SI5-Parcours AL

GitHub : LeoBurette
`https://github.com/LeoBurette`

## Tâches effectuées

Au cours de ce projet, j'ai effectué les tâches suivantes :

— Mise en place de la carte *OpenStreetMap* à l'aide des librairies *Leaflet* et *react-leaflet*. La localisation est demandée lors du premier chargement de la page pour placer un marqueur sur la carte à l'emplacement de l'utilisateur.

— Mise en place de la gestion des thèmes avec l'utilisation du package "*use-local-storage*" pour pouvoir se souvenir du choix de l'utilisateur et de "*match-media*" pour prendre la valeur de préférence de l'utilisateur par défaut.

— D'autre taches mineurs de composants responsive ou bien de composants qui change en cas d'input (comme le prix qui se grise dans les filtres de sélection).

## Stratégie employée pour la gestion des versions avec Git

Pour la gestion des versions Git, nous avons opté pour une solution classique de branching Stratégie. Chaque issue correspond à une *feature*, chaque *feature* est une branche qui part de la dernière version sur *develop*. De cette manière, toute les features peuvent être développé indépendamment. De plus, quand l'on a fini sur une branche, on fait une *pull request* pour pouvoir merger avec develop. Cela permet de garder la trace de toutes les modifications faites sur une branche et de faire valider les changements par l'équipe.

## Solutions choisies

Au niveau des choix de technologies sur les tâches qui m'ont été attribuées :

— Le choix de *OpenStreetMap* (abrégé en OSM) s'est fait dès le début de par son aspect ouvert, communautaire et gratuit. Elle fait référence dans le domaine ce qui nous assure une documentation de qualité et une maintenabilité élevée. À l'inverse de *Google maps* ou bien de *Mapbox*, OSM ne nécessite ni compte ni API token. Ce choix est donc parfait pour notre application, les fonctionnalités nécessaires étant uniquement l'affichage de la carte et le placement de marqueurs. *OpenLayers* et *MapLibreGL* furent aussi envisagé, mais notre expérience avec OSM a été au final décisive dans ce choix.
La librairie *Leaflet* est utilisé pour la carte. Cette librairie permet une intégration et une utilisation de la carte de manière très simple, surtout couplé avec *react-leaflet* qui intègre tout cela dans un environnement *React*.
Enfin, la librairie *react-leaflet-markercluster* permet de créer des cluster de marqueurs sur

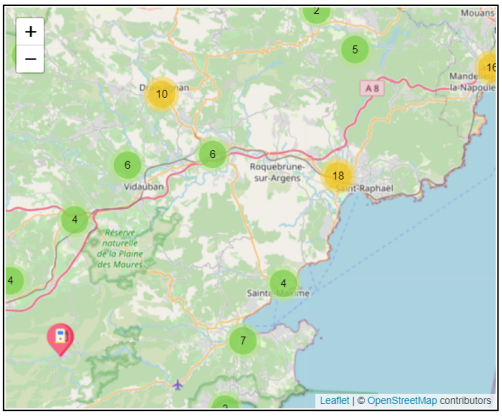



FIGURE 1 – Carte avec des marqueurs et des clusters


FIGURE 2 – Différence entre thème *light* et *dark*

la carte si trop sont présent. Essentiel pour la lisibilité de la carte lorsque de nombreux marqueurs sont présent, cette fonctionnalité n'est pourtant pas présente nativement dans *Leaflet*.

— Pour le thème, nous utilisons le package *use-local-storage* qui nous permet de stoker des informations dans le navigateur en React. Ce système de clef valeur stock la préférence de thème de l'utilisateur.

## Difficultés rencontrées

L'intégration d'OSM dans *React* fût compliqué au début. Beaucoup de contraintes sont présentes avec cette carte lors de son intégration, les deux principales sont :

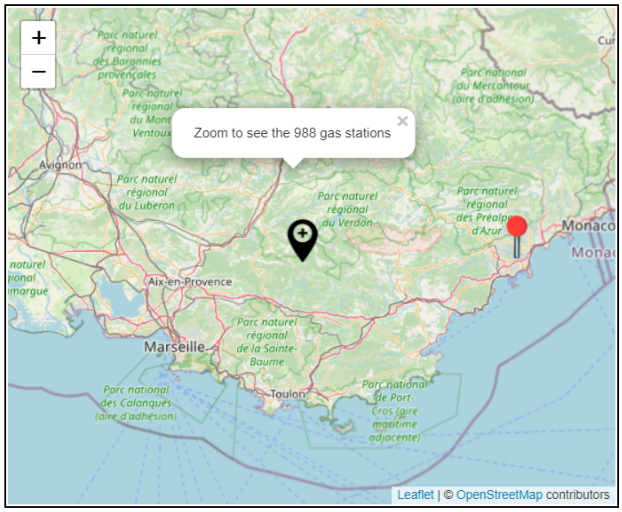
— La présence d'un fichier CSS nécessaire à l'affichage de la carte.

— Le fait que le container de la carte doive avec une taille fixe (pas de pourcentage).

### Chargement des données

Une fois ces deux problèmes gérés, le chargement de notre pool de données était à questionner. Comment charger les données avec notre carte, qu'affichons nous si l'on de-zoom au maximum ?

Pour la récupération des données, on appelle le backend avec le point central de la carte et la distance entre le point sud-ouest et nord-est. De cette manière, on charge les points dans la zone visible de la carte. Beaucoup d'appels sont faits au backend, mais l'application est légère et réactive. Nous aurions pu charger tous les points d'un coup et les afficher, mais le temps de chargement était trop long. Plusieurs secondes se




2 FIGURE 3 – Centre de la carte avec le nombre de points

déroulaient entre le chargement des données et l'apparition des marqueurs, plusieurs secondes pendant lesquelles la carte demeurée indisponible.

Néanmoins, quand la carte affiche l'ensemble de la France, toutes les données sont renvoyées par le backend ? Ici, la solution idéale aurait été de, à partir d'un certain niveau de dé-zoom, demander au backend uniquement combien d'éléments sont présent sans qu'il ne nous renvoie la liste et d'afficher un marqueur unique avec le nombre d'éléments. Nous avons gardé cette idée sans modifier le backend en regardant le nombre d'élément dans la liste : si cette dernière dépasse une limite (fixée à 1000 dans notre cas), on ne charge qu'un marqueur au centre de la carte avec un texte indiquant combien de points s'affichent quand on zoomera.

### Thème

Le thème n'a pas posé de problème particulier lors de l'intégration.

## Temps de développement / tâche

Nous ne nous sommes pas chronométré donc les temps seront relatifs, mais de mémoire :
— Thèmes : 2-H
— Carte : Plus d'une dizaine d'heures
— Compréhension et adaptation à React : plusieurs heures

La quasi-totalité de mon temps a été pris par la création de la carte, la gestion de ses erreurs, la localisation et l'affichage des marqueurs. De plus, l'environnement React m'étant inconnu, il était très compliqué de faire fonctionner ces deux inconnues en parallèle.

## Code

### Code élégant

Pour moi, le système de gestion de thème est optimal et élégant :

```
1  import React from "react";
2  import './App.css';
3  import RouterFunction from "./components/routing/Router";
4  import {FilterProvider} from "./contexts/FilterContext";
5  import useLocalStorage from 'use-local-storage';
6
7  function App() {
8      const defaultDark = window.matchMedia('(prefers-color-scheme: dark)').matches;
9      const [theme, setTheme] = useLocalStorage('theme', defaultDark ? 'dark' : '
           light');
10
11     const switchTheme = ()=>{
12         const newTheme = theme === 'light' ? 'dark' : 'light';
13         setTheme(newTheme);
14     }
15
16     return (
17     <>
18         <div className="App" data-theme={theme}>
19             <FilterProvider>
20             <RouterFunction theme={theme} switchTheme={switchTheme}></
                   RouterFunction>
21             </FilterProvider>
22         </div>
```

```
23      </>
24      );
25  }
```

Listing 1 – Système de gestion de thème

Quand l'on charge l'app, la première instruction ligne 8 permet de déterminer si un thème de préférence est choisi par l'utilisateur. Ligne 9, nous utilisons le LocalStorage pour stocker le thème en sombre si l'utilisateur préfère le thème sombre, en blanc autrement.

Enfin, nous définissons la méthode *switchTheme* qui permet de changer le thème.
Nous englobons notre application dans un div app en lui donnant une propriété *data-theme* qui permet de savoir quel thème est utilisé.

Le *RouterFunction* à deux props *theme* et *switchTheme* car c'est la *navbar* qui possède le bouton permettant de switch de thème. Nous aurions pu créer un contexte pour cela plutôt que de faire descendre les informations sur "deux générations".

```
1  :root {
2    --background: white;
3    --background-secondary: rgb(159, 81, 211);
4    --text-primary: black;
5    --text-secondary: rgb(11, 19, 41);
6    --accent: white;
7  }
8  [data-theme='dark'] {
9    --background: black;
10   --background-secondary: rgb(37, 37, 37);
11   --text-primary: white;
12   --text-secondary: rgb(182, 0, 0);
13   --accent: darkred;
14 }
```

Listing 2 – Feuille de style du index

Dans la partie CSS, on défini les variables de couleur qui seront utilisées dans l'application, et on les surcharge dans le cas où le data-theme serait égal à *dark*. Il suffit alors d'utiliser les variables définies dans le CSS pour changer la couleur de fond, de texte, de fond des boutons, etc.

### Code à améliorer

Au niveau de la carte, nous avons beaucoup de points à améliorer :

- Le marqueur indiquant le nombre de points quand ils dépassent la limite met du temps à s'actualiser.
- Comme indiqué plus haut, nous pourrions modifier le backend pour qu'il renvoie uniquement le nombre d'éléments présent dans certains cas.
- Nous utilisons un système de *callback* pour les requêtes, ce qui n'est pas forcément pertinent dans notre utilisation actuelle.
- Le location marker créer une fuite de mémoire si l'on change de page trop vite avant son apparition. Ce bug est connus, reproductible, mais nous n'avons pas la connaissance nécessaire en React et Leaflet pour le corriger. Nous aurions dû enquêter plus dessus si nous avions eu plus de temps.