Programmation Web

Rapport Client-Side

Frédéric Météreau - Identifiant GitHub

Tâches effectuées

Pour ce projet, nous avons réalisé plusieurs fonctionnalités. Il fallait avoir trois types de visualisation : un graph, une table et une map.

Personnellement, je me suis occupé de la partie concernant la map. J'ai également pu m'occuper avec Charly de la partie sur les filtres.

Afin d'avoir accès aux données, nous passons par une API REST qui est faite en Nest JS.

Stratégie employée pour la gestion des versions avec Git

Concernant la stratégie employée pour ce projet, nous avons opté pour un branching strategy de type Git Flow. Nous avons donc une branche master et une branche develop. Lorsqu'une version nous satisfait, elle sera mise sur la branche master. Dans notre cas, nous avons qu'une version mise sur la branche master, et celle-ci correspond à la version du projet que nous avons présenté lors de l'évaluation.

Étant donnée la complexité du projet relativement faible, il n'était pas nécessaire de mettre en place un système plus élaboré. Pour pouvoir faire les requêtes sur l'API, nous avons utilisé la librairie fetch.

Solutions choisies

Pour le composant de la Map, j'ai utilisé la librairie *Leaftlet* qui permet de facilement créer une map sur notre application web. Il existe d'autres alternatives pour créer une map mais *Leaftlet* me paraissait la librairie la plus simple à utiliser.

J'ai utilisé les *useContext* pour gérer les filtres. J'ai d'abord utilisé les props et les *useStates* mais je me suis vite perdu et cela était assez lourd. J'ai donc décidé de passer à des useContext pour que ça soit plus simple.

Difficultés rencontrées

J'ai eu une première difficulté concernant la map, car je n'arrivais pas à l'afficher correctement sur la page. J'ai finalement réussi à régler après un moment à lire la documentation et à chercher des solutions.

Une deuxième difficulté rencontrée est l'appel des fonctions pour récupérer les données du back. En effet, je n'arrivais pas à mettre à jour les markers présents sur la map avec les nouvelles stations que le back nous renvoyaient. Finalement, après plusieurs essais, et en passant par les props, j'ai réussi à régler ce souci.

Temps de développement de la tâche

Il m'a fallu apprendre React et utiliser les différents hooks qui existent. Cela m'a pris un peu de temps mais je dirais que le développement global de ma partie sur le front m'a pris environ 6-7 heures.

Code

```
function Map({stations, position}) {

  return (
    <MapContainer center={[43.773644112573116, 6.546820326950372]} zoom={6}>
      <TileLayer
        attribution='&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors'
        url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"
      />
      <MarkerClusterGroup disableClusteringAtZoom={13}>
        {stations.map((value, index) => (
          <Marker position={[value.latitude, value.longitude]} icon={customMarker} key={index}>
              <Popup className="popup">
                  <span style={{fontWeight: "bold", fontSize: "1.4em"}}>{value.adresse === undefined ? value.ville : value.adresse}</span>
                  <div style={{display: "flex", flexDirection: "row", width: 'max-content'}}>
                      <PopupFuel price={value.prix} rupture={value.rupture}/>
                      <PopupSchedule schedule={value.horaires}/>
                  </div>
              </Popup>
          </Marker>
        ))}
      </MarkerClusterGroup>
        <Marker position={[position.latitude, position.longitude]} icon={customMarkerMyPosition}>
            <Popup>
                Ceci est votre position !
            </Popup>
        </Marker>
    </MapContainer>
  );
}

export default Map;
```

Pour moi, ce composant est bien car il va être rendu à chaque fois que les stations changent ce qui est essentiel car on veut mettre à jour les markers sur la map après les avoir filtrées.

Ensuite, l'utilisation de composants de la popup pour séparer l'affichage des différentes informations me semble plutôt correcte.

```
function useForceUpdate(){
    const [value, setValue] = useState( initialState: 0); // integer state
    return () => setValue( value: value => value + 1); // update the state to force render
}

export default function Filters({fuels, changeStations, position}) {

    const [localisation, setLocalisation] = useContext(FiltersLocalisationContext)
    const [filterPrice, setFilterPrice] = useContext(FilterPriceContext)
    const [filterFuel, setFilterFuel] = useContext(FilterFuelContext)
    const { toggleTheme, theme } = useContext(ThemeContext)
    const forceUpdate = useForceUpdate();

    function handleSubmit() {
        let filters = []
        filterPrice.forEach(element => filters.push({"fuel": element.fuel, "priceMin": element.priceMin, "priceMax": element.priceMax}))
        filterFuel.forEach(element => filters.push({"fuel": element}))
        let args;
        if (localisation === '')
        {
            let distance = {
                "distance": 100,
                "position": {
                    "lat": position.latitude,
                    "long": position.longitude
                }
            };
            args = {distance, "fuelFilter": filters}
        } else {
            args = {"postalCode": localisation,"fuelFilter": filters}
        }
        console.log(args)
        changeStations(args).then()
    }
```

Pour moi, ce composant n’est pas optimisé avec l’utilisation de la méthode forceUpdate. J’ai dû utiliser cette méthode car je n’arrivais pas à mettre à jour le composant lorsque je voulais ajouter un filtre de prix.