Rapport

Table des matières

Identifiant GitHub

Figure 1 - Image de profile

JeromeFroment

https://github.com/JeromeFroment

Stratégies de versionnage

Pour le système de versionnage de notre application, nous avons utilisé Github. 2 branches communes ont été créées : Main – Develop. La branche Main est celle qui garde une version stable du projet et est celle qui sera utilisée pour un éventuel déploiement. Develop est la branche où l'on merge les branches de développement de chacun. Chacune de ces branches sont créées à chaque nouvelle tâche. Elles sont généralement désignées pour une personne et pour une tâche en particulier (nommage : feature/nom_tache). Une fois la tâche réalisée, une Pull Request est créée afin de la faire valider par les autres membres de l'équipe et ensuite être mergée sur la branche Develop. Une fois le merge réalisé, la branche est supprimée.

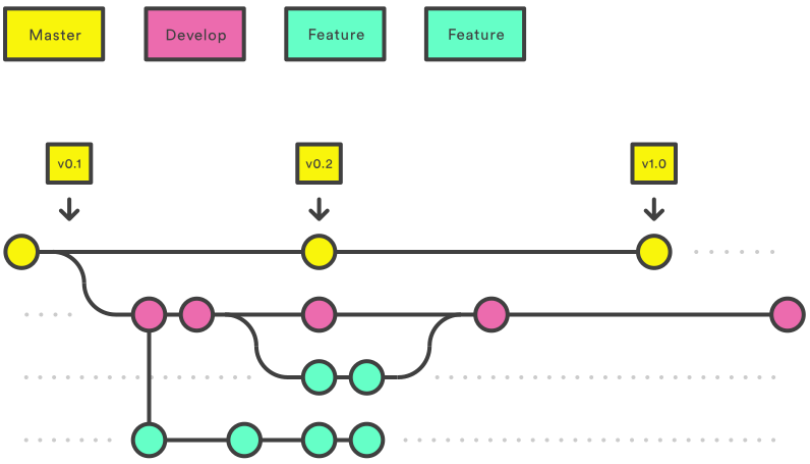


Figure 2 - Système de branches et de versionning

Tâches effectuées

Durant ce projet, mes tâches ont été multiples et diverses :

Router – Navbar

Durée : 1 jour

Pour la navigation dans notre application, j'ai réalisé un Router et une Navbar de sorte que le projet respecte bien le concept de Single Page App. Le retour Router, qui provient de la librairie react-router, permet de spécifier des routes vers nos composants selon l'url définie par les choix d'utilisateur. C'est à ce moment là que la Navbar intervient. Elle permet à l'utilisateur de choisir sur quelle page il veut se diriger, la navbar paramètre alors l'url et le router lorsqu'il détecte ce changement fait naviguer vers le composant associé au path renseigné dans l'url.

```
<NavLink to='/stateMap'>
  Statistiques
</NavLink>
```

```
<Route path="/stateMap"
  element={<StatisticsAccess />} />
```

Figure 3 – Extrait de la Navbar et du Router

Page Statistiques

Durée : 1 semaine

Cette page a été créée dans le but de pouvoir visualiser les statistiques régionales et départementales. Pour cela, j'ai choisi d'utiliser la librairie **D3JS**[1] afin d'afficher la carte des départements et régions de la France. Pour éviter une surcharge d'information sur une seule page, nous avons tout d'abord la carte des région (Figure 2), sur laquelle on peut soit survoler une région, soit en sélectionner une. Dans le cas d'un survol, des données apparaissent sous forme de tableau pour indiquer la moyenne des prix par type d'essence pour la région. Si l'on clique sur la région, on descend en granularité et une carte de la région et de ses départements remplace l'ancienne (Figure 3). Ici, mêmes fonctionnalités, l'utilisateur peut visualiser les données des différents types de carburant.

Un bouton « Localisez-moi » a aussi été implémenté afin que l'utilisateur puisse se situer sur la carte (Figure 4).

Figure 4 - Carte des régions de France

Figure 5 - Carte des départements de la région Auvergne - Rhône-Alpes

[1] D3JS : https://www.datavis.fr/index.php?page=map-firststep

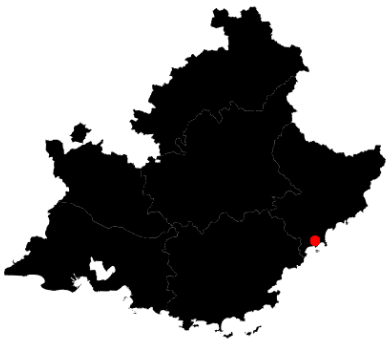


Figure 6 - Point de géolocalisation (rouge)

Le choix de cette librairie a été pris pour son élégance visuelle ainsi que le cas d'utilisation intéressant avec ses différents types de carte (Nationale, Régionale).

La récupération des données statistiques se fait au travers de **deux services** (depStatistics.service.jsx et regionStatistics.service.jsx), qui respectivement requêtent les statistiques des départements d'une région donnée et les statistiques des régions de France.

Pas de réels points de blocage mais néanmoins des difficultés à implémenter la carte. Il a fallu prendre en main cette nouvelle librairie et savoir se documenter efficacement.

La requête pour récupérer les données est insérée dans la fonction **useEffect**. Pour mettre à jour les données de statistiques selon l'emplacement de la souris, est géré par le Hook **useState**, qui renvoie une valeur d'état local et une fonction pour la mettre à jour.

Page Login

Durée 3 jours

La page Login est une simple page de login et d'inscription où l'utilisateur doit soit inscrire ses identifiants pour pouvoir accéder à l'application soit s'inscrire pour créer un compte. Lors de la validation du formulaire d'inscription, l'utilisateur doit y renseigner son nom, prénom, mail (qui sera son ID) et mot de passe. Une fois le bouton pressé, une requête sera envoyée au serveur avec en body les informations de l'utilisateur et va créer un nouveau compte si les informations sont bonnes.

Le second formulaire est le formulaire de login. Une requête sera envoyé au Back-end afin de confirmer l'identité de l'utilisateur. Dans le cas où les identifiants sont bons, le serveur renvoie un **Token JWT** qui sera valable jusqu'à un refresh de la page. Au contraire, si les

identifiants sont mauvais la réponse sera vide et donc l'utilisateur n'aura pas les droits de rentrer dans l'application.

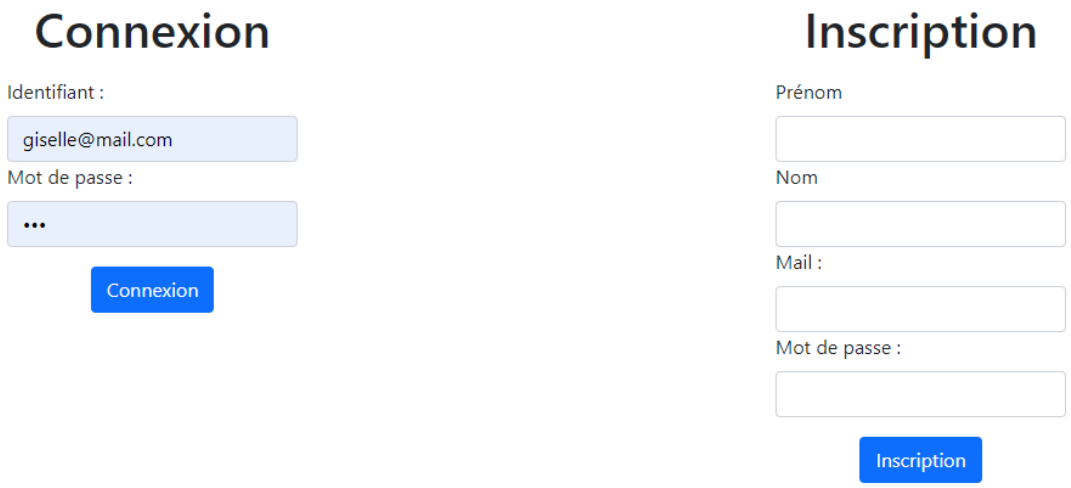


Figure 7 - Page d'accueil (login et inscription)

Pour cette authentification, j'ai utilisé un **LocalStorage** afin de stocker notre token afin qu'il soit valable le temps de la validité du token JWT. J'ai également **créé un Hook** afin de le personnaliser de sorte que quand un token JWT est renvoyé par le serveur, le hook le stock dans le localStorage et mette à jour la variable *token* qui permet à l'utilisateur d'accéder à l'application.

La principale difficulté de cette résidait dans la communication entre composants couplé avec un appel au serveur. Mais une bonne rigueur et compréhension des hook a permis de venir à bout de cette tâche.

Page Liste Stations

Durée 1 semaine

Cette page a été conçue afin d'afficher les mêmes informations que sur la page Carte, mais sous forme de liste. Elle se découpe en deux parties, une liste de stations triée par défaut alphabétiquement, avec la possibilité de cliquer sur une station et afficher plus d'informations sur cette station. Elle regroupe également une partie Filtre qui permet de filtrer cette liste selon le type de station (Autoroute, standard), le prix et le type d'essence. Le composant filtre est un composant commun avec la page Carte pour éviter une redondance de code. Le nombre de résultat est paramétré par défaut à 10 pour éviter

d'avoir une surcharge d'éléments. L'utilisateur aura néanmoins la possibilité d'augmenter cette limite via les filtres.

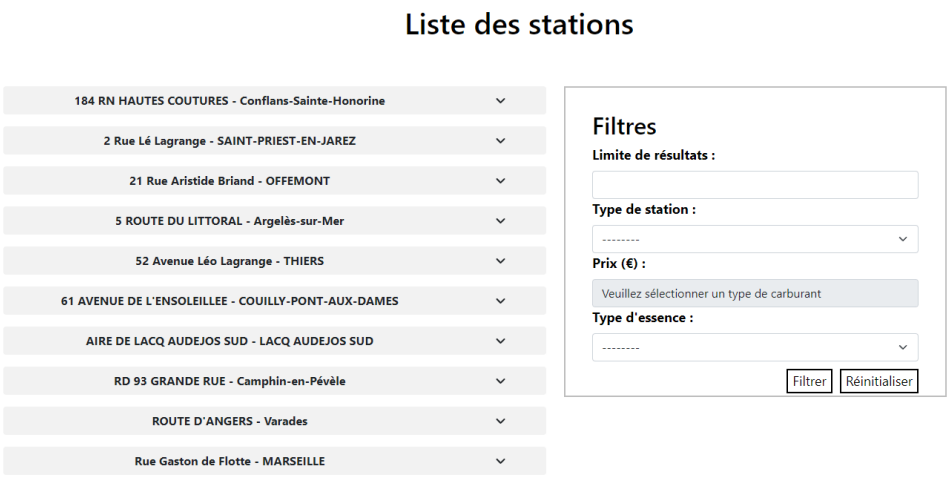


Figure 8 - Liste des stations

Je n'ai pas utilisé de librairie ou d'autres Hook que ceux utilisés dans la page Statistiques (**useEffect**, **useState**). Je n'ai pas non plus eu de points de blocage.

Mise en page de l'application

Durée : quelques jours

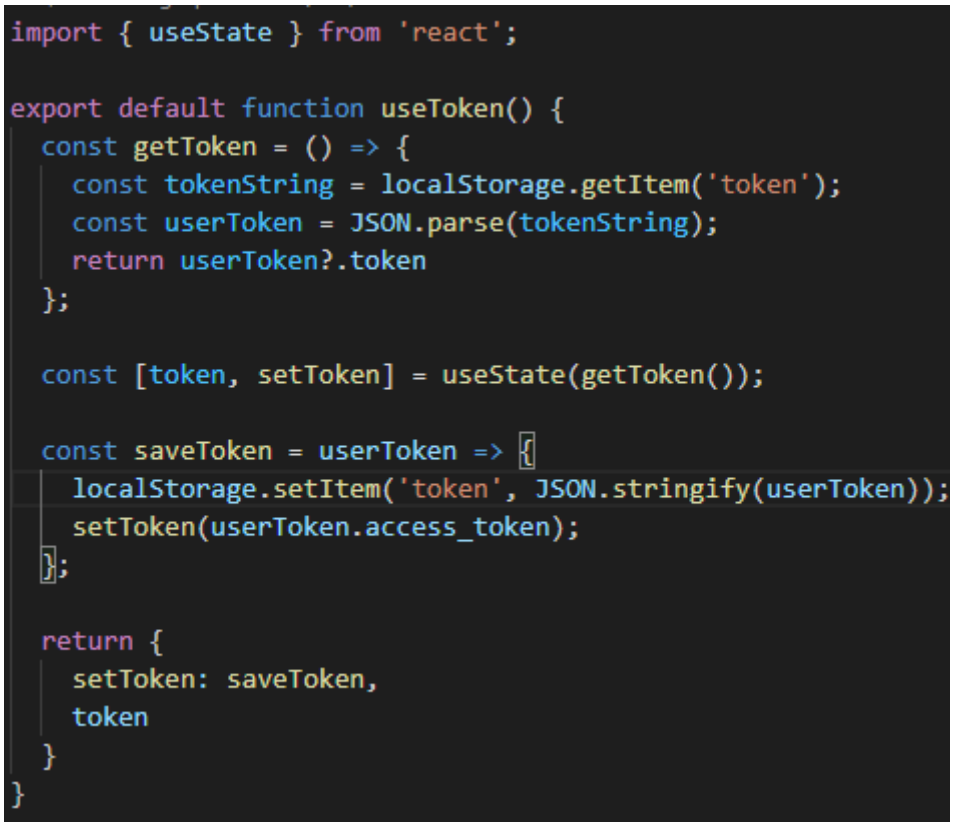
Pour garder une concordance entre les pages, il a fallu mettre en forme correctement les éléments et gérer l'affichage sur différents appareils (responsive design). Pour cela, nous avons fait le choix d'utiliser **react-bootstrap**[2]. Ce choix s'est fait naturellement car c'est une librairie que nous avons déjà tous pratiquée au travers d'anciens projets. Quelques adaptations ont en revanche dû être apportées car pour ma part, j'avais utilisé cette librairie au travers d'un projet Angular. Mais le concept de Grille reste identique.

Nous avons mis en place un système de thème Clair – Sombre selon l'affinité de l'utilisateur. Il a fallu alors créer un design spécifique selon ces deux thèmes.

[2] React-bootstrap : https://react-bootstrap.github.io/

Code

Code optimal (useToken.js)



```
import { useState } from 'react';

export default function useToken() {
  const getToken = () => {
    const tokenString = localStorage.getItem('token');
    const userToken = JSON.parse(tokenString);
    return userToken?.token
  };

  const [token, setToken] = useState(getToken());

  const saveToken = userToken => {
    localStorage.setItem('token', JSON.stringify(userToken));
    setToken(userToken.access_token);
  };

  return {
    setToken: saveToken,
    token
  }
}
```

A mon sens, la création d'un Hook dans le cas de l'authentification d'un utilisateur fut une idée bonne du point de vue conception et architecture. Premièrement, j'ai pu adapter mon Hook à mon cas d'usage et lui ajouter seulement les fonctionnalités dont il a besoin pour fonctionner. Cela permet de masquer une logique complexe derrière une interface simple, ou aider à démêler un composant dont le code est incompréhensible.

Code à optimiser

Je pense que nous aurions pu optimiser la mise en cache des données récupérées depuis le serveur. En effet, avec une meilleure uniformisation des appels nous arions pu éviter l'appel de requêtes déjà fait auparavant avec un système de mise en cache. Les données déjà récupérées seront simplement stockées le temps de la session de l'utilisateur et récupérées si nécessaires. Cela aurait permis une navigation plus rapide entre les composants (le temps de chargement des données étant supprimé) et nous aurait également permis l'ajout d'un spinner de chargement de données le temps que les composants basés sur les données soient montés.