Louis GREINER - William GUILBAUD - Loïc HUSSON - Benjamin MISSAOUI

SR03 **Devoir 4**

Application web avec React

DÉVELOPPEMENT DU PORTAIL ADMIN DE L'APPLICATION WIKILOGIE

P21

TABLE DES MATIÈRES

1. INTRODUCTION

L'objectif de ce projet est de se familiariser et d'approfondir les concepts de développement web et du framework React. Il consiste pour cela à développer et mettre en place une interface web dynamique pour l'association Wikilogie en dialoguant avec une base de données relationnelle PostgreSQL.

Wikilogie est un portail développé par des étudiants du GI de l'UTC dont la vocation est de promouvoir les principes de base de l'ingénierie soutenable. Notre travail a consisté à développer la partie client du portail administrateur de Wikilogie, qui doit permettre d'afficher, d'ajouter, de modifier ou de supprimer le contenu de la base de données.

1.1 MAQUETTE

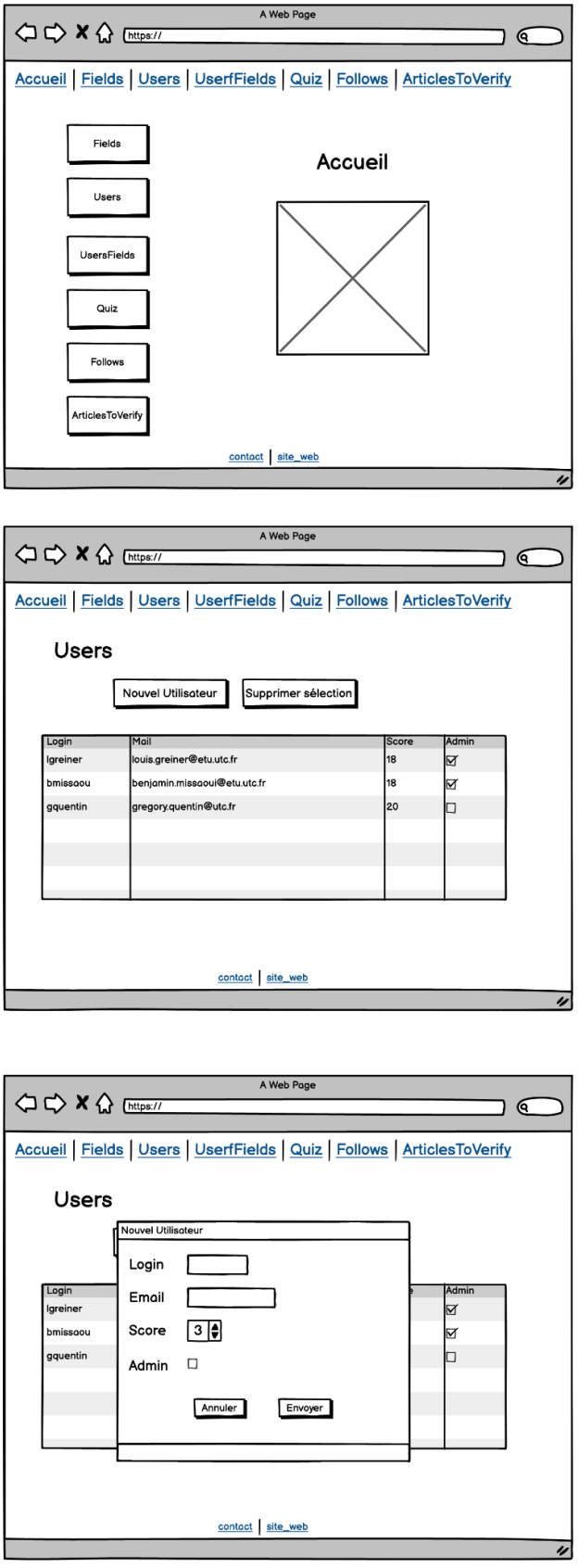


Fig 1 : Maquette du site web

2. FONCTIONNALITÉS DE L'APPLICATION

Dans cette partie, nous détaillerons l'interface utilisateur que nous avons implémenté pour répondre aux besoins du portail Administrateur de Wikilogie.

L'application web est composée d'une page d'accueil et de plusieurs pages de visualisation des données de la base de données.

La page d'accueil de l'application est composée d'un logo et de plusieurs boutons de redirection vers les autres pages de l'application. On y retrouve également :

- un Footer, qui permet de contacter l'association par mail ou de rediriger vers son site web.
- un Header, qui permet d'accéder facilement aux autres pages de l'application.

Ces deux éléments sont présents durant toute la navigation sur le site.

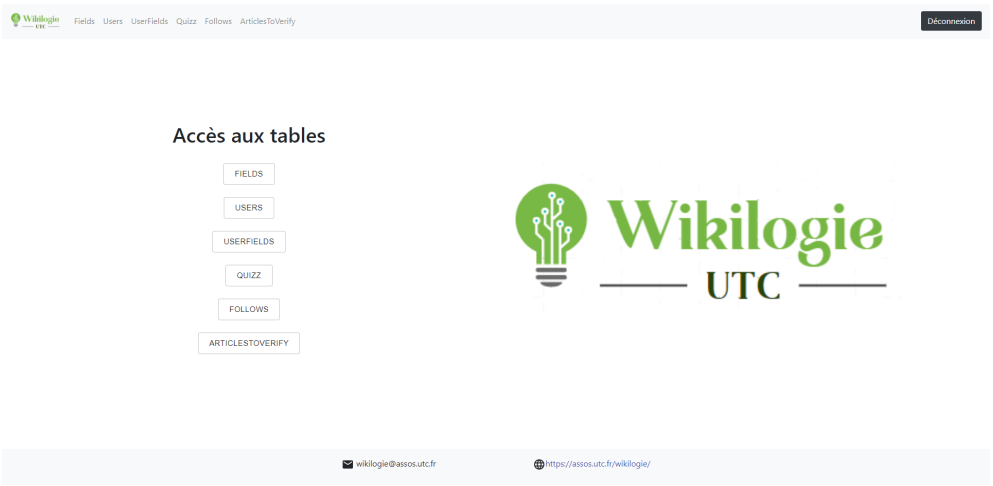


Fig 2 : Page d'accueil - http://localhost:3000/Accueil

Les autres pages de l'application correspondent, quant-à-elle, chacune à une table de la base de données.

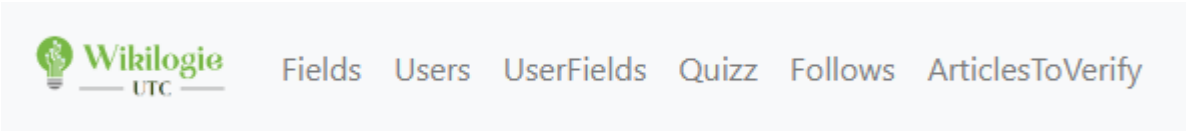


Fig 2 : Nav Bar

Elles sont constitués d'un tableau permettent de réaliser les fonctionnalités centrales du cahier des charges :

- consulter les entrées de la base;
- insérer, modifier et supprimer des entrées;
- trier, filtrer et sélectionner les données pour obtenir des représentations plus affinées.

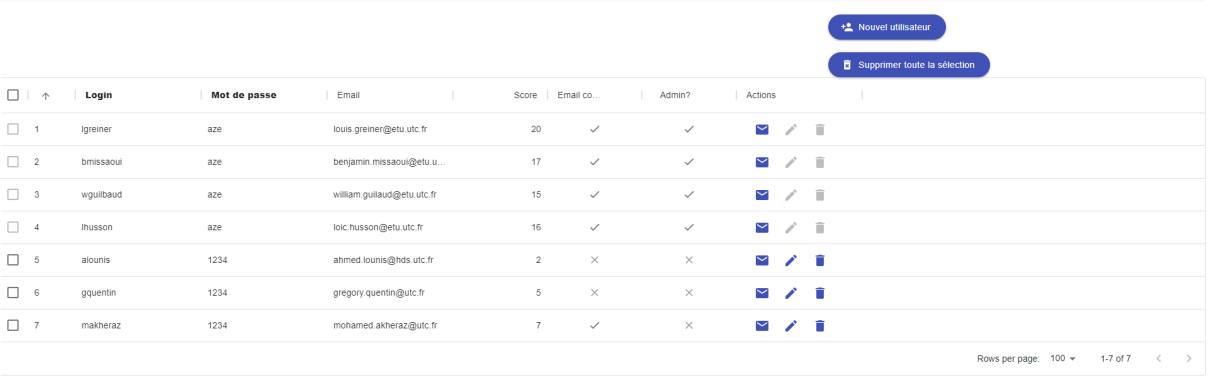
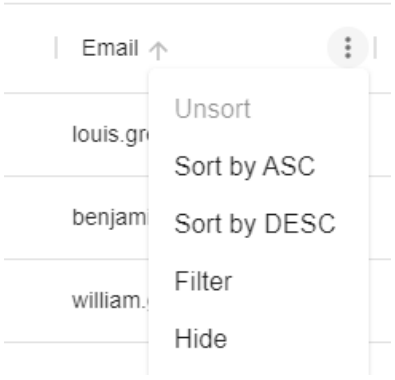


Fig 3 : Exemple de table (ici table Users - http://localhost:3000/Users)

Ce tableau est un DataGrid implémenté à partir de la librairie Material-UI de React. A partir de ce dernier, l'utilisateur peut facilement filtrer et trier les données ainsi que choisir les colonnes de la table visible dans le tableau.



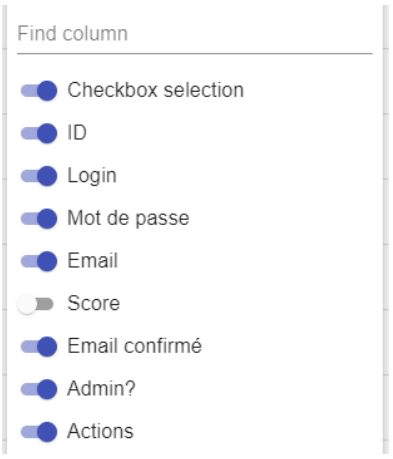


Fig 4&5 : Tri et filtre à l'aide de DataGrid

Sur chaque table, il est possible d'insérer de nouvelle données en cliquant sur le bouton d'ajout et en remplissant un formulaire. En validant ce dernier, une nouvelle ligne sera ajoutée dans le tableau et une nouvelle entrée sera insérée dans la base de données correspondante.

Fig 6 : Fenêtre ajout (ici Users)

A l'inverse, il est également possible de supprimer les données que l'on souhaite. L'utilisateur peut pour cela, appuyer sur l'icône de suppression présent à la fin de chaque ligne ou, s'il souhaite en supprimer plusieurs, les sélectionner et appuyer sur le bouton de suppression multiple en haut de la table. Selon la table, certaines données ne sont cependant pas supprimables comme par exemple les utilisateurs *administrateur* dans la table "User".

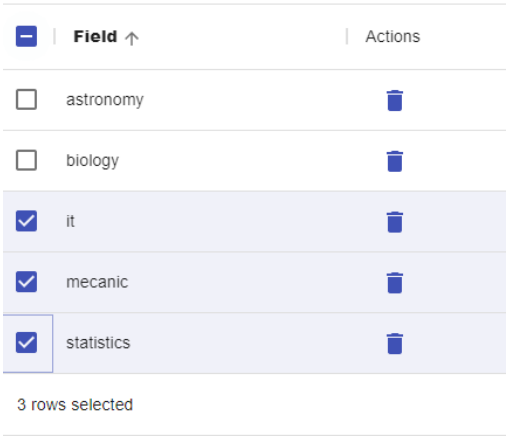


Fig 7 : Sélection de plusieurs champs

Enfin, nous avons implémenté la modification des éléments de la table. Pour toutes les colonnes dont le libellé n'est pas en gras et pour les lignes éditables, l'utilisateur peut modifier la valeur directement dans le tableau, avec un double-clic sur la cellule ou depuis le formulaire d'édition accessible depuis l'icône "éditer" (le crayon) de la colonne *action*.

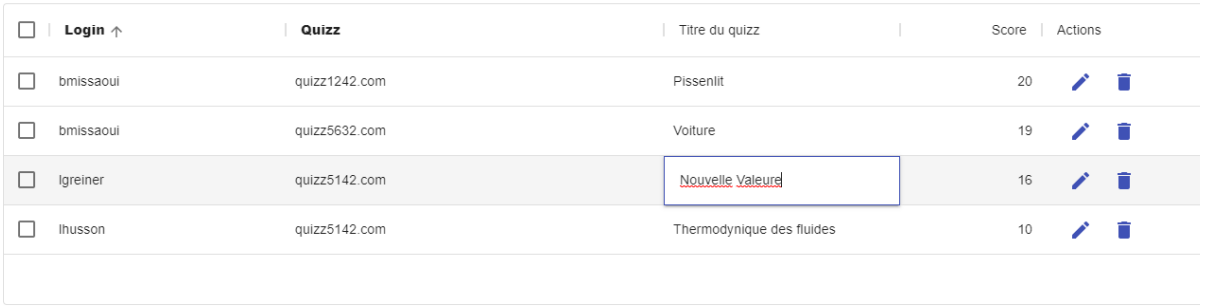


Fig 8 : Modification d'un champ directement depuis la table

3. CONSIDÉRATIONS ÉCO-RESPONSABLES

Cette application étant destinée à être déployée, il est primordial de prendre des précautions pour limiter son impact environnemental. Dans notre cas, les fonctionnalités les plus coûteuses sont toutes les opérations CRUD que l'application doit effectuer pour ajouter un utilisateur, modifier un quizz, supprimer un thème, etc…

Pour réaliser l'impact de notre application, nous avons d'abord simulé une utilisation normale, avec plusieurs requêtes sur différentes pages. Ensuite, nous avons utilisé l'EcoIndex de GreenIT (lien : ici) pour pouvoir évaluer l'application.

Nous avons alors obtenu un score de B. Celui-ci était originellement à A après quelques requêtes, puis est descendu à B lorsque nous avons continué d'effectuer des requêtes sur les différentes tables :

Fig 9: EcoIndex après une utilisation normale de l'application (càd après plusieurs requêtes sur plusieurs tables différentes)

Ainsi, comme dit précédemment, ce sont donc les requêtes HTTP qui diminuent notre score. Pour remédier à cela, nous nous sommes assurés de minimiser ce nombre de requêtes, en faisant attention à la construction de ces dernières.

3.1 MINIMISATION DU NOMBRE DE REQUÊTES

Tout d'abord nous nous sommes assurés que l'utilisateur ne puisse "supprimer toute la sélection" que si des utilisateurs sont sélectionnés. De cette manière, il ne peut pas déclencher une requêtes delete qui ne supprimerait aucun utilisateur de la BDD :

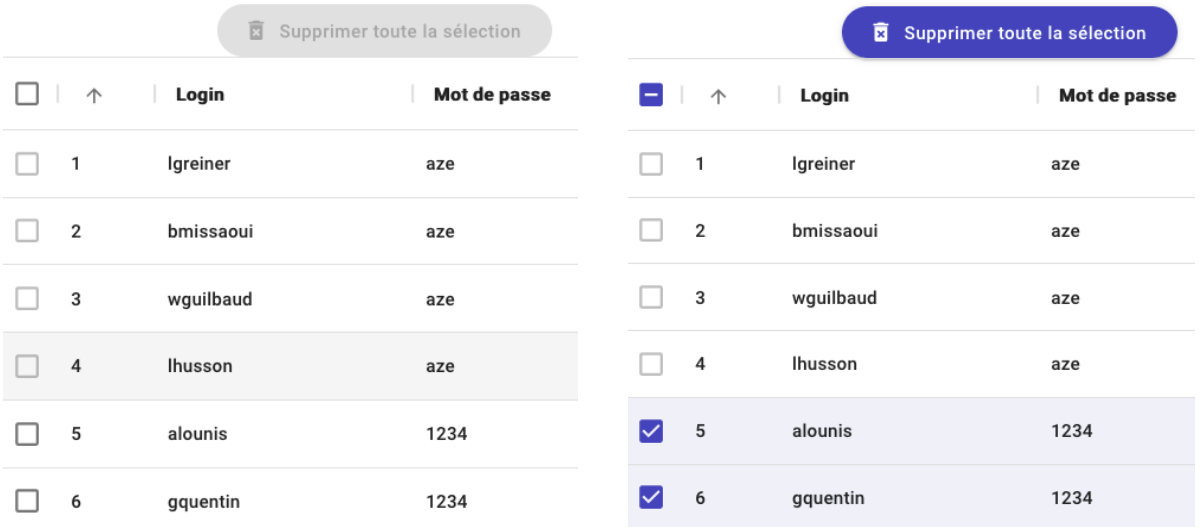


Fig 10: Désactivation du bouton lorsque la sélection est vide pour éviter un Delete inutile

Ensuite, lorsque l'utilisateur appuie sur le crayon pour modifier une ligne, un formulaire apparaît pour le laisser modifier les champs qu'il souhaite. Cependant, s'il envoie le formulaire sans avoir modifié aucun champ, alors nous n'envoyons pas de requêtes put car celle-ci serait inutile :

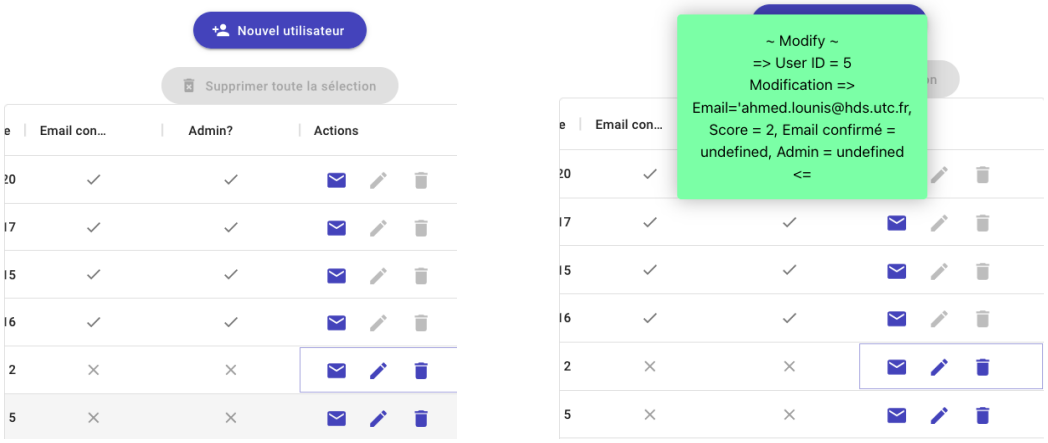


Fig 11: Envoi du formulaire d'édition lorsqu'aucun champ n'est modifié (gauche) et lorsqu'un champ est modifié (droite)

Aussi, pour toutes les requêtes modifiant plusieurs lignes ou plusieurs champs de la BDD, nous n'effectuons bien sûr qu'une seule requête et modifions l'ensemble des données simultanément :

```
// Supprime toutes les lignes sélectionnées
handleSubmitDeleteAll(){
  var request = "";
  for(var i = 0; i < this.state.selected.length - 1; i++){
    request += "id=" + this.state.selected[i] + " OR "
  }
  request += "id=" + this.state.selected[this.state.selected.length - 1];

  const requestOptionsDelete = {
    method: 'DELETE',
    headers: { 'Content-Type': 'application/json' },
    data: {
      tableName: 'users',
      whereConditions: request
    }
  };
```

“Supprimer toute la sélection” supprime toutes les lignes concernées en une seule requête

3.2 AUTRES AMÉLIORATIONS MISES EN PLACE

Au-delà de la minimisation du nombre de requêtes HTTP, nous avons aussi essayé, dans la mesure du possible, de suivre les bonnes pratiques recommandées par le collectif greenIT. Voici quelques exemples :

Bonne pratique	Mise en application
Éliminer les fonctionnalités non-essentielles	L’application se concentre sur les fonctionnalités essentielles voulues par le sujet. A l’origine, nos maquettes prévoyaient un widget d’activité sur la page d’accueil indiquant les derniers follows, les nouveaux quizz et scores obtenus, … Nous avons décidé d’abandonner cette fonctionnalité car non essentielle.
Créer un site responsive	Les éléments React et notamment material-ui sont responsive par défaut. La navbar, par exemple, devient déroulante si la fenêtre est trop étroite.
Créer une architecture applicative modulaire	Notre application est divisée en components pouvant être modifiés séparément et favorisant l’évolutivité. Il y a, entre autres, un composant navbar, footer, users, quizz, ...
Redimensionner les images en dehors du HTML	L’image de la navbar a été exportée directement à la bonne taille pour ne pas avoir à la redimensionner en CSS.
Supprimer tous les warning et toutes les notices	Tous les warnings non critiques ont été supprimés.

4. DIFFICULTÉS RENCONTRÉES

Malheureusement, nous avons rencontré de nombreuses difficultés, notamment avec la partie serveur, qui ont retardé le projet. Le fait qu'il n'y ait aucune donnée s'est montré contraignant à plusieurs reprises, car nous ignorions la fonction et les valeurs qui étaient supposés prendre certains champs (et les auteurs n'ont malheureusement pas répondu à nos questions).

Comprendre comment utiliser les fonctions de l'API a aussi pris du temps car il n'y avait pas d'exemples dans la plupart des cas. Nous avons donc créé un jeu de données implémenté dans la base de données exportée (init.sql).

Nous avons été obligés d'unifier la base (ajouter des contraintes de clés étrangères manquantes, des "DELETE … CASCADE", des types de données non cohérents) pour corriger des erreurs 500 côté client, bien que le sujet précisait que nous n'aurions que la partie client. A ce jour, il demeure des choix de conception que nous ne comprenons pas vraiment (et qui engendre encore des erreurs 500), comme :

- La triple clé primaire `"article_id", "author", "date_report"`, dans `articles_to_verify`, alors que le premier est un ID. Cela peut amener à des incohérences : un article avec le même ID et le même auteur peut être reporté à des dates différentes. De plus, l'UML indique que seul `article_id` est clé.

- L'utilisation du login ou de l'id pour référencer un user. Par exemple, dans `article_to_verify`, l'auteur de l'article est référencé par son login et l'admin qui vérifie l'article est référencé par son id. Cela complexifie légèrement la partie client en rendant le code moins générique.

- La syntaxe du routage avec `Express.js` est très différente de celle que nous avons trouvée le plus sur internet, ce qui a parfois ralenti le débogage des erreurs serveur. En effet , le manque de documentation sur la prise en main de l'API à utiliser a été une des principales pertes de temps.

- Le type de données `timestamptz` dans la base de données est difficilement conciliable avec son équivalent en JavaScript. Nous avons passé beaucoup de temps à pouvoir communiquer au serveur et au client ce type de données.

5. REPRISE DU PROJET

5.1 PROBLÈMES À RÉGLER

Dans l'optique d'une reprise de ce projet par un autre groupe le semestre prochain, ou par nous-même si à l'avenir nous en aurions le temps, il reste quelques fonctionnalités encore fragiles, et des problèmes plus critiques vis-à-vis du fonctionnement global de notre application :

- gestion des insertions avec clé primaires ou attributs `UNIQUE` déjà existants (`login`, `email`) car cela mène à des erreurs 500
- uniformisation du référencement user : soit grâce à `id` soit `login`
- uniformisation des types timestamptz entre le type `timestamptz` de postgresql et l'équivalent en JavaScript.
- disable le bouton valider des formulaires si il reste des champs vides (ou dynamiquement si une des entrées déjà rentrés est déjà présente dans la base de données)

5.2 FONCTIONS BONUS

Fonctionnalités bonus à implémenter selon le cahier des charges initial (qui ne correspond donc plus vraiment à notre application, car nous avons pris des directions différentes à celles proposées initialement) :

- Si des mots de passe ont fuité, forcer les utilisateurs à changer leur mot de passe (implémentation d'une fonctionnalité "Alerte Rouge" qui enverrait un mail à tous les utilisateurs présents dans la base Users. Pratique recommandée par l'ANSI.
- Générer une sauvegarde et restauration de la base de données (par exemple sauvegarde toutes les heures à l'aide de crontab).
- Création d'une table répertoriant les erreurs rencontrées par les utilisateurs.

Une dernière fonctionnalité à implémenter pourrait être un écran de connexion, avant de pouvoir accéder à la base de données, vérifiant ainsi que l'utilisateur qui tente d'y accéder est bien un administrateur de ce service.

6. PACKAGES ET PRISE EN MAIN

En plus des packages initialement installés, nous avons pris la liberté d'installer les packages installés par ces commandes :

```
npm install react-bootstrap
npm install rooter-react-dom
npm install datagrid
```

Le projet utilise postgresql. Les commandes suivantes permettent respectivement de (ré)initialiser la base de données avec le jeu de données que nous avons créé, et de se connecter à la base de données pour pouvoir y interagir directement en ligne de commande:

```
psql -h localhost -U postgres -p 5432 -q -f init.sql
psql -h localhost -U postgres -p 5432 -q
```

7. CONTACTS

Pour toute question relative au projet, contacter :

- Louis Greiner, louis.greiner@etu.utc.fr
- Benjamin Missaoui, benjamin.missaoui@etu.utc.fr
- William Guilbaud, william.guilbaud@etu.utc.fr
- Loïc Husson, loic.husson@etu.utc.fr