# Compte rendu de programmation web – Client side

*Yan CONIGLIARO*

## Identifiant Github

Mon identifiant Github est **YanConigliaro**

## Taches effectuées

Les taches que j'ai pu effectuer durant ce projet sont :

- Voir le taux d'incidence de l'épidémie (partie backend) **#7**
- Filtres des données par classe d'âge, date de début et date de fin **#12**
- Formulaire de contact avec envoi de mail **#19**
- Un fix de bug où le graphique affiché plusieurs fois les mêmes dates **#38**

## Stratégie employée pour la gestion des versions avec Git

La stratégie que nous avons décidé d'adopter et la stratégie par merge. En effet, celle-ci est plus facile à utiliser, bien que l'historique de commit soit moins propre qu'une utilisation de stratégie par rebase.

Par ailleurs, nous avons spécifié différentes règles de développement :

- Création et répartition des différentes tâches au travers d'issue Github en équipe
- Le développement s'effectue sur des branches de développement locales sous la forme « Feature/US-<Numéro de l'issue> »
- Une fois le développement terminé, nous devons effectuer une Pull Request (PR) afin de fusionner notre code avec ceux des autres sur la branche de développement publique « develop »
- Chaque PR doit être approuvée par au moins un développeur (généralement 2), qui va regarder le code produit et y faire des remarques. Ceci permet de donner des conseils ou encore de vérifier qu'il n'y est pas d'erreur de développement qui se retrouve sur la branche de développement publique.

## Solutions choisies

Au niveau des fonctionnalités que j'avais à faire sur les formulaires, j'ai choisi d'utiliser React Hook Form qui m'a été recommandé dans par un développeur de l'équipe. J'ai donc fait mes recherches et en effet, c'était une librairie relativement facile d'utilisation et permettant de faire une validation de formulaire très rapide et facile. J'ai donc choisi de l'utiliser pour mes différents formulaires.

Les autres solutions auraient été :

- Faire le comportement à la main (via des fonctions qui se trigger lors d'un changement comme onChange), pas utile de s'embêter à réinventer la roue autant réutiliser la librairie qui nous fera gagner du temps

- Une autre librairie similaire à React Hook Form est react-form-with-constraints, sauf que cette librairie est beaucoup moins utilisée que React Hook Form et ai moins abouti

Pour le formulaire de contact, j'ai choisi d'utiliser SendPulse permettant d'effectuer des notifications, que j'avais déjà utilisé dans d'autres projets et disposant d'une version gratuite qui est très suffisante pour ce projet.

D'autres services permettant l'envoie de courriel gratuitement sont sendinblue ou encore sarbacane, mais n'ayant jamais utilisé ces services et ayant déjà un compte SendPulse, j'ai préféré l'utiliser.

## Difficultés rencontrées

L'une des premières difficultés que j'ai pu rencontrer, est lors du développement de la fonctionnalité sur les filtres.

En effet, un développeur de mon équipe m'avait conseillé d'utiliser la librairie React Hook Form pour faire le formulaire permettant la sélection des filtres, avec les composants Material-UI.

Lors de l'utilisation des composants DatePickers de material UI, je me suis rendu compte que ceux-ci ne pouvaient être utilisés tels quels avec React Hook Form, car on mixait alors des composants contrôlés avec des composants non contrôlés.

Il a fallu alors passer par le composant « Controller » de React Hook Form, spécifiant quel composant nous souhaitions utiliser via la props « as » et qui permettait alors d'utiliser le composant Material-UI avec React Hook Form.

Il m'a fallu pas mal de temps et de recherche afin de trouver cette solution et réussir à bien l'implémenter.

Une autre difficulté, encore une fois avec un formulaire, est lors du développement du formulaire de contact.

En effet, je n'avais pas très bien compris le principe de la prop register de react hook form, afin que celui-ci soit géré par le hook. Seulement certains champs Material-UI comme les TextField sont naturellement compatibles avec React Hook Form.

Puisque j'avais déjà eu des difficultés, j'avais essayé d'implémenter ce formulaire de la même façon que le formulaire des filtres. Or, si nous souhaitons utiliser pleinement les fonctionnalités des composants Material-UI avec les props « error » avec les fonctionnalités de React Hook Form, il nous faut cette fois si utiliser les composants naturels TextField et non passer par un Controller.

Ceci m'a pris pas mal de temps à trouver ce qui n'allait pas et de recherche, pour finalement réussir à faire fonctionner ce formulaire avec les bonnes vérifications (pattern de l'email, champs requis).

Une dernière difficulté, a été lors du fix du bug qui affecté les données renvoyées par le backend.

Afin que celle-ci soit optimisée, je souhaitais utiliser la fonction agregate de Mongoose, permettant d'agréger plusieurs requêtes mongo DB et plus d'effectuer la somme nécessaire des données sur les différentes régions. L'utilisation de cette fonction qui est effectuée directement sur la base mongodb serait beaucoup plus optimisée sur un nombre important de données, plutôt que de devoir traiter une à une les données en JS.

Ne connaissant pas du tout le fonctionnement de cette fonction, il a été difficile de la prendre en main. Heureusement, un développeur de mon équipe qui avait déjà utilisé cette fonction, m'a recommandé d'utiliser le logiciel « Compass » permettant de construire pas à pas la requête et de voir le résultat de celle-ci au fur et à mesure.

Cela m'a permis de gagner beaucoup de temps et de rattraper le temps que j'avais perdu à essayer de faire le pipeline de cette fonction à la main.

## Temps de développement / tâche

Globalement, je n'ai pas prêté attention au temps passé sur chacune des tâches car il m'arrivait d'étaler le développement de celle-ci sur plusieurs séances de développement dans la semaine en fonction de mes disponibilités.

En moyenne, je pense que je pourrais estimer à une soirée de développement pour effectuer une tâche.

## Code

### Commenter une fonction ou un composant (max 100 lignes) que vous avez écrit et qui vous semble élégant ou optimal

```
async findIncidenceRateFilters(query): Promise<IncidenceDocument[]> {
  const filters = this.createFilters(query);

  const pipeline = [
    {
      '$match': filters
    }, {
      '$group': {
        '_id': {
          'jour': '$jour'
        },
        'tx_std': {
          '$sum': '$tx_std'
        },
        'P_f': {
          '$sum': '$P_f'
        },
        'P_h': {
          '$sum': '$P_h'
        },
        'P': {
          '$sum': '$P'
        },
        'pop_f': {
          '$sum': '$pop_f'
        },
        'pop_h': {
          '$sum': '$pop_h'
        },
        'pop': {
          '$sum': '$pop'
        },
      }
    }, {
      '$sort': {
        '_id': 1
      }
    }, {
```

```
      '$project': {
        '_id': 0,
        'jour': '$_id.jour',
        'tx_std': '$tx_std',
        'P_f': '$P_f',
        'P_h': '$P_h',
        'P': '$P',
        'pop_f': '$pop_f',
        'pop_h': '$pop_h',
        'pop': '$pop',
      }
    }
  ]

  if (query.reg) {
    pipeline[1]['$group']['reg'] = {'$first':'$reg'};
    pipeline[3]['$project']['reg'] = '$reg';
  }

  const data = await this.incidenceModel.aggregate(pipeline).exec();

  data.map(elem => {
    elem.cl_age90 = filters["cl_age90"];
  })

  return data;
}
```

Avec la fonction createFilters :

```
createFilters(query) {
  const filters = {};
  if (!isNaN(parseInt(query.class_age))) {
    filters['cl_age90'] = parseInt(query.class_age);
  } else {
    filters['cl_age90'] = 0;
  }

  if (query.since && query.to) {
    filters['jour'] = {
      $gte: new Date(query.since),
      $lt: new Date(query.to),
    };
  } else if (query.since) {
    filters['jour'] = {
      $gte: new Date(query.since)
    };
  } else if (query.to) {
    filters['jour'] = {
      $lt: new Date(query.to)
    };
  }

//[… Skip des parties que je n'ai pas faites …]


  return filters;
}
```

Cette fonction permet de récupérer les données agrégées par jour selon différents filtres. Par exemple, si aucun filtre n'est passé, elle va renvoyer les données de toutes les régions cumulées (donc de la France) par jour, tout ça en une seule requête mongodb ! Et si l'utilisateur nous passe des filtres, c'est la même requête mongodb qui est utilisée ! Ce qui rend cette fonction très polyvalente.

De plus, la fonction createFilters, permet de rendre très facilement intégrable et utilisable un nouveau filtre car il suffit simplement de modifier cette fonction.

## Commenter une fonction ou un composant (max 100 lignes) que vous avez écrit et qui mériterait une optimisation ou amélioration

```
export const Filters = ({
  classAgesProps, refetch, geolocation,
}) => {
//[… Skip des parties que je n'ai pas faites …]

  const onSubmit = (data) => {
    data.since.setHours(0, 0, 0);
    data.to.setHours(0, 0, 0);

    refetch({
      queryParams:
      {
        ...(currentRegion !== '00') && { reg: currentRegion },
        since: `${data.since}`,
        to: `${data.to}`,
        class_age: `${data.classAge}`,
      },
    });
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <Grid container justify="space-around" className={classes.grid}>

        <FormControl className={classes.formControl}>
          <InputLabel id="class-age-label">Classe d&apos;âge</InputLabel>
          <Controller
            as={Select}
            id="class-age"
            name="classAge"
            labelId="class-age-label"
            control={control}
            defaultValue={classAges[0]}
            onChange={handleChange}
          >
            {classAges.map((classAge) => (
              <MenuItem key={classAge} value={classAge}>{classAge}</MenuItem>
            ))}
          </Controller>
        </FormControl>

//[… Skip des parties que je n'ai pas faites …]

        <MuiPickersUtilsProvider locale={frLocale} utils={DateFnsUtils}>
          <Controller
            as={KeyboardDatePicker}
            disableToolbar
            name="since"
            control={control}
            defaultValue={sinceDate}
            value={selectedSinceDate}
            variant="inline"
            format="dd/MM/yyyy"
            id="since-date-picker"
            label="Depuis"
            onChange={handleSinceDateChangeDate}
            KeyboardButtonProps={{
              'aria-label': 'change date',
            }}
          />
```

```
        </MuiPickersUtilsProvider>

        <MuiPickersUtilsProvider locale={frLocale} utils={DateFnsUtils}>
          <Controller
            as={KeyboardDatePicker}
            disableToolbar
            name="to"
            control={control}
            defaultValue={selectedToDate}
            value={selectedToDate}
            variant="inline"
            format="dd/MM/yyyy"
            id="to-date-picker"
            label="Jusqu'au"
            onChange={handleToDateChangeDate}
            KeyboardButtonProps={{
              'aria-label': 'change date',
            }}
          />
        </MuiPickersUtilsProvider>

        <Button type="submit" color="secondary">Envoyer</Button>

      </Grid>
    </form>
  );
};
```

Ce composant mériterait une révision suite à l'utilisation des Controller et de l'utilsation de React Hook Form qui n'est pas indispensable.

J'ai utilisé les Controller de react-hook-form, afin de pouvoir le faire fonctionner avec les composants DatePicker de material UI. Je découvrais cette technologie et ayant déjà passé pas mal de temps à comprendre et à résoudre les problèmes de compatibilité, je n'ai pas pris le temps de bien regarder d'autres alternatives au Controller.

De plus, l'utilisation de React-hook-form n'est pas utile dans ce cas-là, car nous n'utilisons pas vraiment les mécaniques intéressantes de celui-ci. On pourrait très bien utiliser un Simple Form Material UI et se passer des Controller.