Projet React - Client side

Charles ROGER

Introduction

Identifiant Github : **CharlesRoger62**

Tâches effectuées : - Construire les mocks du projet (1h)

- Développer le thème persistant (6h)
- Ajouter le cadre de visualisation des taux par région / par département (10h)
- Ajouter le graphique de visualisation du nombre de test par département d'une région. (8h)
- (BACK) Ajouter des requêtes sur les datas du back end (2-3h)
- (BACK) Optimiser l'import de données. (2h)

Stratégie employée pour la gestion des versions avec Git :

Notre gestion du git a été un élément déterminant du projet (ou bien comme en mal). En effet, nous avons mis en place une stratégie de découpage des branches par composants sur nos mocks. Bien que cela puisse être efficace puisque que notre projet est lui-même développé selon une structure ou chaque dossier contient un composant, nous avons rapidement rencontré des problèmes de conflits sur l'import de données.

L'ensemble des branches est alors merge sur la branche Unreviewed-Works par pull request ou par programmation per to per. Nous avons eu beaucoup de problèmes liés au merge de données à cause des npm install sans le –save ou l'avancement des branches principales avant une branche component.

Solutions globales choisies

Architecture du projet

Comme vu précédemment, nous avons axé notre développement sur des composants qui affichent des données. Côté Back, nous avons fait diverses réunions pour définir exactement quelle architecture nous allions définir.

Nous avons choisi 3 sets de données pour le back afin de former une architecture composée des régions et départements et des données et par jour (qui par jour regroupe les données nationales).

Côté Front, nous avons défini une structure grâce à des mocks up précises en définissant quelles étaient nos différentes pages et les composants qui la composent. Cela nous a permis de faire une répartition extrêmement précise et rapide de ce que donnera globalement l'application finale. Les mocks finales faites en début de projet sont disponibles en annexe. Le nombre de vues et de visualisation des données sont cohérentes avec notre architecture et permet de donner un sens intéressant à notre front en choisissant des visualisations globales puis plus précises (données nationales -> par région -> par département).

Quelles sont les alternatives ?

Il aurait été possible de fonctionner par spécialité, donner des rôles en front et back et spécialiser les membres pour rendre le travail plus simple et plus profond. Mais la division des deux matières (Server, Client) ne permet pas ce type de configuration.

Difficultés globales rencontrées

La principale difficulté du projet a été le suivi de projet en lui-même ainsi que le côté back et le format de donnée principalement.

Côté gestion de projet nous avons beaucoup de problèmes liés aux merges et au dependances (ne pas oublier –save). Cela est plutôt dû à manque d'expérience côté projet React.

Côté Front, l'utilisation des hooks dans le cycle des states nous ont confrontés à des situations particulières pour l'affichage de composant avec des dépendances de beaucoup de données. De plus, les css en général resteront le point faible de notre projet ; l'aspect esthétique ayant été moins approfondi que l'aspect technique.

Tâches

1 . Réalisation du thème (6h)

Attention : la création du thème est basée sur un tuto existant que j'ai suivi.

Le principe du thème que j'ai développé est plutôt simpliste sur l'affichage mais utilise divers procédés qui m'ont permis de tester les fonctionnalités vues en cours. Le principe de ce thème est simple, il va changer le background-color d'un élément en blanc ou noir et l'inverse pour les textes.

Impact sur le projet :

Le thème est placé à la racine de App.js et permet de faire découler le thème sur l'ensemble du projet par le biais d'un ThemeProvider. Pour la structure du composant, le modèle est semblable au MVC car en plus des deux css a appeler en fonction du thème choisi, j'utilise un fichier pour les fonctions en cas d'une modification et un fichier pour créer et exporter le bouton.

Ces facteurs intéressants sont l'utilisation des useState ainsi que du window.localStorage afin de créer la persistance du thème choisi et ce malgré un rafraîchissement du site.

Solution/Blocages :

Malheureusement, divers problèmes liés à la superposition des css empêchent le thème de s'appliquer partout. Le problème majeur auquel j'ai été confronté a été des problèmes liés à la définition des svg pour les composants qui m'ont empêché de travailler sur le changement de couleur des graphes et de la map.

2. Ajouter le cadre de visualisation des taux par région / par département (10h)

Le cadre de visualisation des données repose sur un composant simple qui puise sa difficulté dans le large panel de données qu'offre le back. En effet, ce composant est un composant qui sert dans deux vues (ie il est réutilisable) et permet d'afficher les taux pour la page globale du projet comme pour les pages de détails des régions.

Impact sur le projet :

Le cadre permet une utilisation pratique pour démontrer la structure de notre projet : il doit être capable d'afficher des données précises en fonction de deux choses : l'endroit ou il se trouve et la date qui a été choisie. Pour cela, il a fallu développer des Loaders précis et multitâches qui étaient capables de retourner des données en fonctions de ces deux contraintes.

Ainsi, ce composant nécessite d'utiliser le useLocation() pour savoir où l'on se trouve mais aussi des useState() pour rafraîchir les données sans avoir à changer de composant ou le détruire.

Ce composant a particulièrement mis en lumière les erreurs provenant de notre BDD puisque l'affichage des taux se fait à partir de données calculées.

Solution/Blocages :

La principale difficulté pour ce composant a été sa communication avec le back et l'utilisation des Loaders et de corriger toutes les erreurs liées au back.

De plus, le date-picker a nécessité de passer par de nombreuses manipulations pour les formats de dates (en utilisant moment js) puisqu'il utilise des objets Date quand les requêtes fonctionnent par string. Le date-picker avait un conflit svg avec la map et ses boutons étaient déformés par les paramètres donnés dans la map alors la lecture complète de la doc a été nécessaire pour sortir le css de node module et le renommer pour le modifier à l'extérieur.

3. Ajouter le graphique de visualisation du nombre de test par département d'une région. (8h)

Le composant line chart n'est actuellement pas visible sur le projet malgré de nombreuses tentatives.

Le composant est capable d'être affiché sans réelle erreur actuellement et se trouve dans le projet (components/case_chart) mais n'est pas inclus car ne permet pas de plot des données. La raison est encore inconnue à ce jour.

Par conséquent, il n'y aura pas d'impact sur le projet ni de solution et son fonctionnement sera détaillé dans la partie code.

Code

Fonction/Composant élégant : le rate component

Voir Annexe 5.

La fonction onDateChange() est appelée lors de la sélection d'une date sur le composant Date-picker.

La fonction envoie alors une nouvelle valeur qui est une date et qui dans un premier temps transformé en string par l'intervention du module moment avant de correspondre aux formats pour les requêtes de données. Ensuite on différencie dans quel endroit du site nous nous trouvons par l'intermédiaire d'une variable qui prend la valeur du router et de useLocation() et enfin on appelle le Loader pour prendre les données. Comme celle-ci est asynchrone, on attend la réception des données afin de les affecter aux trois variables des taux qui sont modifiés par le biais du hook useState()..

Fonction/Composant à améliorer : la line chart

Voir annexe 6.

Le code utilisé pour plot les données récupérées du back n'est pas fonctionnel, cela provient très probablement d'un souci au niveau des données à cause de la complexité. En effet, la requête renvoie les données de toutes les données par jour par départements pour une région précise. Cela représente pour un test que nous avons fait un tableau d'une dizaine de paramètres x 228 jours de données x 12 départements. D'une part la requête prend un certain et aussi le triage des données, mais la structure ne nous a pas permis d'entrevoir la solution pour plot correctement. Les améliorations à apporter sont une optimisation de la requête en importer uniquement ce qui nous intéresse, une amélioration dans l'écriture des données obtenues ainsi qu'une connaissance plus approfondie sur comment faire fonctionner correctement les plots du line chart de d3js lorsque l'on a des données lourdes et très variables.

Conclusion :

Ce projet a permis une expérience très intéressante pour faire du fullstack (ce qui est rare en spécialité IHM) et approfondir la techno React qui s'avère être difficile à prendre en main mais avec une liberté énorme au niveau du code.

Annexes :

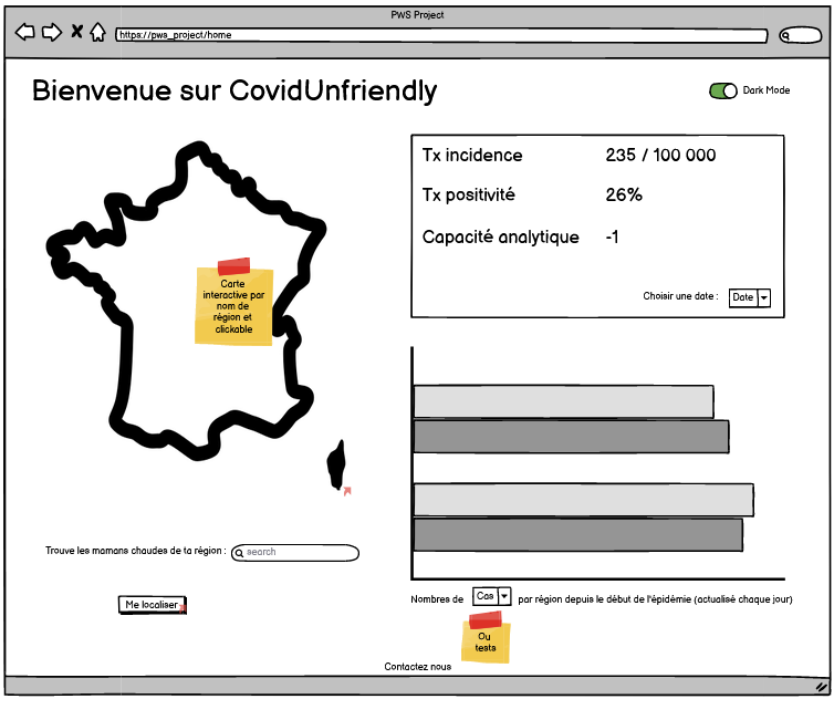

Figure 1 : Mockup de la vue principale

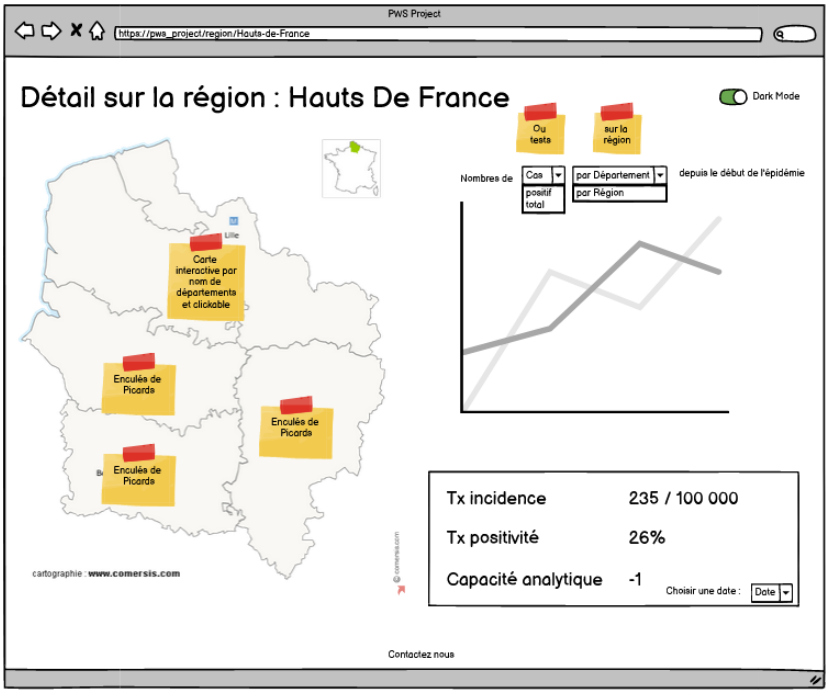

Figure 2 : Mockup de la vue détail région

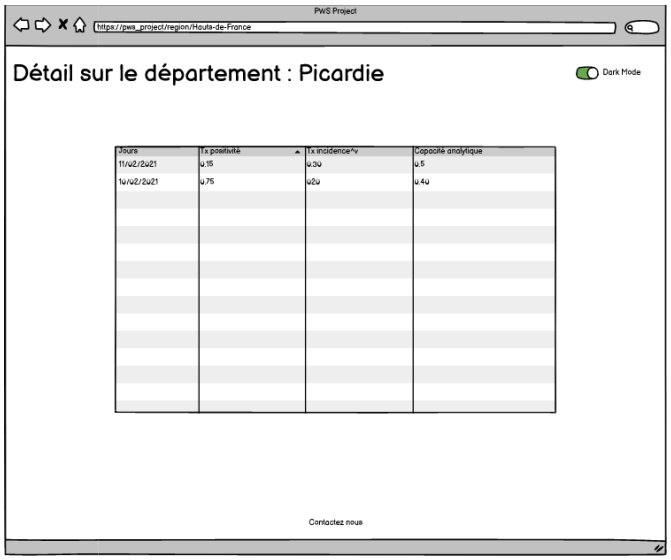

Figure 3 : Mockup de la vue détail département

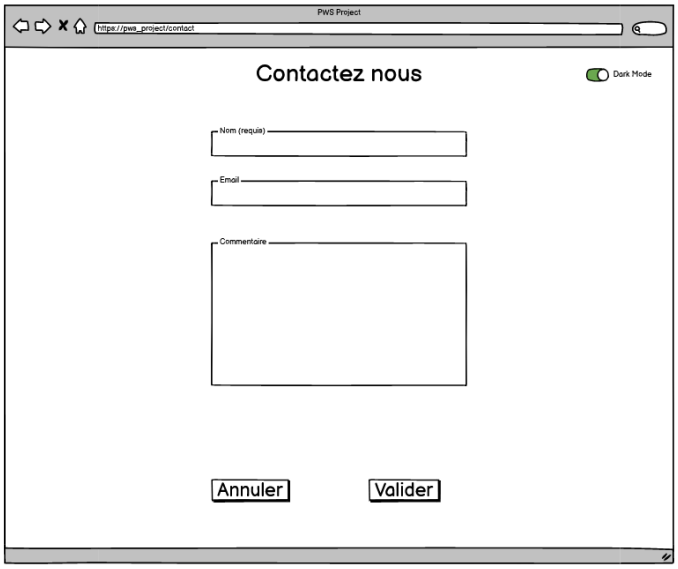

Figure 4 : Mockup du Form pour nous contacter.

```
const onDateChange = value => {
  onChange(value);
  var stringDate = moment(value).format("yyyy-MM-DD");
  var data;

  if(location.pathname == "/state"){
    data = FranceCovidDataLoader(stringDate);

    data.then(
      v => {
        if(v[0].tx_an == null) {
          let error = 'pas de donnée';
          setAnalytic(error);
          setIncidence(error);
          setPositivity(error);
        } else {
          setAnalytic(v[0].tx_an);
          setIncidence(v[0].tx_inc);
          setPositivity(v[0].tx_pos*100);
        }
    })
  } else {

    var regionName = location.state.regionName;
    let regionNum = RegionEnum[regionName];
    data = RegionCovidDataLoader(stringDate,regionNum);

    data.then(
      v => {
        console.log(v);
        setAnalytic(v.data["0"].tx_an);
        setIncidence(v.data["0"].tx_inc);
        setPositivity(v.data["0"].tx_pos*100);
    })

  }

};
```

Figure 5 : code élégant

```
const yScale = scaleLinear()
  .domain([0, max(finalTab[0], (d) => {return d.nbtest;})])
  .range([height - 10, 10]);

const lineGenerator = line()
  .x((d, index) => xScale(index))
  .y((d) => yScale(d.nbtest))
  .curve(curveCardinal);

finalTab.forEach( (e) => {
    svgContent
  .selectAll(".myLine")
  .data(e.slice(0,6))
  .join("path")
  .attr("class", "myLine")
  .attr("stroke", "black")
  .attr("fill", "none")
  .attr("d", lineGenerator);
});

// axes
const xAxis = axisBottom(xScale);
svg
  .select(".x-axis")
  .attr("transform", `translate(0, ${height})`)
  .call(xAxis);

const yAxis = axisLeft(yScale);
svg.select(".y-axis").call(yAxis);

// zoom
const zoomBehavior = zoom()
  .scaleExtent([0.5, 5])
  .translateExtent([
    [0, 0],
```

Figure 6 : code à améliorer