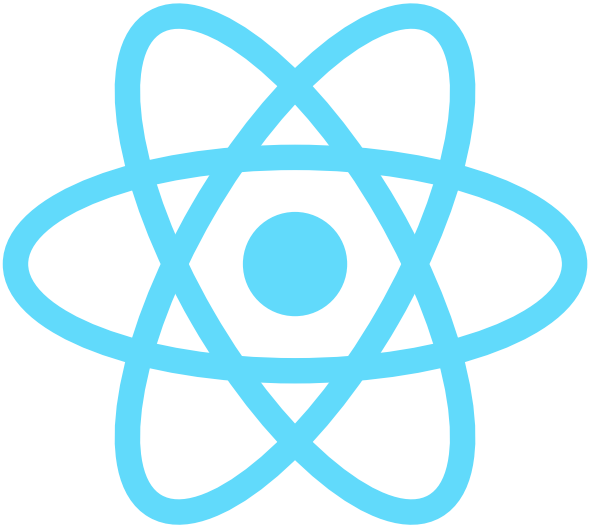
Rapport
Programmable Web - Client Side

Lucas RAKOTOMALALA
lucas.rakotomalala@etu.univ-cotedazur.fr

27 février 2022



Enseignants
Hugo MALLET, François MICHAUDON

POLYTECH NICE-SOPHIA

UNIVERSITÉ CÔTE D'AZUR

Table des matières

1 Informations générales

1.1 Lien GitHub du projet

Le projet est disponible ici. Mon compte GitHub est quant à lui disponible ici.

Aussi, vous pouvez retrouver le frontend ainsi que le backend déployés sur Heroku ici. Le frontend ainsi que le backend sont déployés ensemble, sous la forme d'un conteneur.

1.2 Gestion des versions

Afin de proposer une version stable à n'importe quel moment du développement, nous avons fonctionné selon un modèle de *branching* similaire à Git Flow : nous avons une branche principale (*main*) qui nous permet délivrer des releases, une branche de développement (*develop*) sur laquelle nous ajoutons les fonctionnalités développées généralement sur une branche à part, dont le nom est communément préfixée par *feature*. Cela s'effectue grâce à un *merge* pour ma part.

Concernant les fonctionnalités à développer, nous avons créé des issues sur GitHub pour chacune d'elle et nous l'avons assignée à une ou plusieurs personnes de manière à ce que tout le monde puisse participer au développement du projet, et notamment du frontend.

2 Tâches effectuées

Durant ce projet, j'ai eu l'occasion de réaliser plusieurs tâches :

- Refactor de l'application [1-2 jours]
- *Responsiveness* du menu de navigation [3 heures]
- Choix de la librairie CSS et refonte du style globale de l'application [5 heures]
- Mise en place du router pour obtenir une **Single Page Application** (SPA) [1 heure]
- Mise en place de **ESLint** et **Prettier** [2 heures]
- Déploiement de l'application sur **Heroku** [4 heures]

Le temps de développement de chaque tâche est spécifié entre crochets.

3 Retours sur le développement

3.1 Choix des librairies

3.1.1 Style de l'application

Nous avons jugé qu'il était inutile de réinventer la roue et c'est la raison pour laquelle nous utilisons une librairie CSS, à savoir **Mantine v3.6.11**.

Bien que cette librairie CSS soit récente (première release en Mars 2021), elle propose de nombreux composants React. De plus, elle met à disposition un certain nombre de *hooks*,

comme celui pour les notifications. C'est un avantage considérable lorsqu'on compare avec d'autres libraires CSS, comme `react-bootstrap` par exemple. Elle prend en charge aussi la *responsiveness* de l'application grâce à des composants comme `MediaQuery` ou par le biais d'un hook spécifique : *useMediaQuery()*. La libraire Mantine permet également de gérer un thème sombre et des notifications de manière globale grâce aux différents contextes qu'elle intègre.

Il existe un ribambelle de librairies CSS adaptées à React (`react-bootstrap`, `antd` ou encore `@mui/material`) mais nous voulions surtout avoir une librairie dont le style nous plaisait.

3.1.2 Router

Pour obtenir une SPA, nous avons utilisé `react-router-dom` dans sa version **6**. Son utilisation est simple est bien documentée.

Le choix de cette librairie s'est imposé de lui-même : il s'agit de la seule librairie présentée sur la première page lorsqu'on recherche un moyen de faire des SPA avec React.

3.2 Difficultés rencontrées

Nous avions une première version dans laquelle tout était contenu dans le fichier `App.js`, sans définir de composants. Le *refactor* de l'ensemble de l'application a été long et fastidieux mais était nécessaire lorsque nous avons découvert le sujet. Durant ce dernier, nous nous sommes posés la question de l'utilisation de *contexte* ou de *props* concernant les différents formulaires de l'application. Finalement, nous avons décidé d'utiliser les *props*, pour ne pas changer davantage la structure de notre application mais aussi car nous étions plus à l'aise avec les *props* qu'avec les *contextes*. Si nous avions eu plus de temps, nous aurions effectué un nouveau refactor pour utiliser les *contextes*, notamment pour la position de l'utilisateur.

Ce refactor est un véritable atout pour notre application puisqu'un changement d'une variable d'état générera un nouveau rendu uniquement pour les composants l'ayant en *props* et non toute l'application comme c'était le cas avant. On gagne ainsi énormément en performance.

Étant donné que Mantine est un projet encore jeune, il y a peu d'exemples de son utilisation hormis la documentation. J'ai donc passé beaucoup de temps à investiguer et tester pour avoir le rendu actuel.

Le déploiement de l'application sur Heroku était compliqué. En effet, cela a nécessité quelques changements dans l'application, aussi bien le frontend que le backend puisque je voulais pouvoir déployer l'ensemble dans un seul et même conteneur pour faciliter l'utilisation de notre application. J'ai dû changer des variables d'environnements mais aussi me documenter sur les *layers* des conteneurs pour obtenir une image aussi petite que possible. De plus, je devais suivre certaines *guidelines* d'Heroku, sans quoi l'application ne fonctionnait pas. Par

exemple, il a fallu exposer les ports de l'application avec la variable d'environnement `PORT` et non `HTTP_PORT`.

L'affichage de la map de Leaflet a été dure à gérer, bien que simplifiée avec les composants proposés par la librairie `react-leaflet`. En effet, il est nécessaire de spécifier une hauteur sans quoi la map ne s'affiche pas, mais dans le cas où on spécifie une hauteur (`100vh` pour qu'elle prenne tout l'écran), cette dernière s'affiche par dessus le menu de navigation uniquement lors de l'utilisation de l'application sur mobile. Pour régler ce problème, j'ai créé un style spécifique pour le menu de navigation dans lequel je spécifie la taille (en pixels) de ce dernier lorsque la taille de l'écran est inférieure ou égale au *breakpoint md* (768 pixels).

4 Code

Les composants se trouvent dans le dossier `src/components` du dossier `client`. Dans le cas du composant `Map`, il est dans un sous-dossier où se trouve également son fichier de style.

4.1 Composant élégant / optimal

Le composant `MainLinks.js` est celui que je trouve le plus élégant. En effet, ce dernier est séparé en plusieurs sous-composants internes à celui-ci.

Le premier sous-composant (`CustomLink`) est tiré de la documentation de `react-router-dom` et permet de changer le style du bouton si la route qui lui est associée est active. Cela permet à l'utilisateur de savoir d'un rapide coup d'oeil sur quelle page de l'application il se trouve.

Le second sous-composant, à savoir `CustomButton`, regroupe quant à lui deux autres sous-composants :

- `ButtonIcon` qui permet d'afficher un bouton qui change de couleur suivant le thème choisi.
- `ResponsiveText` qui permet de ne pas afficher le texte du bouton lorsque l'écran est trop petit.

L'avantage d'un tel découpage est la maintenabilité du composant. En effet, si on souhaite changer la couleur du fond des images lorsque le thème est sombre, il suffit de modifier une seule ligne pour appliquer le changement à tous les boutons.

De plus, le composant s'occupe d'un seul métier : afficher les boutons correspondants aux routes disponibles sur notre application.

Par souci de lisibilité, je vous invite à consulter Github pour voir la partie concernée.

4.2 Méthode à optimiser

Je pense que le `useEffect` contenu dans le composant `Map` pourrait être mieux écrit. Dans ce *hook*, on appelle l'API pour récupérer les stations autour de l'utilisateur. Afin de notifier l'utilisateur de l'avancée de la requête, on crée une notification qui s'affiche sur son écran. Or, le `useEffect` est appelé dès lors que `position`, `distance` ou `gas` change de valeur. Une nouvelle notification est donc créé pour chaque requête lancée. Cependant, le comportement souhaité est que si la première requête n'a pas eu le temps de se finaliser, on aimerait écrire par dessus la notification associée pour ne pas surcharger l'écran de l'utilisateur. Ce n'est pas actuellement le cas puisque la notification est créé à l'intérieur du `useEffect`.

Par souci de lisibilité, je vous invite à consulter Github pour voir la partie concernée.