Server & Client side

Rapport personnel
ROBERT Florent

ID GitHub :
FlorentRO

Tâche effectuée

Durant ce projet, je me suis principalement concentré sur le front, plus précisément sur l'affichage de graphes. J'ai dû pour cela dans un premier temps regarder les données dans notre base de données, et discuter avec Théo Fafet des différentes routes et filtres à mettre en place afin de pouvoir faire un affichage de graphes intéressants. J'ai ensuite recherché différentes librairies permettant l'affichage de graphe répondant à différents critères. Une fois cela fait, j'ai pu développer la page "Statistique" du front, en récupérant les données du back, avec les différents éléments qui viennent avec celui-ci :

- Select permettant à l'utilisateur de choisir les données à afficher
- Dark & Light mode modifiant les select & graphes
- Lazy import appliqué aux graphes

Stratégie employée

Pour ce qui est de la stratégie employée, nous avons l'habitude lors de ce type de projet de mettre en place un kanban en définissant les différents EPIC, que nous divisons en tâches que l'on va pouvoir attribuer à un membre précis. C'est une

organisation très utile pour avoir un bon suivi de projet, mais trop lourde pour un projet comme celui-ci. Nous avons dans un premier temps définit les différentes EPIC :

- La base de données MongoDB
- Le back, route & filtres
- L'affichage du tableau
- L'affichage de graphes
- L'affiche de la carte

Nous nous sommes ensuite oralement attribué ces tâches. Nous avons par la suite discuté de points plus secondaires pour le fonctionnement du projet (lazy import, dark mode, etc).

Nous avons créé une branche dev sur laquelle nous avons fait le développement, pour garder une version toujours fonctionnelle sur la branche main.

Enfin, comme conseillé lors des cours de client side, nous avons utilisé les préfixes refactor : / fix : / build : lors de nos pushs afin de préciser la nature du push. C'est une pratique que je n'avais encore jamais utilisée, je la trouve très pratique et continuerai probablement de l'utiliser pour mes prochains projets de développement.

Solutions choisies

Choix de la librairie

Pour le travail effectué, j'ai dû choisir une librairie permettant l'affichage de graphes sous React. Il y avait le choix entre de nombreuses librairies (nivo, BizCharts, ...), mais mon choix s'est porté sur CanvasJS. Cette librairie permet la création de nombreux types de graphes relativement simplement, et propose une documentation très complète. De plus, CanvasJS propose des thèmes permettant de faire le thème dark & light, étant l'une des contraintes obligatoires du projet.
CanvasJS n'étant pas disponible dans npm, il a cependant été nécessaire de l'installer manuellement, étant le principal défaut de cette librairie.
https://canvasjs.com/

Appel aux composants fetch

Contrairement aux autres parties du front, faisant un unique appel au composant fetch dans le App permettant la récupération de l'intégralité des données du back, ensuite envoyées en input aux composant, le composant des graphes "CasesNumberGraph.js" fait lui-même appel à son composants de fetch.
Sur la page des graphes, l'utilisateur peut choisir les données à afficher dans les graphes, demandant alors des données différentes du back. Face à ce problème, il y avait deux solutions possibles :

- Récupérer toutes les données du back directement dans le App.js

Cette première solution permet d'éviter de devoir faire plusieurs requêtes, cependant les données voulues pour les graphes sont des données très volumineuses, concernant le nombre de cas pour chaque tranche d'âge, chaque semaine, pour chaque département, prenant plusieurs secondes à récupérer. (avec un bon réseau).

- Récupérer une petite partie des données du back dans le composant

Cette seconde solution nous permet à l'aide de filtre d'effectuer de petites requêtes à chaque fois que l'utilisateur le demande. Cela permet au final de bien moins solliciter l'accès réseau au serveur, étant donné qu'un utilisateur la majorité du temps va seulement regarder une petite partie précise des données, et non l'intégralité.

Cela souligne aussi la problématique du "faire le travail côté client ou serveur", la première solution visant à faire travailler le client, là où la seconde vise à faire travailler le serveur. Il me paraît logique qu'il faille faire travailler le serveur plus que le client, afin de ne pas rendre pénible l'utilisation de notre site pour un utilisateur avec un mauvais ordinateur ou réseau. Cela demande cependant évidemment de meilleurs serveurs pour faire des filtres sur les requêtes plutôt que de simplement tout envoyer.

Difficultés rencontrées

Données d'input vide

Un problème commun lorsque l'on travaille avec des requêtes vers un back ou une API est le temps de récupération des données. Nous avons eu ce problème plusieurs fois lors du projet. Dans mon cas pour la récupération des données nécessaires pour les graphes, ainsi que pour la récupération des données de géolocalisation de l'utilisateur afin de lui afficher par défaut les graphes correspondant à son département. Face à ce problème, j'ai mis en place des lazy imports, permettant d'afficher une image de chargement lorsque ces données ne sont pas encore reçues

Thème des graphes

Un second problème qui a été remarqué assez tôt est le fait que les graphes ne s'adaptent pas forcément au thème actuel de l'application (dark & light). Nous avons pour cela cherché une librairie permettant la gestion de thème, et avons trouvé CanvasJs gérant cela.

Architecture des graphes

Enfin, un problème que j'ai remarqué vers le milieu du développement du projet était que l'architecture de mes composants de graphe était peu extensible. J'ai pour cela fait un refactor afin de centraliser dans un composant parent les différentes informations, de sorte à ce que les composants de graphes soient réutilisables pour plusieurs types de données totalement différents, simplement en leur donnant les différents titres, axes et données en input.

Temps de développement / tâche

J'ai travaillé sur ce projet un total d'environ 20h.

- Choix de la librairie permettant l'affichage de graphe (~2h)
- Mise en place des graphes et des select pour les requêtes au back (~10h)
- Refactor de l'architecture & composants des graphes (~3h)
- Light & Dark thème pour les graphes (~2h)
- Lazy import pour les graphes (~3h)

Code

Composant élégant ou optimal

```
if(GraphCurve == null) GraphCurve = lazy(
      () => new Promise((resolve, reject) =>
      FetchServerInputData("http://localhost:8080/covid_data/heb/dep?de
      p="+currentDep).then(data => {
            updateCurveGraph(data);
            resolve(import("./GraphCurve"));
      }))
);
```

Ce bout de code (CasesNumberGraph.js, ligne 42), dans le cas où GraphCurve n'a pas encore de valeur (ce qui correspond au moment où l'utilisateur arrive sur la page la première fois), va appeler le composant Fetch en lui donnant en paramètre le path avec le département donné en input au composant. Ce composant est appelé après un lazy import, assurant que la valeur currentDep est la bonne. Après avoir récupéré les données du back, on va update le graphe curve, et effectuer le lazy import du graphe curve pour afficher celui-ci. Je trouve ce bout de code élégant, appelé après un lazy import, effectuant un lazy import à la fin, permettant de donner un bon exemple d'une application gérant bien l'affichage de données en asynchrone.

Composant qui mériterait une optimisation

Le composant CasesNumberGraph.js s'occupant des différents graphes est un composant faisant trop de choses différentes. Celui-ci pourrait devenir une dette technique par la suite, devenant peu extensible, voire pas du tout. J'ai déjà lors du projet fait un refactor de celui-ci permettant de réduire ce problème, en séparant totalement les composants de graphe de ce composant parent, mais je pense qu'il serait bon pour la suite de le refactor à nouveau pour le séparer en deux composants différents, un pour la gestion du graphe "GraphColumn", et un autre pour la gestion du graphe "GraphCurve".