Rapport programmation web

- Client -

Auteur : AZZOUZI Nouamane

Equipe :
AFKS

Identifiant Github

AzzouziNouamane / nouamane.azzouzi@etu.unice.fr

Tâches effectuées

- Composant qui affiche le nombre total de cas découverts en France en temps réel, et qui se met à jour automatiquement **(Backend : pour récupérer les données en temps réel depuis une API externe / Frontend : Récupération des données et l'affichage du composant)**
- Mode jour et nuit, persisté en local chez l'utilisateur
- Utilisation de l'API du navigateur du client pour détecter son thème par défaut, et ainsi adapter le mode jour et nuit en accord avec son thème.
- Affichage d'un spinner lors du chargement de données depuis le backend
- Affichage d'un message d'erreur en cas d'erreurs survenues sur le backend
- Déploiement du backend sur Heroku
- Déploiement du frontend sur Netlify

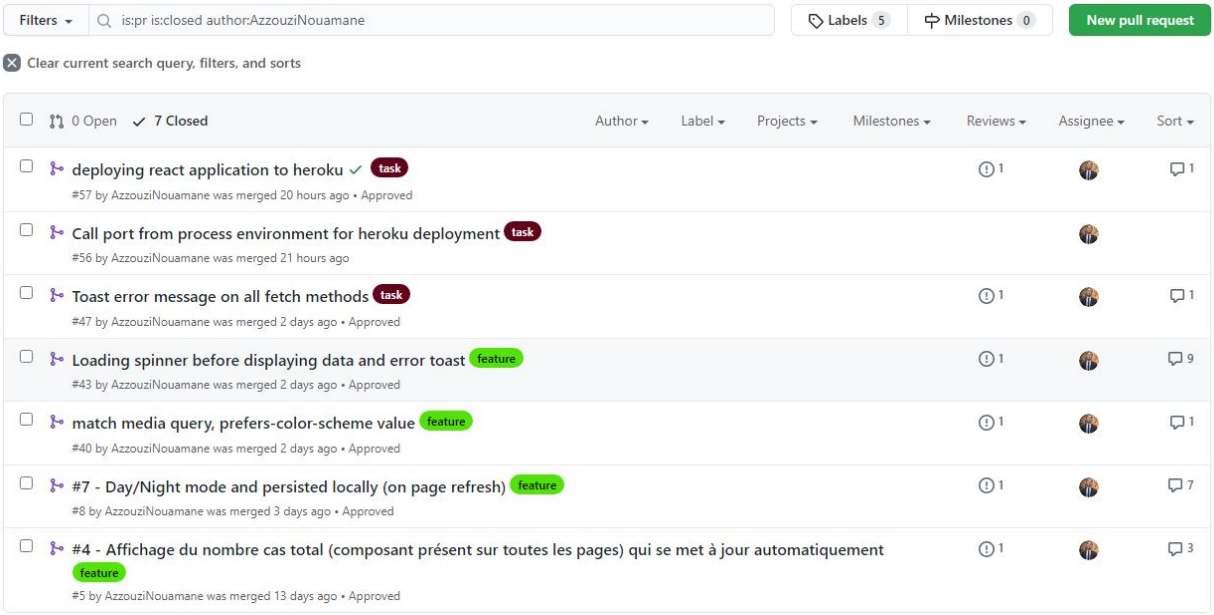

Figure 1 : Historique de mes Pull Requests

Stratégie employée pour la gestion des versions avec Git

Pour la gestion des versions avec Git, nous nous sommes tous mis d'accord au début sur les tâches à effectuer, et nous les avons toutes indiquées sur le **Kanban**. Ensuite, chacun devait s'assigner la tâche sur laquelle il allait travailler et la mettre dans la catégorie **In Progress** du

Kanban puis créer une nouvelle branche destinée à cette tâche. Une fois terminé, il crée la Pull Request qui pointe sur la branche Main, en mettant **1 Reviewer**, et l'informe sur le channel slack de l'équipe. Une fois la review terminée et acceptée, **la PR peut être merge sur Main.**

Le scénario de notre stratégie est le suivant :
On prend par exemple 2 membres de l'équipe, nommés **A** et **B** :

- **A** prend un ticket textuel
- **A** convertit le ticket en issue et précise dans la description de l'issue ce qu'il va faire
- **A** s'assigne dessus et met les bons labels
- **A** crée une branche pour bosser sur cette issue
- quand l'issue est terminée, **A** crée une Pull Request, assigne **B** en review et écrit dans le channel slack "Pull Request prête pour review par **B**: lien vers la Pull Request"
- **B** fait la review de code (et la review fonctionnelle si possible)
- si tout est bon, **B** merge la branche dans Main et ferme l'issue, sinon il demande à **A** de corriger des trucs
- **A** corrige et redemande une review à **B** etc. jusqu'à ce que le merge ne soit effectué
- pendant son développement, **A** n'oublie pas de merge Main dans sa branche pour gérer les conflits au fur et à la mesure

Solutions choisies

L'utilisation de **Material-UI** nous a été d'une grande aide. En effet, ce dernier dispose d'un très grand nombre de composants très utiles pour notre développement. Le spinner utilisé pour le chargement de données est le composant **CircularProgress** de Material-UI, qui permet d'afficher facilement un spinner (avec la couleur de notre choix, et donc c'est possible de l'afficher à la couleur de la marque par exemple et donc garder un code couleur cohérent au sein de l'application).

Alternatives

Il existe une bibliothèque **react-spinner** qui, elle aussi, permet d'afficher un spinner de chargement. Mais nous avons préféré celui de Material-UI car on utilise les composants de cette bibliothèque dans pleins d'autres exemples et que ça ne servait à rien d'ajouter d'autres dépendances à l'application.

Pour les messages d'erreurs, nous avons utilisé la bibliothèque React-toastify, qui dispose d'un large éventail de style de popup à afficher, notamment pour les messages de succès, d'erreur ou d'information. Cela permet d'afficher un message personnalisé en fonction du contexte.

Alternatives

Il est également possible d'afficher uniquement un message d'erreur en brut dans l'application, mais visuellement ce n'est pas très attractif, et si on souhaitait régler l'ergonomie il aurait fallu créer tout une règle CSS (ou plusieurs) pour cela. React-toastify permet de faire tout cela très facilement sans perdre de temps.

Difficultés rencontrées

L'implémentation de la notion de Context a été assez compliquée à mettre en place. En effet, c'est après plusieurs tutoriels et plusieurs essais que j'ai saisi l'idée derrière cette implémentation.

Ainsi que l'utilisation du LocalStorage, où je n'arrivais pas à cerner où je devais faire appel au Setter du localStorage, où et quand récupérer la donnée persisté. Également, lors de la première connexion d'un utilisateur sur le site, il n'a à priori aucune donnée persisté localement, et donc il a fallu mettre en place une valeur par défaut au début.

Temps de développement/Tâche

Le temps de développement par tâche est bien sûr approximatif ***(Divisés en semaines de travail)***!

Tâche	**Temps de développement**
Affichage du nombre total de cas découverts en France en temps réel, et qui se met à jour automatiquement.	2 Semaines pour créer les différents composants, créer la requête sur le backend correspondante, persister les données sur le local et utiliser l'API du navigateur.
Mode jour et nuit, persisté en local chez l'utilisateur	
Utilisation de l'API du navigateur du client pour détecter son thème par défaut, et ainsi adapter le mode jour et nuit en accord avec son thème.	
Affichage d'un spinner lors du chargement de données depuis le backend	1 semaine pour mettre en place le spinner sur tous les fetch() effectués dans l'application et également pour afficher les messages d'erreur.
Affichage d'un message d'erreur en cas d'erreurs survenues sur le backend	
Déploiement du backend sur Heroku	1 journée pour déployer le front et le back sur les 2 hébergeurs.
Déploiement du frontend sur Netlify	

Code

Fonction/Composant écrit de manière élégante et optimale

Le composant du Thème de l'application **(ThemeMode.js)** et également l'utilisation de la notion de Context pour transférer et partager la valeur du thème avec tous les autres composants de l'application.

Sans utilisation du Context, on aurait dû passer en props la valeur du thème à tous les composants manuellement, ce qui est très redondant et très lourd pour l'application. Mais en utilisant le Context, on peut ainsi éviter de passer les props à travers des éléments intermédiaires.

```
1   import React from 'react';
2
3   export const themes = {
4       dark: {
5           isDark:  true,
6           color:  '#F5F5F5',
7           background: 'black'
8       },
9
10      light: {
11          isDark:  false,
12          color: 'black',
13          background: '#F5F5F5'
14      }
15  };
16
17  const ThemeContext = React.createContext(themes.dark);
18
19  export default ThemeContext;
20
```

Figure 2 : ThemeContext.js

Fonction/Composant écrit méritant une amélioration

L'affichage d'un message d'erreur lors d'une erreur entre le frontend et le backend.
En effet, ça aurait été plus intéressant de créer un composant spécifique (**Error.js** par exemple), qui est chargé spécialement de l'affichage d'un message d'erreur spécifique en fonction du contexte et du type d'erreur rencontré.

Pour le moment, nous avons juste une fonction **dataLoadingError()** dans notre fichier utilitaire **Utils.js** qui est chargé de faire cette affichage là.
On utilise la bibliothèque **react-toastify** pour afficher la popup d'erreur

```
25  export const dataLoadingError = () => toast.error('Une erreur est survenue lors du chargement des données ! Merci de réessayer ultérieurement.',
26      position: "bottom-left",
27      autoClose: 5000,
28      hideProgressBar: false,
29      closeOnClick: true,
30      pauseOnHover: true,
31      draggable: true,
32      progress: undefined,
33  });
34
```

Figure 3 : fonction dataLoadingError()