Développement d'une plateforme de gestion de ramassages de déchets

RAPPORT TECHNIQUE

Du 15 janvier au 05 avril 2024

Giuliana GODAIL-FABRIZIO

Entreprise : **Natural Solutions**

BUT Informatique, 3ème année

Tuteur en entreprise : **Vincent BOURGEOIS**, Développeur Full Stack

Tuteur IUT : **Corinne PATERLINI**

Sommaire

Introduction

L'application développée, appelée DepollutionMap, est une plateforme de gestion de missions de ramassages de déchets. Elle est destinée aux membres de l'association Wings of Ocean ainsi qu'aux visiteurs venus découvrir les missions de ramassages effectuées par l'association.

L'objectif de cette application est de faciliter la gestion des ramassages en réduisant le temps nécessaire à leur organisation et en évitant les erreurs fréquentes liées aux fautes de frappe ou aux omissions. Au-delà de cette fonctionnalité, la plateforme offre la possibilité d'explorer différents types de données grâce à une variété de graphiques. De plus, elle permet de gérer les informations relatives aux membres de Wings of Ocean, aux partenaires des ramassages, etc.

Ce rapport vise à faciliter la prise en main du projet et à fournir des informations détaillées pour d'éventuelles améliorations futures. Pour cela, nous commencerons par examiner les prérequis, puis nous aborderons l'arborescence du projet, et enfin, nous expliquerons certaines parties du code.

1 Prérequis

1.1 Installation préalable

Pour lancer l'application, Docker doit être installé sur votre ordinateur. Il s'agit d'un outil permettant de créer et d'exécuter des conteneurs logiciels légers. Ces conteneurs encapsulent toutes les dépendances nécessaires à l'application, simplifiant ainsi son déploiement et assurant une exécution cohérente sur divers environnements. Aucune autre installation n'est nécessaire, car les dépendances et autres éléments du projet seront installés directement dans les conteneurs Docker.

1.2 Structure du projet

L'application est composée de six parties qui interagissent étroitement :

- le **backend** récupère les données reçues du frontend et effectue le traitement approprié. Le backend s'exécute sur le serveur ;
- le **frontend** est l'interface par laquelle l'utilisateur passe pour effectuer une action (création d'un ramassage, visualisation de données…). Il collecte les données saisies et les transmet au backend pour qu'elles soient traitées de manière appropriée ;
- la **base de données PostgreSQL** contient toutes les informations nécessaires au bon fonctionnement de la plateforme de gestion de ramassages de déchets. Elle stocke les données telles que les détails de chaque ramassage. Elle peut être consultée via le frontend et modifiée par le backend ;
- la **base de données Minio** est utilisée pour stocker des images. Tout comme PostgreSQL, cette base de données peut être consultée via le frontend et modifiée par le backend ;

- **Hasura** est un serveur GraphQL qui simplifie les interactions avec la base de données PostgreSQL. Pour cela, il donne notamment accès à une interface graphique ;
- le **serveur** est l'infrastructure logicielle chargée d'héberger et de gérer l'application. Il assure le traitement des requêtes entrantes et sortantes, permettant ainsi l'exécution de l'application et la communication entre le frontend et le backend.

1.3 Contraintes techniques

Bases de données : PostgreSQL, Minio

API : Hasura (version 2.36.1)

Langages utilisés : GraphQL, JS, NodeJS, TypeScript

Template et dépendances utilisés :

1. **Serveur :**
    - **Next.Auth.js** (version 4.24.5) : une bibliothèque qui facilite l'implémentation de l'authentification en fournissant des fonctionnalités prêtes à l'emploi ;
    - **Next.js** (version 14.0.3) : un framework web pour Node.js utilisé pour la création des routes et la gestion des requêtes HTTP.
2. **Utilitaires ou fonctions de manipulation de données :**
    - **axios** (version 1.6.2) : une bibliothèque JavaScript utilisée pour effectuer des requêtes HTTP depuis le frontend ou le backend ;
    - **date-fns** (version 2.25.0) : une bibliothèque JavaScript pour la manipulation de dates et d'heures ;
    - **lodash** (version 4.17.21) : une bibliothèque utilitaire qui fournit des fonctions de manipulation de données.

3. **React-admin** (version 4.16.2) : un framework React pour la création rapide d'interfaces d'administration basées sur des API REST ou GraphQL.

4. **Design des interfaces utilisateur (UI) :**

    - **@mui/icons-material** (version 5.14.19) : une bibliothèque qui contient une collection d'icônes prêtes à l'emploi ;
    - **@mui/material** (version 5.14.20) : une bibliothèque qui fournit des composants réutilisables ;
    - **@mui/x-date-pickers** (version 6.19.3) : une bibliothèque qui fournit des composants de sélection de date et d'heure pour Material-UI ;
    - **Materio - Next.js Admin Template** : un template d'administration basé sur Next.js et Material-UI, fournissant une structure prête à l'emploi (utilisé pour les graphiques).

5. **Cartographie :**

    - **mapbox-gl** (version 3.0.0) : une bibliothèque JavaScript pour l'affichage de cartes interactives ;
    - **maplibre-gl** (version 3.6.2): une alternative open-source à mapbox-gl, offrant des fonctionnalités similaires pour l'affichage de cartes interactives ;
    - **react-map-gl** (version 7.1.6) : une bibliothèque React pour l'intégration de cartes interactives dans les applications web.

6. **Bibliothèques de développement et de configuration :**

    - **react** (version 18) : une bibliothèque JavaScript pour la construction d'interfaces utilisateur réactives ;
    - **react-dom** (version 18) : une bibliothèque utilisée pour le rendu des composants React dans le navigateur.

2 Arborescence

2.1 Globale

DepollutionMap est une application multi-conteneur qui utilise une architecture basée sur un monorepo.

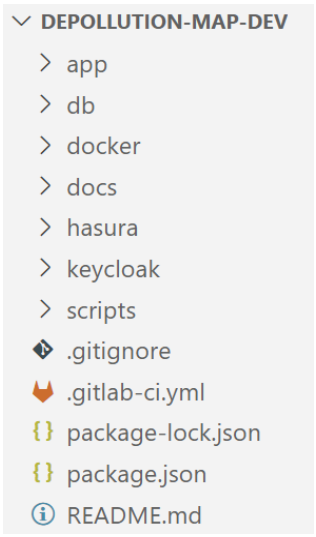


Figure I : Arborescence à la racine du projet

Le répertoire « app » rassemble à la fois la partie backend et frontend de l'application, constituant ainsi son noyau.

Dans « docker » se trouvent des fichiers YAML[1] qui décrivent et configurent les conteneurs nécessaires à l'exécution de l'application pour les environnements de développement ou de production.

Le dossier « hasura » regroupe des fichiers détaillant la structure de la base de données, ainsi que les instructions SQL pour la mettre à jour.

[1] YAML est un format de fichier texte utilisé pour échanger des données entre diverses applications.

Le répertoire « keycloak » contient la configuration de Keycloak, un gestionnaire d'identité et d'accès open source. Il englobe la définition des utilisateurs autorisés à accéder à l'application et de leurs rôles respectifs.

Le dossier « scripts » facilite l'automatisation des tâches liées au développement ou au déploiement de DepollutionMap. Il contient de nombreux scripts Shell (Bash), chacun dédié à une tâche spécifique.

Le fichier « .gitlab-ci.yml » est utilisé pour le CI/CD (Continuous Integration and Continuous Deployment) dont je parlerai ci-après.

2.2 Noyau de l'application

DepollutionMap est une application basée sur Next.js. Il s'agit d'un framework qui accélère et simplifie la création d'un site Web.

L'utilisation de cette technologie intègre un routage automatique basé sur la structure des fichiers du dossier « src ». Chaque fichier JavaScript ou TypeScript placé dans ce répertoire devient une route accessible dans l'application.

De plus, Next.js prend en charge le Server Side Rendering[2] tout en offrant des fonctionnalités du Client Side Rendering[3]. Cette approche permet de combiner les fonctionnalités du backend et du frontend dans une seule application, offrant ainsi une expérience de développement plus fluide et de meilleures performances.

---

[2] Le Server Side Rendering (SSR) consiste à générer le contenu de la page du côté du serveur avant de l'envoyer au navigateur. Cela signifie que lorsque l'utilisateur demande une page, le serveur exécute le code de l'application et génère le HTML correspondant, qui est ensuite renvoyé au navigateur.

[3] Le Client Side Rendering (CSR) consiste à modifier dynamiquement le contenu de la page sans la recharger en intégralité. Contrairement au SSR, c'est le navigateur qui se charge de l'exécution du code pour créer et mettre à jour dynamiquement le contenu de la page.

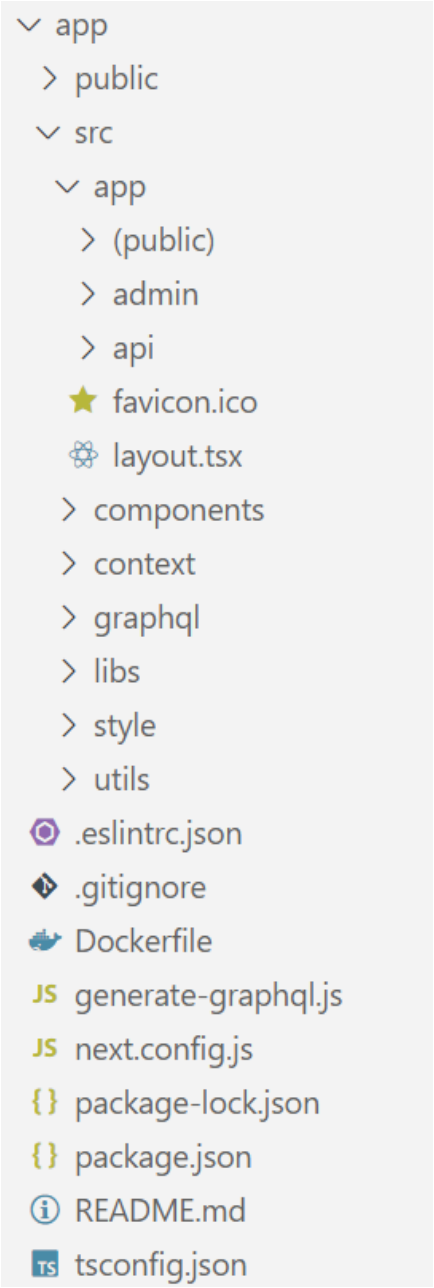


Figure II : Structure de « app »

Le répertoire « api » contient l'ensemble du code dédié au traitement des requêtes provenant de l'interface utilisateur. C'est l'API de l'application.

Les dossiers « src/app/(public) » et « src/app/admin » regroupent respectivement toutes les pages accessibles sans connexion (public) et avec (admin).

Le répertoire « components » réuni les composants utilisés dans l'interface utilisateur.

Dans « graphql » sont rassemblés des fichiers qui définissent des fonctions ou des constantes utilisées pour interagir avec l'API d'Hasura.

Le dossier « utils » contient un ensemble d'utilitaires.

Le Dockerfile est utilisé lors de la construction de chaque environnement. On en retrouve aussi un dans Hasura.

Le fichier « tsconfig.json » fournit des instructions au compilateur sur la manière de traiter le code source TypeScript.

2.3 Hasura

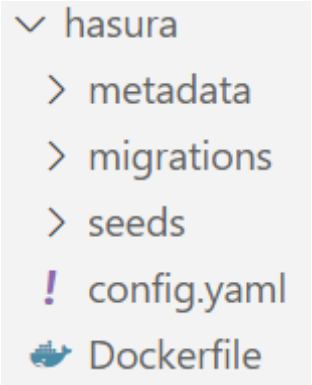


Figure III : Structure de « hasura »

Le dossier « metadata » comprend les métadonnées de l'application, telles que les schémas GraphQL décrivant la structure de la base de données, les autorisations pour chaque table, les relations entre celles-ci et d'autres configurations spécifiques à l'application.

Le répertoire « migrations » contient les instructions SQL nécessaires pour mettre à jour la structure de la base de données. Chaque migration représente un changement de schéma de base de données, tel que la création de tables ou de colonnes.

Le dossier « seeds » de Hasura contient des instructions SQL qui seront utilisées pour enrichir les tables de la base de données lors de l'initialisation ou de la mise à jour de l'application.

3 Explications techniques

3.1 Docker

Quand on lance la commande Shell « `docker-compose up` » le processus est le suivant pour l'environnement de développement :

- Docker Compose lit le fichier « docker-compose.yml » pour définir la configuration des services ;
- pour chaque service défini, il vérifie si l'image Docker correspondante existe localement et la télécharge si nécessaire ;
- il crée ensuite des conteneurs distincts pour chaque service et les démarre dans l'ordre spécifié dans le fichier « docker-compose.yml » ;
- les commandes définies dans les fichiers Dockerfile de chaque service sont exécutées pour construire les images des conteneurs.

Lorsqu'on est dans un environnement de production, le même processus est suivi à l'exception près que le fichier pris en compte est le « docker-compose.prod.yml ».

Dans les fichiers « docker-compose.yml » et « docker-compose.prod.yml » on retrouve une section pour chaque service :

- la base de données PostgreSQL ;
- la base de données Minio ;
- Hasura ;
- l'application Next.js.

Pour chacune de ces sections, on spécifie des informations telles que l'image Docker à utiliser, les variables d'environnement nécessaires, les ports utilisés, les volumes montés pour le stockage des données, et le nom du conteneur. Le nom du réseau auquel chaque conteneur appartient est également indiqué. Cela permet aux différents conteneurs de communiquer entre eux en utilisant ce réseau.

```yaml
version: "3.9"

# Définition de paramètres par défaut pour les services.
x-project-defaults: &project_defaults
  networks: # Définition du réseau auquel les services appartiennent.
    - project
  env_file: ./.env # Spécification du fichier .env contenant les variables d'environnement.

# Définition des services Docker.
services:
  db:
    <<: *project_defaults # Utilisation des paramètres par défaut définis précédemment.

    image: ${DB_IMAGE} # Image Docker utilisée pour le conteneur de la base de données.

    healthcheck: # Configuration du contrôle de santé du service.
      test: # Définition du test à exécuter.
        - "CMD-SHELL"
        - "pg_isready -U ${DB_USER} -d ${DB_DATABASE} -h 127.0.0.1"
      interval: 10s
      timeout: 5s
      retries: 3 # Nombre de tentatives avant de considérer le service comme non disponible.
      start_period: 60s # Délai d'attente après le démarrage du service.

    volumes:
      - db_data:/var/lib/postgresql/data # Montage du volume db_data pour stocker les données.
```

Figure IV : Extrait du fichier « docker-compose.yml »

Le projet comprend un total de trois Dockerfile qui décrivent les instructions, les dépendances et les configurations nécessaires à la création d'une image Docker spécifique.

```dockerfile
# 1. Installation des dépendances uniquement lorsque nécessaire.
FROM node:lts-alpine AS deps

# Définition du répertoire de travail.
WORKDIR /app

# Copie des fichiers package.json et package-lock.json.
COPY package*.json ./

# Installation des dépendances Node.js.
RUN npm install

# 2. Reconstruction du code source uniquement lorsque nécessaire.
FROM node:lts-alpine AS development

# Définition du répertoire de travail.
WORKDIR /app

# Copie de tous les fichiers du projet.
COPY --chown=node:node . .

# Construction du projet.
RUN npm run build

# Exposition du port 3000 pour la communication avec l'extérieur.
EXPOSE 3000

# Commande par défaut à utiliser lorsque le conteneur démarre.
CMD ["npm", "run", "dev"]
```

Figure V : Dockerfile de l'application Next.js

3.2 CI/CD

Le CI/CD est une approche qui permet d'augmenter la fréquence de distribution des applications grâce à l'introduction de l'automatisation au niveau des étapes de développement. L'intégration continue et le déploiement continu sont les principaux concepts du CI/CD. En effet, cette approche résout les problèmes liés à l'intégration de nouveaux segments de code, également connus sous le nom « d'integration hell » (l'enfer de l'intégration), rencontrés par les équipes de développement et de déploiement.

```yaml
stages:
  - build # Cette étape du pipeline est chargée de construire l'application.
  - deploy # Cette étape du pipeline est chargée de déployer l'application.

build_app:
  stage: build
  script: # Commandes utilisées pour construire l'image de l'application.
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER $CI_REGISTRY --password-stdin
    - docker build --target production -t ${CI_REGISTRY_IMAGE}:app-${CI_COMMIT_REF_NAME} ./app
    - docker push ${CI_REGISTRY_IMAGE}:app-${CI_COMMIT_REF_NAME}
  only:
    refs:
      - master # Le travail ne s'exécute que lorsque la branche « master » est modifiée.
      - dev # Le travail ne s'exécute que lorsque la branche « dev » est modifiée.
    changes:
      - app/**/* # Le travail ne s'exécute que lorsque des changements sont effectués dans le dossier « frontend ».
  tags:
    - NSCICDDOCKER # Balise spécifiant où exécuter ce travail de construction.
```

Figure VI : Mise en place du CI/CD

3.3 Base de données

3.3.1 Structure

Le MCD (Modèle Conceptuel de Données) comprend vingt-sept tables, chacune représentant une entité distincte dans la base de données. De plus, il contient deux vues[4].

L'application organise et gère les données en fonction des ramassages. Cela implique que toutes les tables de l'application doivent être liées, directement ou indirectement (via des clés étrangères), aux ramassages (cleanup).

[4] Une vue en SQL est une représentation d'une ou plusieurs tables dans une base de données. Elle est créée à partir d'une requête SQL qui extrait et transforme les données de ces tables selon les besoins spécifiques de l'utilisateur. Les vues permettent de simplifier les requêtes complexes, de masquer la complexité de la structure des données et de limiter l'accès aux données sensibles en fournissant une interface de consultation restreinte.

Ci-dessous, un extrait du MCD met en évidence les tables directement liées à celle des ramassages (cleanup). Deux de ces tables sont particulièrement importantes :

- « **cleanup_area** » : permet de stocker pour chaque ramassage des zones précises où les participants concentrent leurs efforts de nettoyage ;
- « **cleanup_form** » : est une table de liaison entre un ramassage et les données telles que la quantité, le poids et le volume de déchets collectés. Elle est consultée pour générer des statistiques.

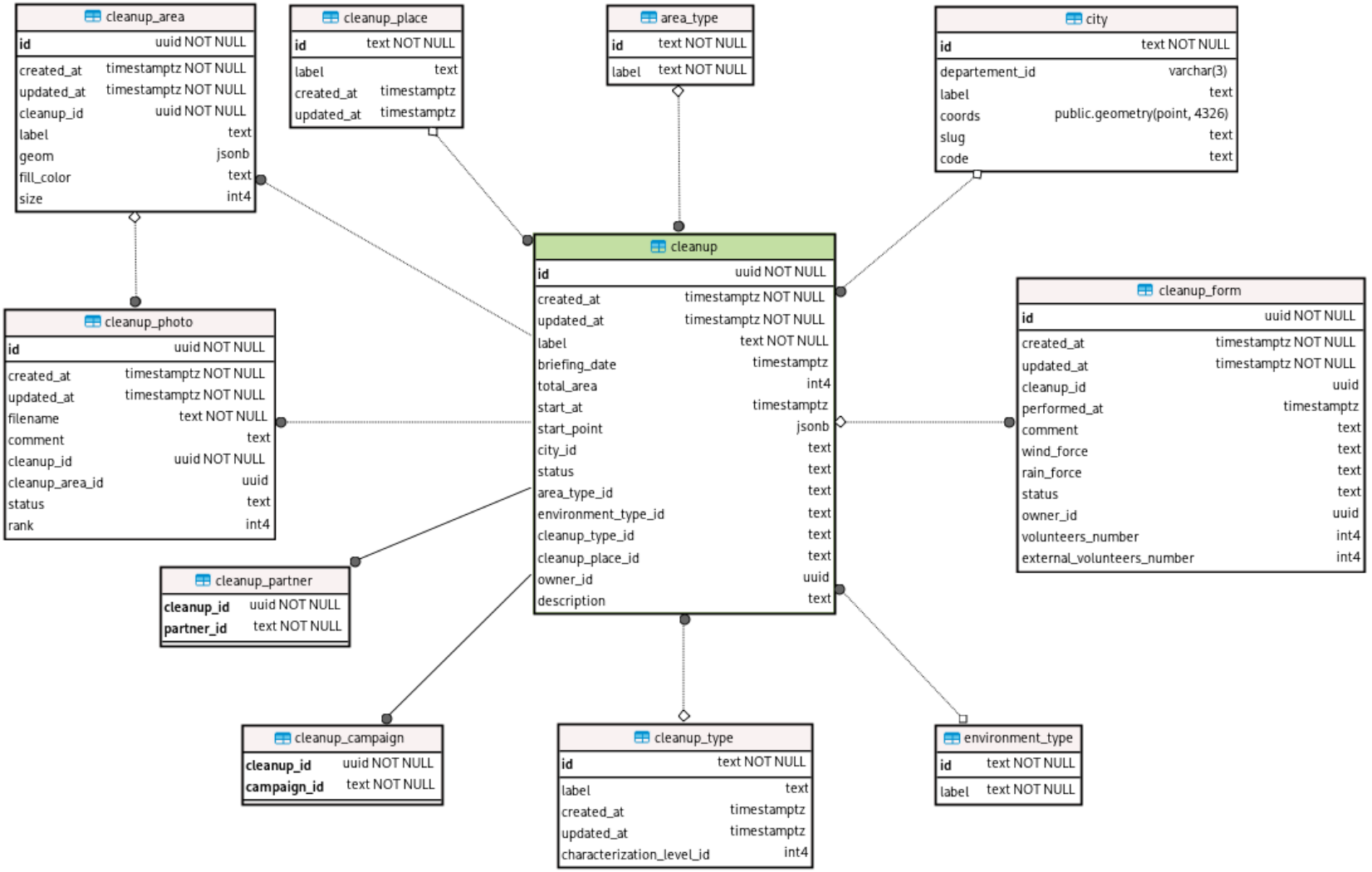


Figure VII : Extrait du MCD de l'application - Tables directement liées à la table des ramassages

3.3.2 Hasura

Pour rappel, nous utilisons le serveur GraphQL Hasura pour interagir avec la base de données. C'est une solution open source qui facilite la création, le déploiement et la gestion des API GraphQL. Par exemple, Hasura propose une interface graphique pour modifier la structure de la base de données et exécuter des requêtes GraphQL. De plus, cette solution permet de personnaliser le comportement de l'application et de gérer les autorisations d'accès aux données. Enfin, Hasura simplifie l'intégration du code développé par d'autres membres de l'équipe grâce aux migrations, des instructions SQL versionnées qui maintiennent la base de données à jour. Chaque modification crée une migration automatiquement intégrée chez les autres développeurs lorsqu'ils récupèrent la dernière version du code.

Pour lancer l'interface graphique d'Hasura, il faut :

- avoir démarré le conteneur Docker nommé « hasura » ;
- ouvrir un terminal à la racine du projet ;
- lancer la commande Shell suivante : « `npm run hasura:console` ».

L'exécution de cette commande ouvre automatiquement l'interface graphique d'Hasura dans la page Web du navigateur par défaut de la machine hôte.

3.3.3 Création d'une table

Pour créer une table, il faut se diriger dans l'onglet « SQL » de l'interface graphique d'Hasura.

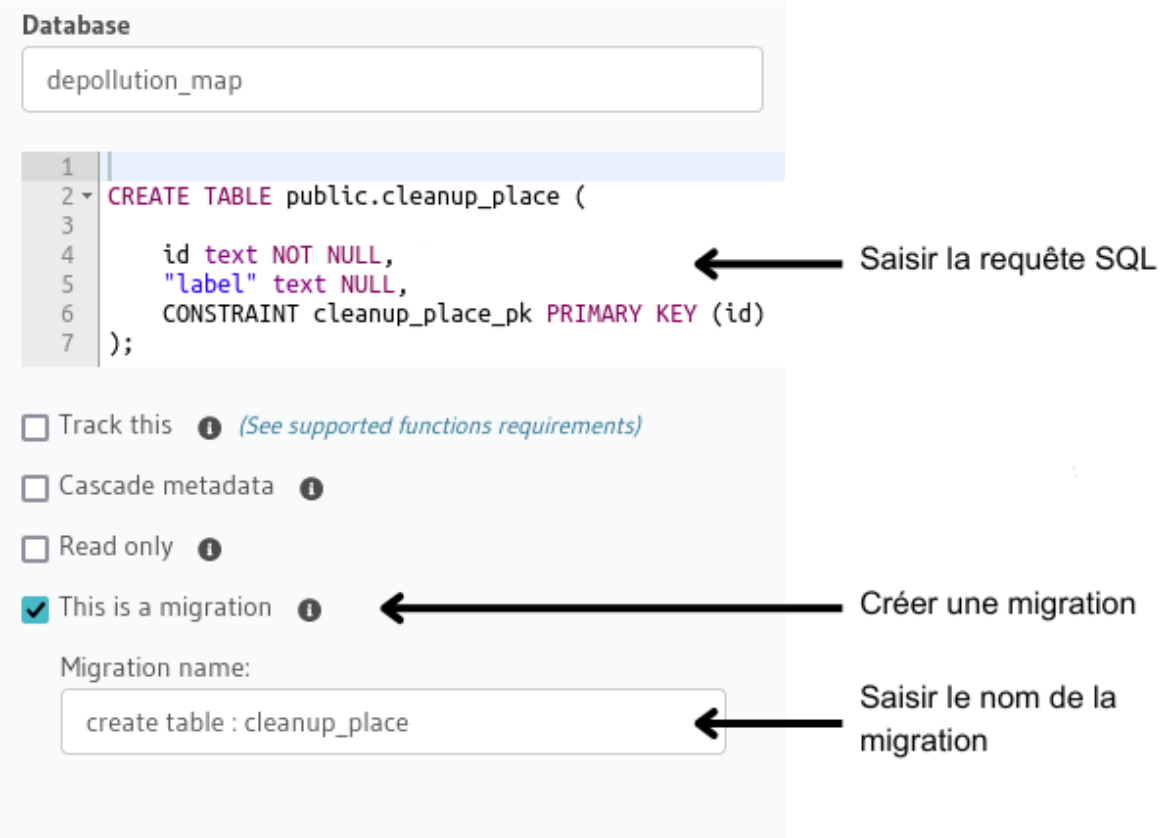


Figure VIII : Création de la table « cleanup_place »

Une fois l'action confirmée la table est ajoutée dans la base de données de la machine hôte et une migration est automatiquement créée. Cela se traduit par l'ajout d'un dossier dans le répertoire « migrations », nommé selon le nom attribué à la migration. Ce dossier se compose de deux fichiers :

- **« up.sql »** : contient les instructions SQL nécessaires pour réaliser la mise à jour ou la modification de la structure de la base de données, ici ces instructions créent la table « cleanup_place » ;
- **« down.sql »** : renferme les instructions permettant d'annuler les modifications apportées par « up.sql », ici ces instructions suppriment la table « cleanup_place ».

3.3.4 Permissions utilisateurs

Il existe **deux types d'utilisateurs** possibles dans l'application :

- l’utilisateur non connecté qui visite le site pour découvrir les missions réalisées par l'association (anonymous) ;
- l'utilisateur connecté qui peut être un super administrateur (admin) ou un utilisateur également membre de l’association mais ayant des privilèges restreints (editor).

Les ressources et les actions accessibles varient en fonction du rôle de l'utilisateur, qu'il soit administrateur, éditeur ou anonyme. Il est donc crucial de configurer les droits associés à chaque rôle pour chaque table de la base de données.

Cette gestion des permissions est effectuée via Hasura et doit être ajustée à chaque expansion de la base de données. Cela permet de garantir que chaque utilisateur a un accès approprié aux informations. Par exemple, pour une table, un utilisateur peut avoir le droit de lire toutes les données, mais seulement de modifier certaines d'entre elles, tandis qu'un autre utilisateur ne pourra que les lire.

Pour implémenter cette solution, il faut cocher pour chaque table les droits de création, de modification et de suppression pour chaque rôle. Lors de cette opération, les fichiers détaillant les propriétés de chaque table ainsi que les droits d'accès des utilisateurs, se mettent automatiquement à jour.

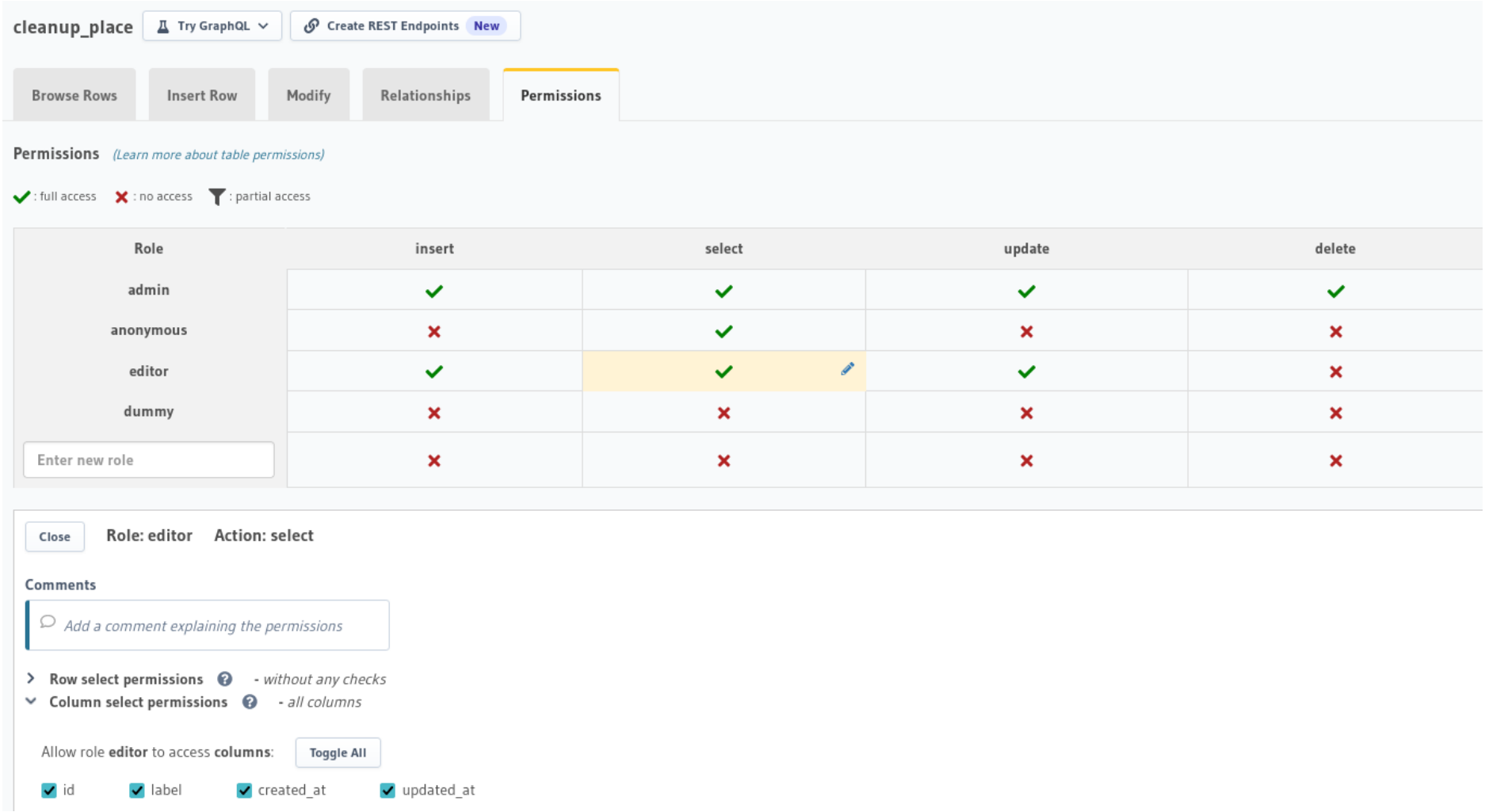


Figure IX : Gestion des droits des utilisateurs pour la table « cleanup_place »

3.4 React Admin

3.4.1 Présentation

L'**interface administrateur** est principalement générée grâce au framework React Admin.

L'utilisation de cet outil simplifie la création des pages destinées aux administrateurs. En effet, React Admin permet une interaction fluide avec les requêtes GraphQL fournies par Hasura. Cette combinaison simplifie la récupération et la modification des données, réduisant ainsi la charge de développement en évitant de se focaliser sur la gestion des données.

React Admin offre un accès à une multitude de composants qui exploitent la bibliothèque Material UI, assurant ainsi l'ergonomie et la simplicité d'utilisation de l'application. Chaque composant de React Admin prend en paramètre le nom d'une table de la base de données. Pour chaque table renseignée, il est capable de réaliser automatiquement les opérations CRUD (Création, Lecture, Mise à jour, Suppression), à condition que ces requêtes n'impliquent pas de liaisons avec d'autres tables.

3.4.2 Exemple d'utilisation

Pour afficher les données d'une table dans l'interface administrateur, nous devons d'abord ajouter un nouvel onglet dans le menu. Pour cela, on commence par créer un répertoire dans le dossier *« app/src/components/admin »* et on y ajoute un fichier contenant le code nécessaire pour afficher les données de la table concernée.

```tsx
import { useSessionContext } from "@/utils/useSessionContext";
import React, { FC } from "react";
import {···
} from "react-admin";

const filters = [<TextInput key="a" label="Lieu" source="label" />];

type AdminCleanupPlaceListProps = {};

export const AdminCleanupPlaceList: FC<AdminCleanupPlaceListProps> = () => {
  const sessionCtx = useSessionContext();
  const isAuthorized = sessionCtx.getIsAdmin();

  return (
    <List filters={filters}>
      <Datagrid isRowSelectable={() => isAuthorized || false}>
        <TextField source="id" />
        <TextField source="label" label="Lieu" />
        <DateField source="created_at" label="Créé le" />
        <DateField source="updated_at" label="Modifié le" />
        {isAuthorized && <EditButton />}
      </Datagrid>
    </List>
  );
};
```

Figure X : Code pour afficher les éléments de la table « cleanup_place »

Ensuite, pour **rediriger les utilisateurs** vers cette nouvelle page, on ajoute l'élément « Item » au menu personnalisé dont le chemin d'accès est « *app/src/components/admin/navigation/AdminMenu.tsx* ». Dans le fichier « *app/src/components/r_admin/ResourcesAdmin.tsx* », on intègre le composant « Resource » de React Admin dont les principaux paramètres sont :

- name : le nom de la ressource, correspondant généralement au nom de la table dans la base de données ;
- list : le composant à utiliser pour afficher la liste des éléments de la table ;
- edit : le composant à utiliser pour modifier les données de la ressource ;
- create : le composant à utiliser pour créer de nouveaux éléments de la ressource.

```tsx
// Définition des types de props attendues par le composant ResourcesAdmin.
interface RAdminProps {
  dataProvider: DataProvider; // Fournisseur de données pour React Admin.
  i18nProvider: I18nProvider; // Fournisseur de traduction pour React Admin.
}

// Définition du composant ResourcesAdmin.
export const ResourcesAdmin = (props: RAdminProps) => {
  const { dataProvider, i18nProvider } = props;

  // Vérifier les droits de l'utilisateur.
  const sessionCtx = useSessionContext();
  const isAuthorized = sessionCtx.getIsAdmin();

  return (
    // Le layout spécifié est construit via le menu personnalisé.
    <Admin layout={AdminLayout} dataProvider={dataProvider} i18nProvider={i18nProvider}>

      <Resource
        // Nom de la table dans la base de données.
        name="cleanup_place"

        // Composant utilisé pour afficher la liste des éléments de la table.
        list={AdminCleanupPlaceList}

        // Composant pour modifier et créer des éléments, basé sur les droits de l'utilisateur.
        edit={isAuthorized ? AdminCleanupPlaceEdit : undefined}
        create={isAuthorized ? AdminCleanupPlaceCreate : undefined}
      />
    </Admin>
  );
};
```

Figure XI : Extrait du fichier « ResourcesAdmin.tsx »

Lorsqu'un utilisateur accède à une page renseignée dans le fichier « ResourcesAdmin.tsx », React Admin utilise les informations fournies dans la déclaration de la ressource pour effectuer les requêtes GraphQL appropriées à la base de données via Hasura. Par exemple, si nous sommes dans une page d'affichage (list), React Admin récupère les informations de la table spécifiée par le paramètre « name ». De même, si nous sommes dans un formulaire de modification (edit), React Admin envoie une requête pour modifier les données correspondantes dans la base de données.

3.5 Dashboard

Le tableau de bord de l'application est accessible à tous les utilisateurs. Il offre la possibilité de visualiser des statistiques générées à partir des données des **ramassages terminés**, telles que le poids, la quantité et le volume de déchets collectés. Un ramassage est considéré comme terminé lorsqu'un formulaire de caractérisation, nommé « cleanup_form », atteint le statut « published ».

3.5.1 Ajout des filtres dans l'URL

Sur cette page, les utilisateurs peuvent utiliser des filtres pour affiner les résultats. Chaque fois qu'un filtre est sélectionné, il est automatiquement intégré à l'URL. Cette approche garantit que, lorsque l'utilisateur recharge la page, les filtres précédemment sélectionnés sont toujours appliqués. De plus, elle facilite le partage du lien du dashboard filtré avec un collègue, évitant ainsi à ce dernier de devoir appliquer les mêmes critères de recherche.

Pour ajouter les filtres dans l'URL, nous utilisons le code suivant :

```
const handleFilter = (filters: { [key: string]: any }) => {
  setAddFilter(false);

  // Vérifier si des filtres ont été fournis.
  if (filters) {
    // Supprimer le paramètre « filters » des paramètres existants de l'URL.
    existingParams.delete("filters");

    // Remplacer l'URL actuelle par une nouvelle URL avec les nouveaux filtres.
    router.replace(
      `${baseUrl}?filters=${JSON.stringify(
        filters
      )}&${existingParams.toString()}`
    );
  }
};
```

Figure XII : Fonction pour ajouter des filtres à l'URL

3.5.2 Mise en place du dashboard

Au chargement initial de la page, on interroge la base de données pour récupérer tous les ramassages terminés. Ces ramassages sont stockés dans un tableau « A » réutilisé par la suite.

```
query GetCleanups {
  cleanup(
      where: {
          // Ne sélectionner que les ramassages dont la date antérieur au jour actuel.
          start_at: { _lt: today },
          // Ne sélectionner que les ramassages dont le formulaire est "published".
          _and: { cleanup_forms: { status: { _eq: "published" } } }
      }
    ) {
      // Informations récupérées.
      id
      label
      start_at
      start_point
      cleanup_campaign { campaign_id }
      cleanup_forms { id, performed_at }
      cleanup_type {
          id
          label
          characterization_level_id
      }
      cleanup_partners { partner_id }
      city {
          id
          label
          departement_id
          departement { region_id } // Equivalent d'un inner join en SQL.
      }
    }
}
```

Figure XIII : Requête pour récupérer les ramassages terminés

On applique ensuite les filtres de l'URL à ce tableau « A » et on stocke le résultat dans un tableau B. On extrait les **identifiants** des ramassages du tableau « B » et on les transmet à l'API.

L'API va alors faire les requêtes et les calculs nécessaires afin de récupérer les données spécifiques des ramassages filtrés (poids, quantité et volume total de déchets collectés).

```
useEffect(() => {
  (async () => {
    // Récupérer les identifiants des ramassages filtrés.
    const cleanups_id = tableau_B.map((cleanup: any ) => cleanup.id);

    // Effectuer une requête à l'API pour obtenir les données à afficher.
    const resp = await api.get(`/dashboard?ids=${cleanups_id.join(",")}`);

    // Extraire les données spécifiques de la réponse de l'API.
    const respBasicStats = resp.data.basicStats;
    const respRubbishPerCampaign = resp.data.rubbishPerCampaign;
    const respRubbishPerCategory = resp.data.rubbishPerCategory;
    const respMainRubbish = resp.data.mainRubbish;

    // Calculer les séries de données pour les graphiques.
    const respSeries = Object.entries(respRubbishPerCampaign.series);
    const totalPerMission = respRubbishPerCampaign.stack.map(
      (_: any, index: number) => {
        return respSeries
          .map((item: any) => item[1][index] || 0)
          .reduce((acc, val) => acc + val);
      }
    );
    const series = respSeries.map((item: any, index) => {
      const data = item[1].map((i: number, id: number) =>
        round((i * 100) / totalPerMission[id], 1)
      );
      return { name: respRubbishPerCategory.data[index].label, data: data };
    });

    // Mettre à jour les variables utilisées pour l'affichage avec les données récupérées.
    setBasicStats(respBasicStats);
    setRubbishPerCampaign({
      xaxis: respRubbishPerCampaign.stack,
      series: series,
    });
    setRubbishPerCategory(respRubbishPerCategory);
    setMainRubbish(respMainRubbish);
  })();
}, [tableau_B]); // Exécuter le code précédent lorsque les ramassages filtrés changent.
```

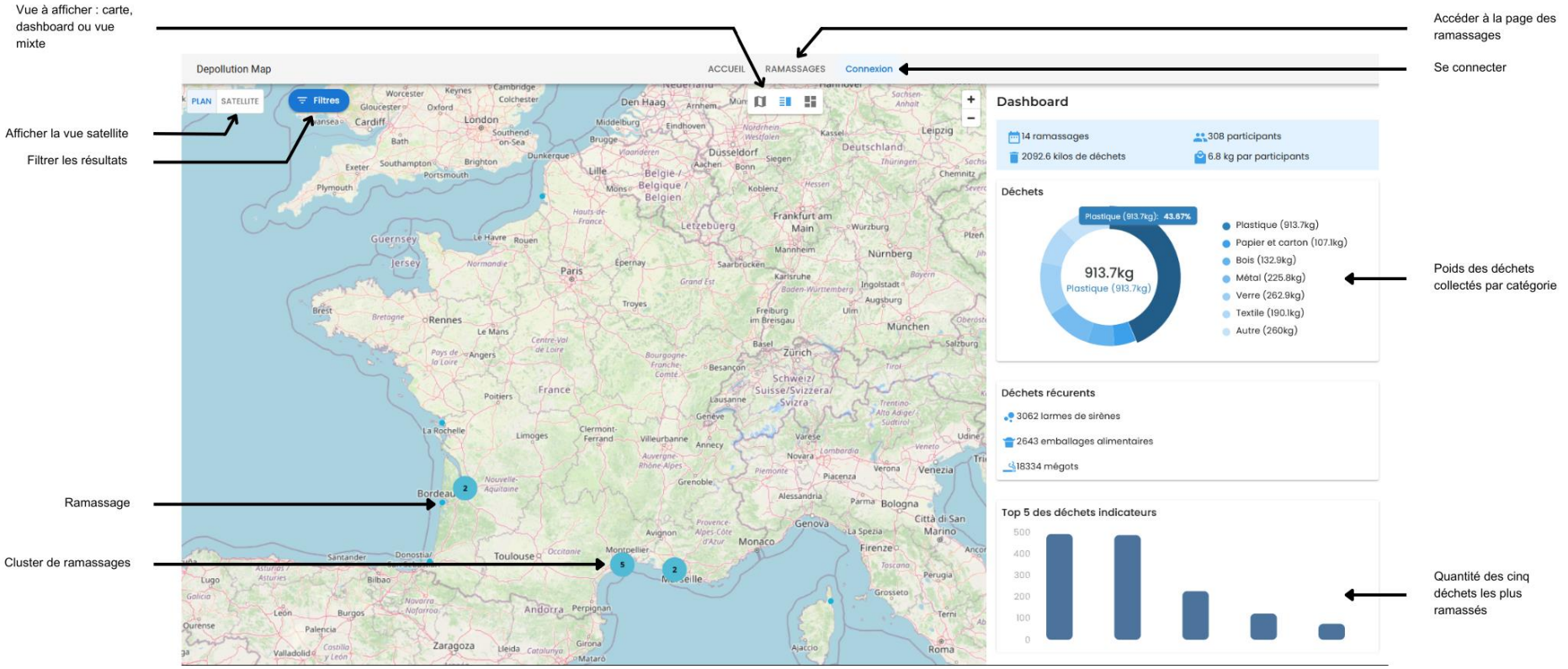
Figure XIV : Code pour interroger l'API

Enfin, les données renvoyées par l'API sont affichées sur le tableau de bord.

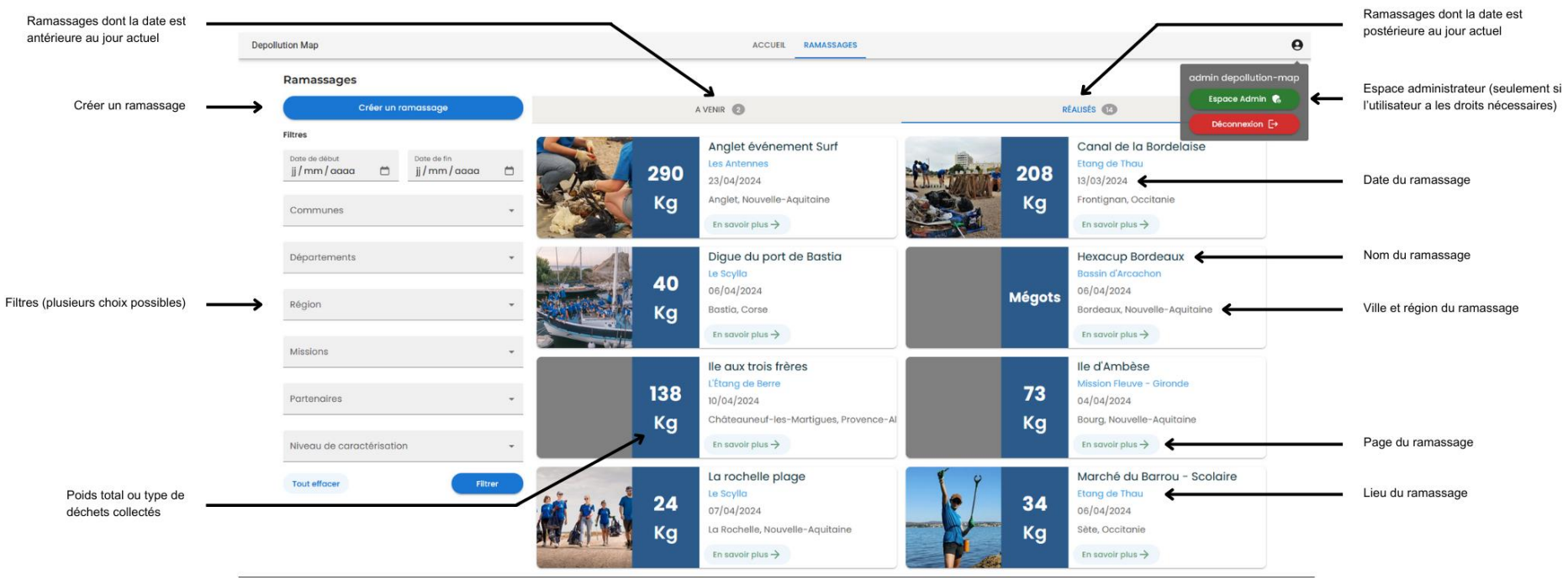
Ce processus se répète à chaque modification des filtres de l'URL, assurant ainsi une mise à jour dynamique des données affichées.

Table des illustrations

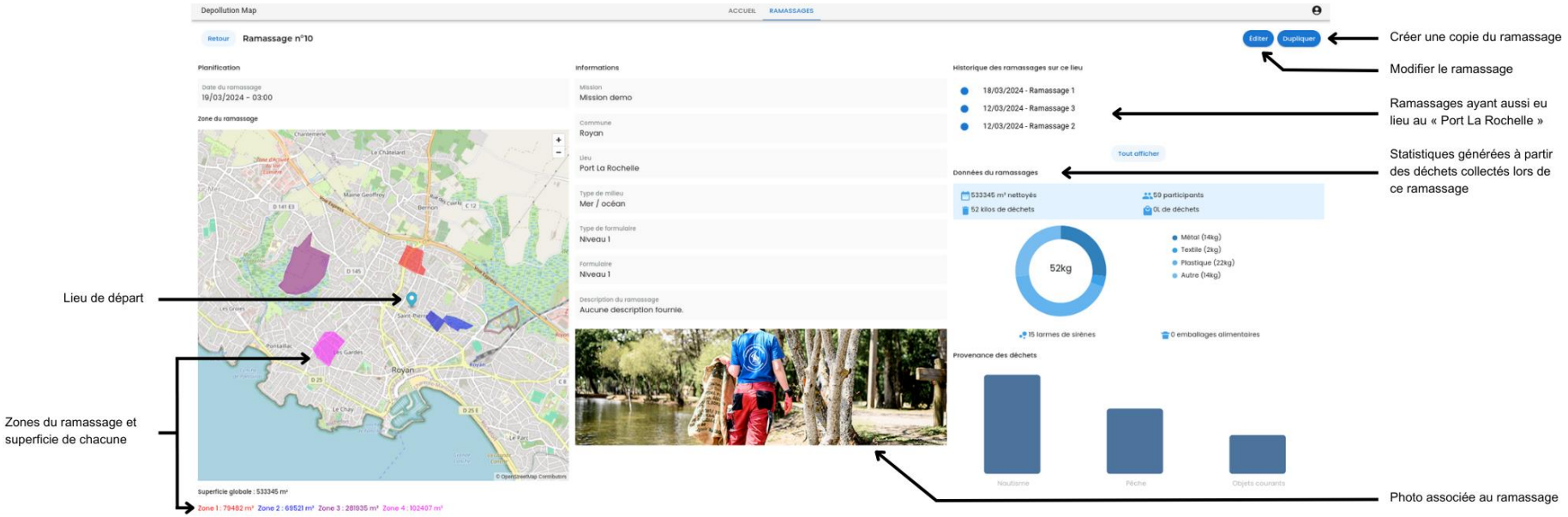
Tables des annexes



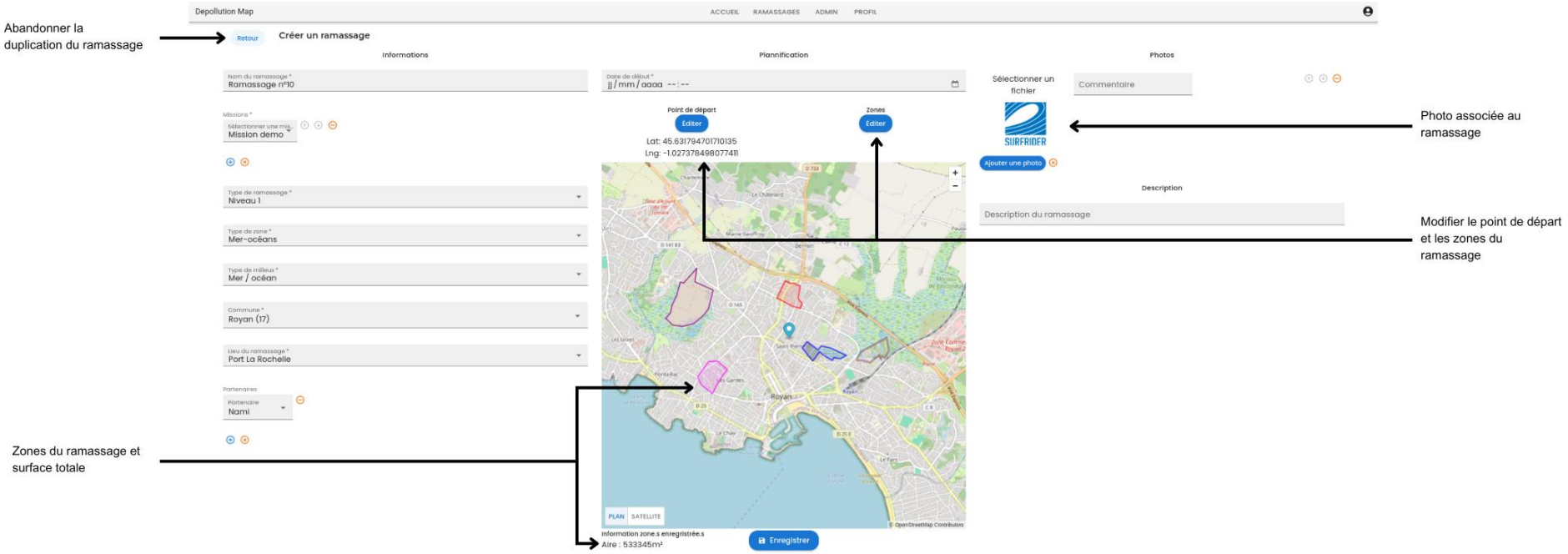
Annexe I : Page d'accueil



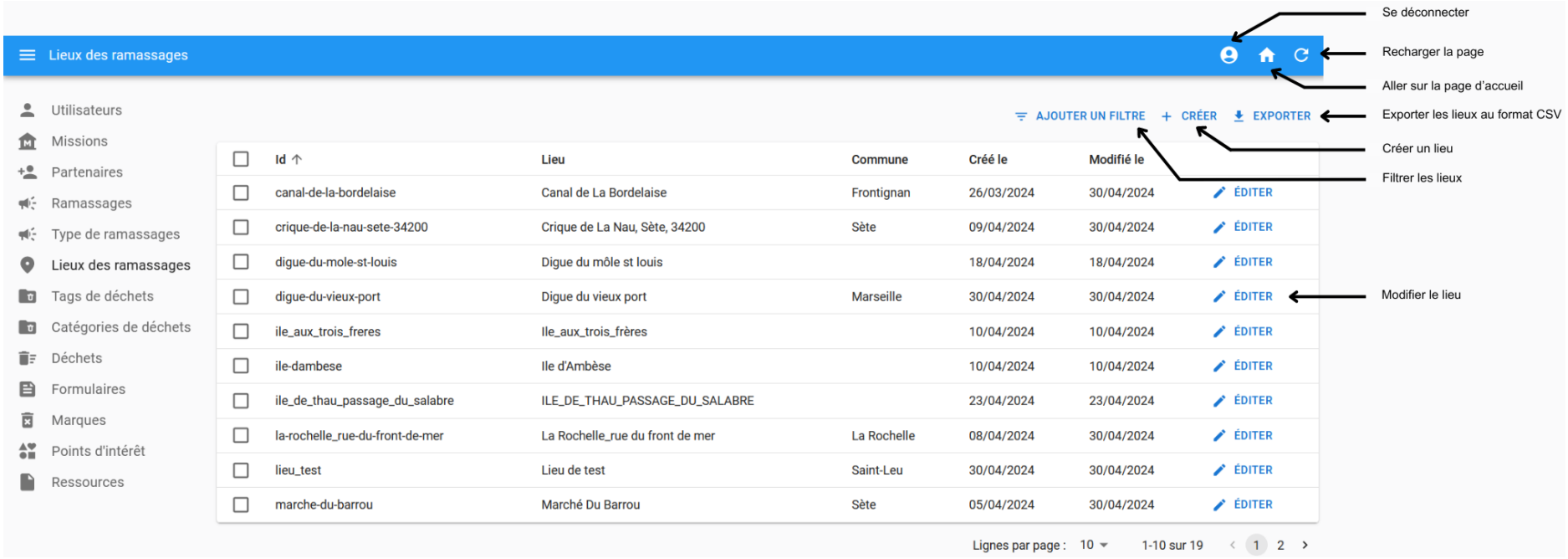
Annexe II : Page des ramassages (version connectée)



Annexe III : Page d'un ramassage (version connectée)



Annexe IV : Formulaire de duplication d'un ramassage



Annexe V : Interface administrateur