PROJET PROFESSIONNEL

GREENBIN

DÉVELOPPEUR WEB ET WEB MOBILE

Dossier de Projet – GREENBIN

Titre Professionnel : Développeur Web & Web Mobile (DWWM)
Projet réalisé en vue de la certification professionnelle

Présenté par :
Gilles Ruszczycki

Année : **2025**

Centre de formation : *CEF Centre Européen de Formation*

Session d'examen : *2025*

SOMMAIRE

1. Introduction & Présentation du projet

GreenBin est une application web permettant la **gestion des déchets** et des **utilisateurs** via une interface moderne, responsive et sécurisée.

Son objectif est d'offrir une solution simple et efficace permettant :

- d'identifier les types de déchets,
- d'organiser leur classification,
- de gérer les utilisateurs avec rôles (user / admin),
- d'assurer une cohérence et une sécurité des données.

Le projet sert de support à l'épreuve du **Titre Professionnel DWWM**, démontrant la maîtrise :

- du front-end moderne (React),
- du back-end sécurisé (Node.js + Express),
- d'une base de données relationnelle (MySQL),
- d'une architecture MVC,
- d'un système d'authentification JWT,
- et du responsive design.

2. Analyse du besoin et objectifs

Besoins fonctionnels :

- Connexion via email + mot de passe.
- Accès au tableau de bord après authentification.
- Gestion des déchets (CRUD complet).
- Gestion des utilisateurs (CRUD réservé admin).
- Différenciation claire des droits entre utilisateur simple et administrateur.

Besoins techniques :

- API REST structurée et sécurisée.
- Base de données relationnelle fiable.
- Application front-end performante, lisible et responsive.
- Architecture claire : routes → contrôleurs → modèles → BDD.

Public visé :

- Entreprises de recyclage,
- Collectivités,
- Associations écologiques,
- Utilisateurs internes.

Contraintes du projet

- Respect du cadre du TP DWWM
- Architecture Front-end / Back-end séparée
- Base de données relationnelle
- Sécurisation des accès (JWT, rôles)
- Application responsive

Livrables attendus

- Application web fonctionnelle
- API REST sécurisée
- Base de données MySQL
- Dossier projet
- Dépôt Git
- Jeu de données d’essai

3. Conception de l’application

3.1. Architecture générale

L’application respecte une structure **client/serveur** :

Front-End (React)

- Composants modulaires
- Router moderne
- Appels API via Axios

- Responsive Design
- Gestion du token JWT dans localStorage

Back-End (Node + Express)

- Architecture MVC
- Routes organisées par ressource
- Middleware d'authentification JWT
- Middleware d'autorisation (admin)
- Connexion MySQL via mysql2
- Sécurité : bcrypt, variables d'environnement

Base de données (MySQL)

- Table `users`
- Table `wastes`
- Contraintes FK
- Requêtes préparées pour éviter injections SQL

3.2. Maquettage et UX

Avant développement, l'interface a été pensée pour être :

- claire,
- accessible,
- épurée,
- efficace.

Principes suivis :

- Navigation simple et cohérente
- Navbar responsive (menu burger mobile)
- Formulaires centrés
- Couleurs liées à l’écologie (gamme de verts)
- Mise en avant des actions principales

Formats de maquettage

Les maquettes ont été conçues pour les formats :

- Mobile
- Tablette
- Desktop

Parcours utilisateurs

- Connexion
- Tableau de bord
- Liste des déchets
- Création / modification d’un déchet
- Gestion des utilisateurs (admin)

3.3. Modélisation de la base de données

Table USERS

- id (PK)
- name
- email (unique)

- password (hashé)
- role (user/admin)

Table WASTES

- id (PK)
- name
- category
- description
- user_id (FK → users.id)

Chaque entité correspond à un **Modèle (Model)** dans l'architecture du back-end.

Relations entre les entités (MCD)

- Un utilisateur peut créer plusieurs déchets (1,N)
- Un déchet est créé par un seul utilisateur (N,1)
- Un déchet appartient à une seule catégorie
- Une catégorie regroupe plusieurs déchets

4. Développement Front-End (React)

4.1. Stack technique

Le front de GreenBin repose sur :

- **React 19**
- **React Router DOM 7.9**
- **Axios** pour la communication avec l'API

- **Vite** pour le bundling rapide
- **ESLint** pour la qualité de code

Avantages :

- performances élevées,
- modularité des composants,
- mise à jour instantanée avec HMR,
- transitions fluides entre pages.

4.2. Réalisation des interfaces statiques

Les pages créées :

- Page de connexion
- Dashboard
- Liste des déchets
- Formulaire de création / modification
- Liste des utilisateurs (admin)
- Formulaire utilisateur
- Navbar responsive

Chaque interface a été pensée pour être simple, accessible et rapide à comprendre.

L'application comporte plusieurs vues distinctes : connexion, tableau de bord, liste des déchets, formulaire de création/modification et gestion des utilisateurs.

4.3. Développement dynamique

Toutes les pages sont connectées au back-end via Axios :

Fonctionnalités dynamiques :

- Récupération de données en temps réel
- Envoi d’un token JWT dans les requêtes
- Gestion des erreurs HTTP
- Redirections automatiques après actions (navigate)

La navbar s’adapte automatiquement au rôle de l’utilisateur.
(ex. : le lien "Utilisateurs" n’apparaît que si le rôle = admin)

4.4. Responsive Design

L’application a été optimisée pour plusieurs résolutions :

- mobile < 768px
- tablette
- desktop

Techniques utilisées :

- Flexbox
- Media queries
- Scroll horizontal sur les tableaux
- Menu burger sur petits écrans

Cela démontre la compétence **“Réaliser des interfaces web statiques et dynamiques”**.

---

5. Développement Back-End (Node.js / Express)

5.1. Architecture MVC

Le back-end est organisé selon 4 couches :

1. **Routes** → gèrent les endpoints accessibles par l’API
2. **Controllers** → logique métier
3. **Models** → interactions SQL MySQL
4. **Middleware** → sécurité, permissions, validation

Cette structure améliore :

- la lisibilité
- la maintenabilité
- la séparation des responsabilités

5.2. Authentification & Sécurité

Login sécurisé (email + password)

- Vérification email
- Comparaison du mot de passe hashé via **bcrypt**
- Génération d’un token **JWT** valable 24h

Middleware d’authentification (`authMiddleware.js`)

- Vérifie la présence du token

- Décode le token
- Injecte `req.user` dans la requête

Middleware Admin (`adminMiddleware.js`)

Vérifie :

```
if (req.user.role !== "admin")
```

Ce qui sécurise totalement les routes sensibles :

- gestion des utilisateurs
- certaines opérations critiques

Validation des données

- Les données issues des formulaires sont contrôlées avant traitement afin d'éviter les champs manquants, incohérents ou malveillants.

5.3. Routage et logique métier

Routes Auth :

- POST /login
- POST /register
- GET /me

Routes Users (admin uniquement) :

- GET /users
- GET /users/id/:id
- POST /users
- PUT /users/id/:id

- DELETE /users/id/:id

Routes Wastes :

- GET /wastes
- GET /wastes/:id
- POST /wastes
- PUT /wastes/:id
- DELETE /wastes/:id

Chaque route appelle un controller dédié.

5.4. Accès aux données SQL (Models)

Les models utilisent **mysql2** avec des **promises** :

Exemples :

- `User.getAll()`
- `User.update()`
- `Waste.create()`

Avantages :

- code propre
- gestion élégante des erreurs
- protection contre injections SQL

5.5. Gestion des rôles & Permissions

Le système repose sur :

- rôle stocké en base (`user` ou `admin`)
- rôle intégré au JWT
- vérification automatique par les middlewares

Ce système garantit une séparation claire entre :

- les utilisateurs simples
- les administrateurs

---

6. Tests, validations et qualité

Tests réalisés :

- Connexion / déconnexion
- Essais avec mauvais mot de passe
- Accès interdit pour un non-admin
- Création de déchets
- Modification et suppression
- Responsive sur mobile
- Gestion des erreurs API
- Vérification des permissions

Jeu de données d'essai

- 1 administrateur
- 2 utilisateurs
- Plusieurs déchets de test
  Ce jeu permet de valider les droits, la sécurité et les fonctionnalités CRUD.

- **Résultat :**

L’application est stable et conforme aux besoins fonctionnels.

7. Conclusion & Bilan personnel

Ce projet GreenBin m’a permis de démontrer l’ensemble des compétences nécessaires au TP DWWM :

- Développement front-end moderne (React)
- Réalisation d’interfaces responsives
- Développement back-end sécurisé (JWT, bcrypt)
- Implémentation d’une API REST complète
- Mise en place d’une base de données MySQL
- Utilisation d’une architecture MVC professionnelle

Ce travail m’a permis :

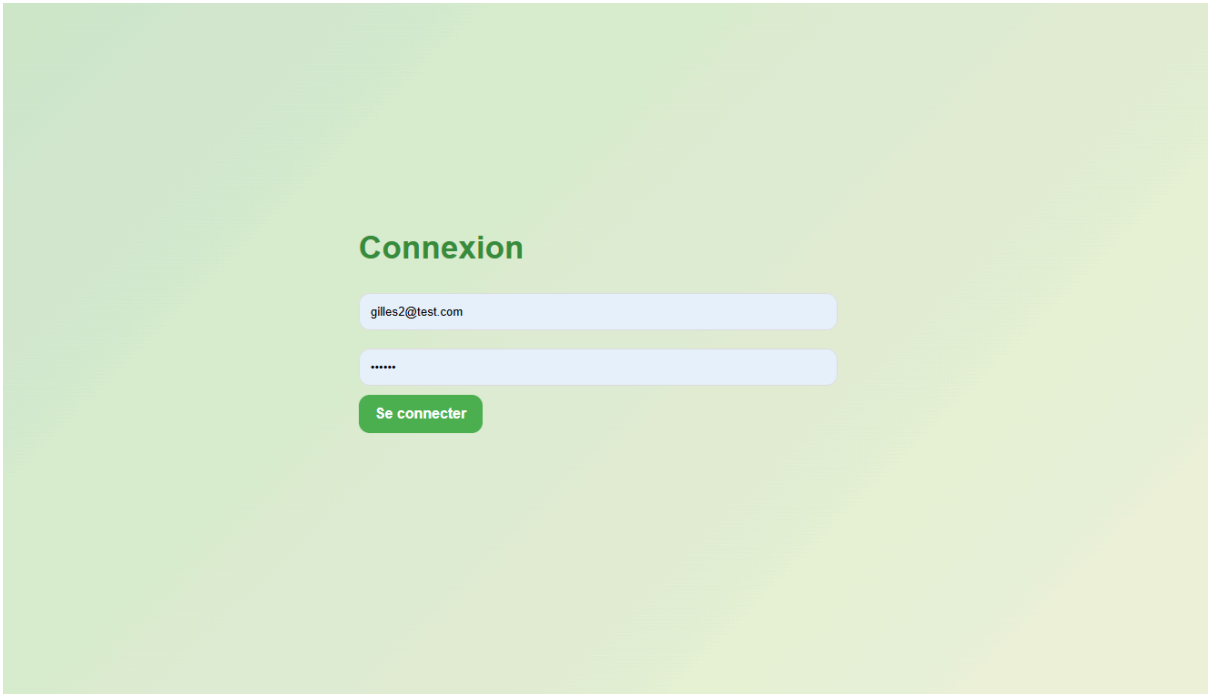
- de renforcer mes compétences techniques,
- de comprendre l’importance de la sécurité,
- d’apprendre à structurer un projet complet,
- de préparer efficacement la soutenance finale.
  GreenBin représente pour moi l’aboutissement du parcours DWWM et une base solide pour évoluer dans le monde professionnel.

Amélioration futur :

Le projet a été conçu pour être évolutif, notamment avec l’ajout futur d’une gestion plus fine des catégories de déchets.

ANNEXE 8 — Captures d’écran et éléments techniques

8.1. Page de connexion



- Champ email / mot de passe
- Bouton “Se connecter”
- Design épuré

8.2. Dashboard admin et user

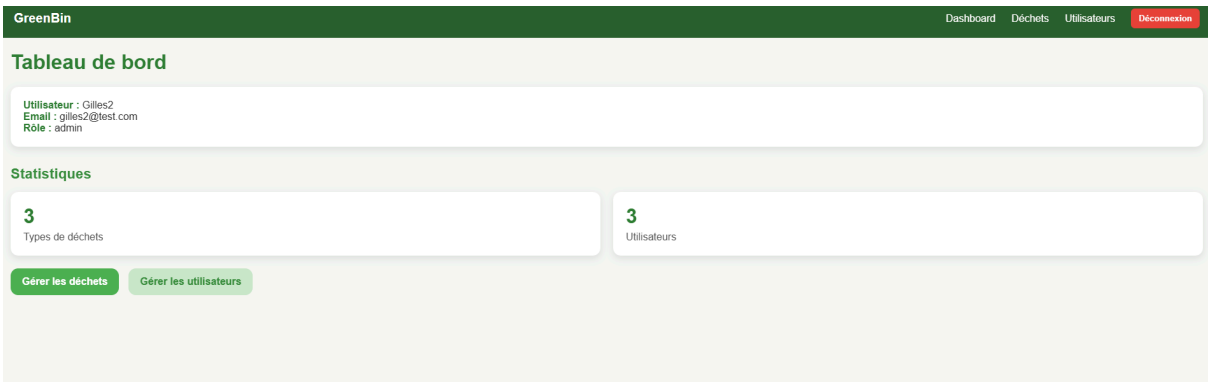


tableau de bord user

Structure générale

Navigation accessible

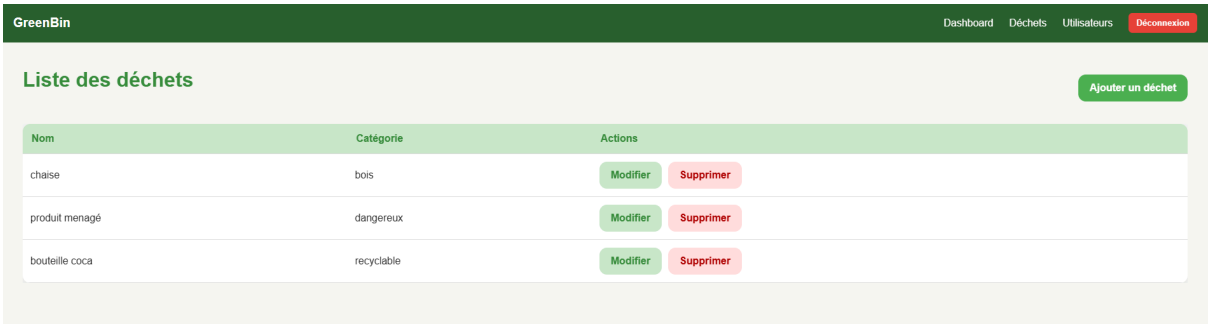
Affichage responsive

8.3. Liste des déchets (WastesList)

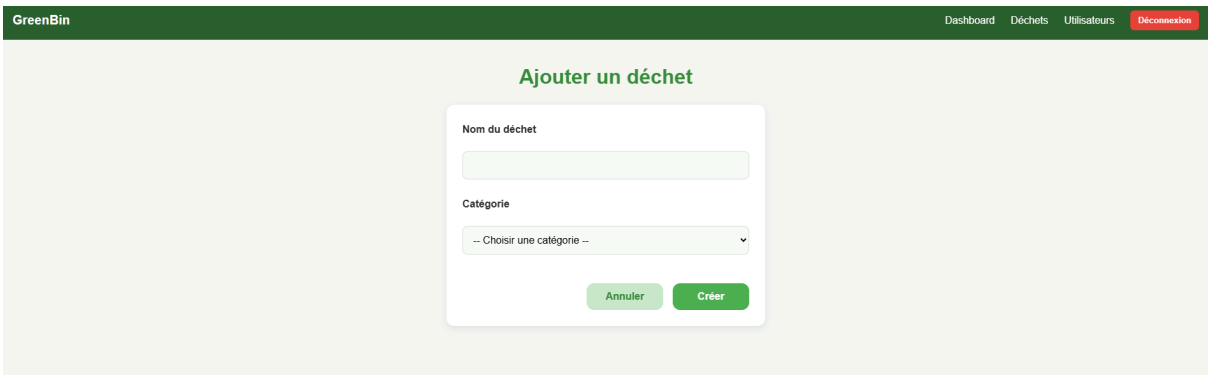
Tableau des déchets

Boutons : Modifier / Supprimer

Bouton “Ajouter un déchet”



8.4. Formulaire d’ajout / modification d’un déchet (WasteForm)

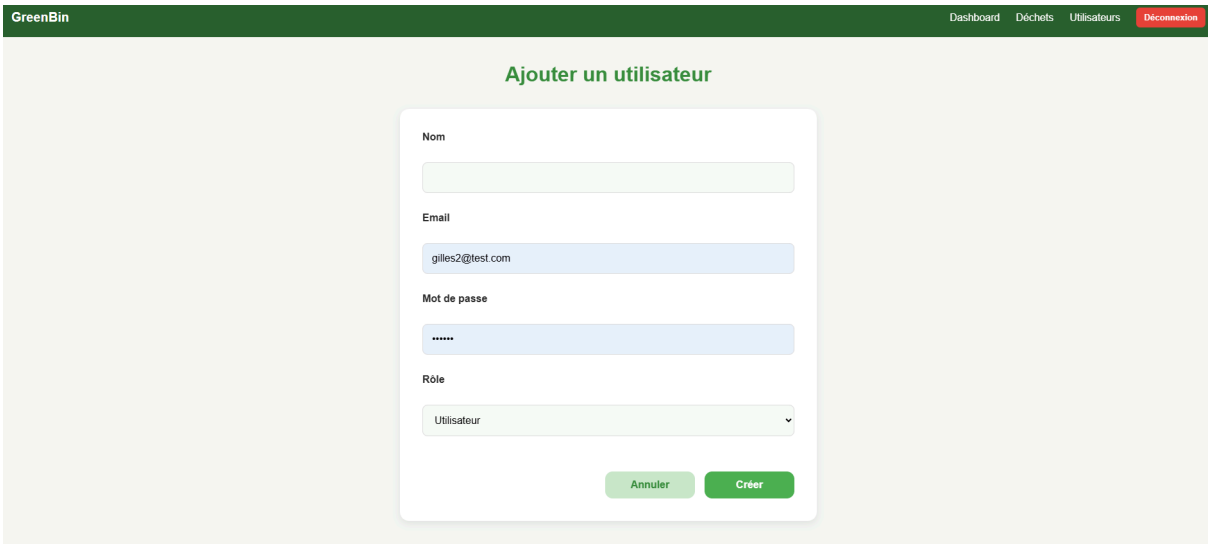


le formulaire de modification et le même visuellement mais il n’a pas les mêmes fonction.

Champs : nom, catégorie, description

Boutons : Annuler / Créer / Enregistrer

8.6. Formulaire d’ajout d’utilisateur



Nom

Email

Mot de passe (si création)

Sélecteur de rôle (user/admin)

8.7. Navigation & responsive design

8.8. Base de données MySQL

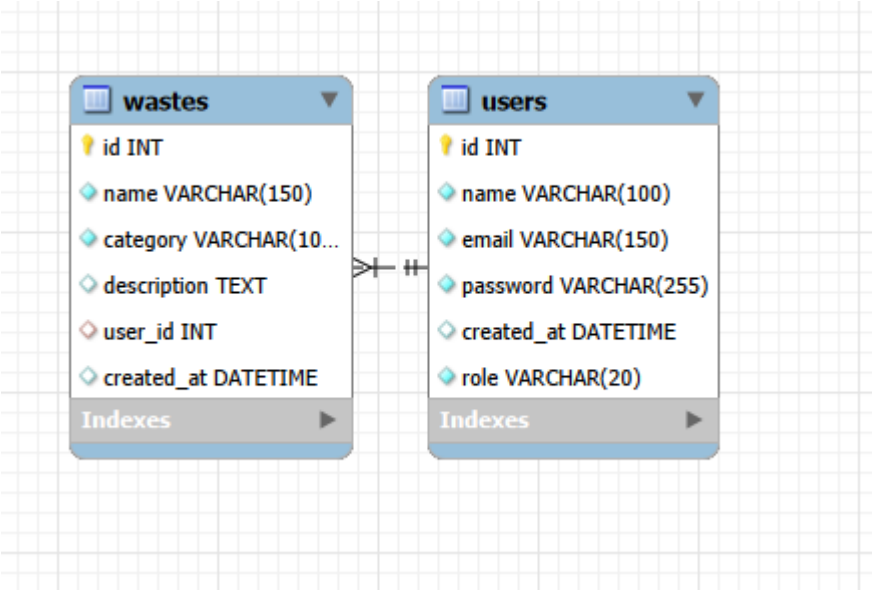


Table: users

Columns:

id	int AI PK
name	varchar(100)
email	varchar(150)
password	varchar(255)
created_at	datetime
role	varchar(20)

Table: wastes

Columns:

id	int AI PK
name	varchar(150)
category	varchar(100)
description	text
user_id	int
created_at	datetime

Tables principales :

📌 `users`

- **id**
- **name**
- **email**
- **password (hashé bcrypt)**
- **role (user/admin)**

📌 `wastes`

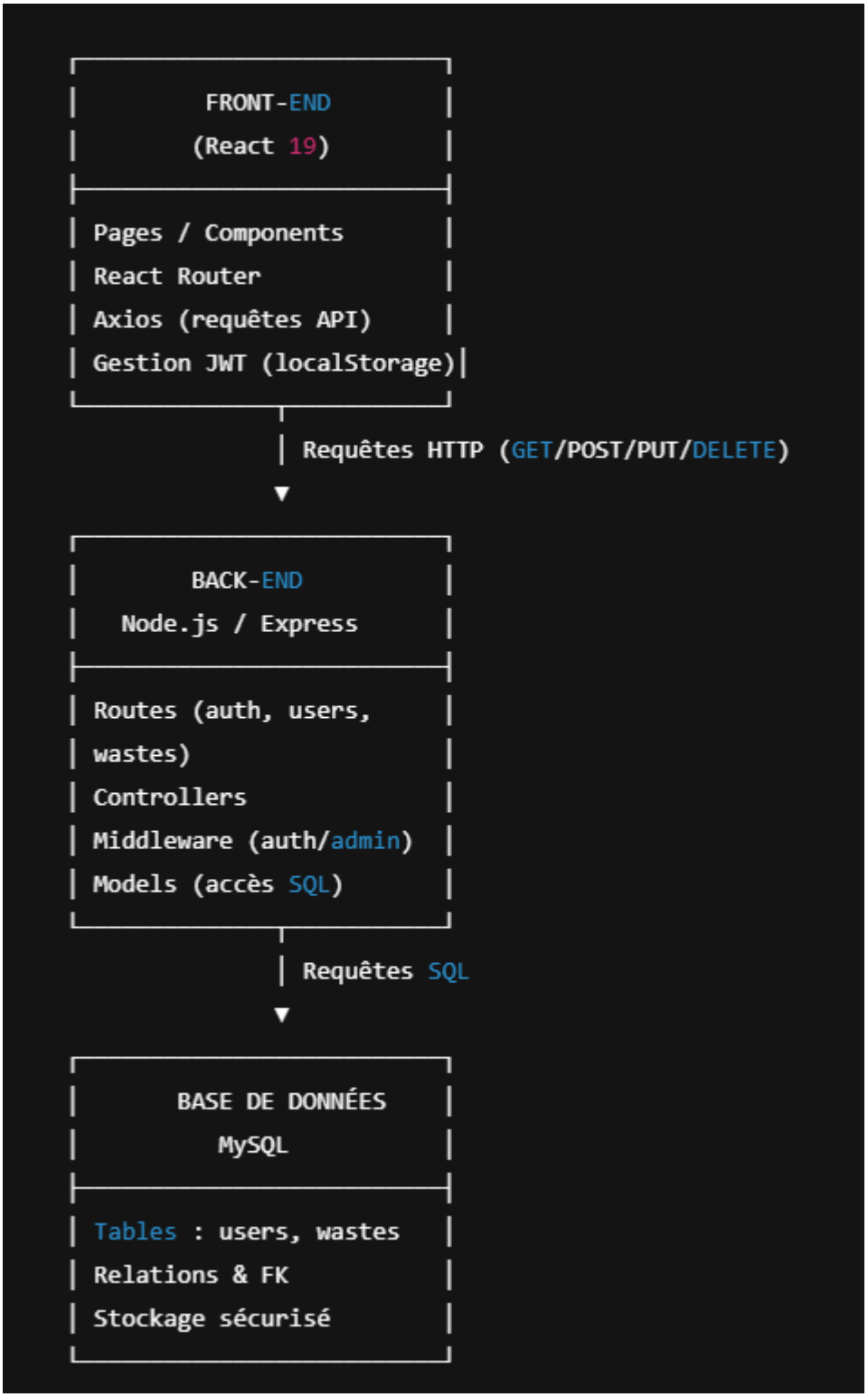
- **id**
- **name**

- **category**
- **description**
- **user_id (FK → users.id)**

🔐 **Sécurité :**

- **mots de passe hashés**
- **relations protégées**
- **accès filtré par middleware**

8.9. Architecture du projet


FRONT-END
(React 19)
Pages / Components
React Router
Axios (requêtes API)
Gestion JWT (localStorage)
Requêtes HTTP (GET/POST/PUT/DELETE)
BACK-END
Node.js / Express
Routes (auth, users, wastes)
Controllers
Middleware (auth/admin)
Models (accès SQL)
Requêtes SQL
BASE DE DONNÉES
MySQL
Tables : users, wastes
Relations & FK
Stockage sécurisé

8.10. Extraits de code significatifs

🟩 1. Middleware d’authentification (JWT)

Ce middleware protège les routes en vérifiant la présence et la validité d’un token JWT.
Il permet d’identifier l’utilisateur faisant la requête.

```
const jwt = require("jsonwebtoken");

module.exports = (req, res, next) => {
  const authHeader = req.headers.authorization;

  if (!authHeader || !authHeader.startsWith("Bearer ")) {
    return res.status(401).json({ message: "Token manquant ou mal formaté" });
  }

  const token = authHeader.split(" ")[1];

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);

    req.user = decoded;  // Ajoute l'utilisateur décodé à req.user

    next();
  } catch (error) {
    return res.status(403).json({ message: "Token invalide" });
  }
};
```

Vérifie l’en-tête `Authorization`

Extrait le token

Déchiffre le token JWT avec la clé secrète

Ajoute `req.user` pour rendre l’utilisateur disponible dans les controllers

En cas d’erreur → renvoie 401 ou 403

🟩 2. Route d’ajout d’un déchet

Cette route permet à un utilisateur authentifié d’ajouter un déchet dans la base de données.

Code : `wasteRoutes.js`

```
router.post("/", auth, wasteController.create);
```

Explication

- L’utilisateur doit être connecté → middleware `auth`
- La route appelle `wasteController.create`
- Le déchet sera associé à `req.user.id` (l’utilisateur créateur)

🟩 3. Fonction Login (Controller)

Elle gère la connexion d’un utilisateur existant, vérifie le mot de passe, puis génère un **token JWT** valable 24h.

🔧 Code : `authController.js` (login)

```
login: (req, res) => {

  const { email, password } = req.body;


  if (!email || !password) {

    return res.status(400).json({ message: "Email et mot de passe
requis" });

  }
```

```
  const query = "SELECT * FROM users WHERE email = ?";

  db.query(query, [email], async (err, result) => {

    if (err) return res.status(500).json({ message: "Erreur
serveur", err });



    if (result.length === 0) {

      return res.status(400).json({ message: "Email incorrect" });

    }



    const user = result[0];



    const isMatch = await bcrypt.compare(password, user.password);



    if (!isMatch) {

      return res.status(400).json({ message: "Mot de passe
incorrect" });

    }



    const token = jwt.sign(

      { id: user.id, email: user.email, role: user.role },

      process.env.JWT_SECRET,

      { expiresIn: "24h" }

    );



    res.json({
```

```
      message: "Connexion réussie",
      token,
      user: {
        id: user.id,
        name: user.name,
        email: user.email,
        role: user.role
      }
    });
  });
}
```

Explication claire

1. Vérifie que l'email et le mot de passe sont fournis.
2. Vérifie que l'utilisateur existe dans la table SQL.
3. Compare le mot de passe avec le hash bcrypt stocké.
4. Si tout est bon → génère un **JWT** signé.
5. Renvoie le profil minimal + token.

🟩 4. Exemple de modèle MySQL (Modèle Waste)

Ce modèle gère les requêtes SQL pour les déchets.

Code : `wasteModel.js`

```
const db = require("../config/database");


module.exports = {
  getAll: () => {
    return new Promise((resolve, reject) => {
      db.query("SELECT * FROM wastes", (err, result) => {
        if (err) reject(err);
        resolve(result);
      });
    });
  },


  getById: (id) => {
    return new Promise((resolve, reject) => {
      db.query("SELECT * FROM wastes WHERE id = ?", [id], (err,
result) => {
        if (err) reject(err);
        resolve(result[0]);
      });
```

```
    });
  },


  create: (data) => {
    return new Promise((resolve, reject) => {
      const query =
        "INSERT INTO wastes (name, category, description, user_id)
VALUES (?, ?, ?, ?)";


      db.query(
        query,
        [data.name, data.category, data.description, data.user_id],
        (err, result) => {
          if (err) reject(err);
          resolve(result);
        }
      );
    });
  }
};
```

Explication

- Le modèle encapsule toutes les requêtes SQL.
- Retourne des Promises pour être utilisé avec `async/await`.

- L’insertion ajoute :
    - le nom
    - la catégorie
    - la description
    - l’ID du créateur (`user_id`)