

# PROJET FINAL – DELIVECROUS API BACKEND APPLICATION DE LIVRAISON DE REPAS DU CROUS

koman Boni

Développeur Authentification & Données | Mise en place du système \*\*JWT\*\*, création du modèle `Plat`, intégration de la base SQLite avec Prisma, creation de front

Anas chebbi

finalisation du modèle `Commande`, creation du Rapport PDF, Développeur Tests & Intégration | Création et automatisation des \*\*tests\*\* Jest/Supertest\*\*, ajout du fichier ` `.env.test` ,

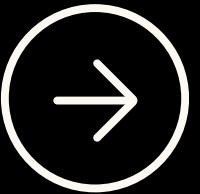
ilias Hanfaoui

Développeur Back-End | Initialisation du serveur Express, création des routes `users`, et gestion de la logique serveur principale



# OBJECTIF GÉNÉRAL :

## DÉVELOPPER UNE API BACKEND QUI GÈRE LA LIVRAISON DE REPAS POUR LES ÉTUDIANTS DU CROUS



### Objectifs du projet :

- 1 Authentification des utilisateurs
- 2 Gestion des plats (création, modification, suppression)
- 3 Gestion des commandes (suivi, statut)
- 4 Sécurisation avec JWT
- 5 Rôles utilisateur
- 6 Vérification des routes via Postman avec captures d'écran.

Outils de test : Vs code / Postman



# Choix Techniques :

## ■ Choix du SGBD : SQL (PostgreSQL)

Nous avons choisi SQL (PostgreSQL) pour ce projet car :

- Les données sont fortement reliées (utilisateurs, plats, commandes).
- SQL assure une intégrité et une cohérence des relations entre tables.
- Prisma facilite la création, la migration et la gestion d'une base SQL.

NoSQL aurait offert plus de flexibilité, mais la structure relationnelle du projet s'adapte mieux à une base SQL.

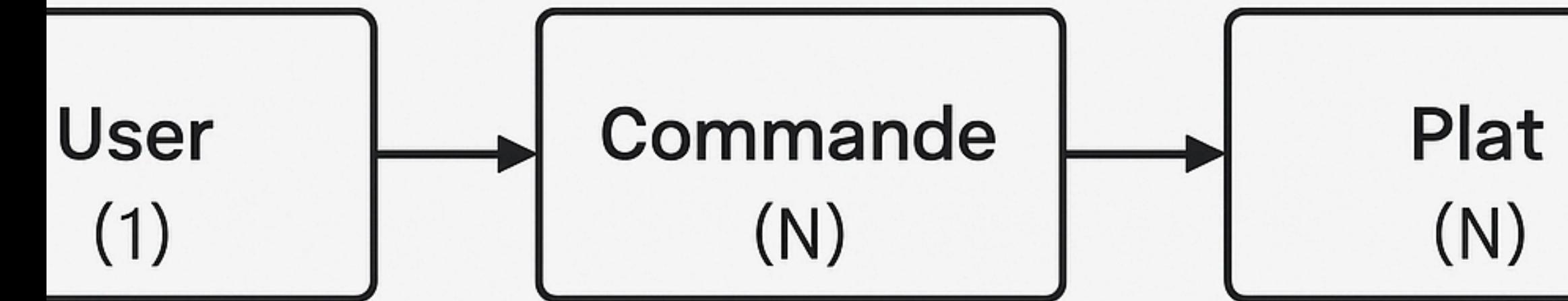
# STACK TECHNIQUE

- Node.js : environnement d'exécution JavaScript
- Express.js : framework de création d'API
- Prisma ORM : interface entre Node.js et la base SQL
- PostgreSQL : base de données choisie
- Thunder Client : test des routes HTTP
- Html,css,js pour le front





# Schéma BDD



# Principales Relations

Relations principales (vue conceptuelle)

1) User → Commande

- Un utilisateur passe des commandes.

Cardinalité : User (1) – passe – (N) Commande

Lecture inverse : une commande appartient à un seul user.

2) Commande ↔ Plat (via CommandePlat)

- Une commande contient plusieurs plats ; un plat peut apparaître dans plusieurs commandes.

Cardinalité : Commande (1) – contient – (N) CommandePlat – référence – (1) Plat

Interprétation : relation N-N matérialisée par l'entité d'association CommandePlat (avec quantite, prixUnitaire).

3) User ↔ Plat (Favori) – optionnel / bonus

- Un utilisateur peut marquer des plats en favoris.

Cardinalité : User (1) – favorise – (N) Favori – concerne – (1) Plat

Contrainte : (userId, platId) unique.

# LISTE DES ROUTES PRINCIPALES

Méthode	Route	Description
GET	/users	Liste tous les utilisateurs
post	/users	Crée un nouvel utilisateur
Put	/users/:id	Met à jour un utilisateur
Delete	/users/:id	Supprime un utilisateur



# captures Thunder Client/Postman (register, login, CRUD)

Thunder Client interface showing captures for Postman requests:

- POST http://localhost:3000/login**: Headers (9), Body (raw JSON):

```
1 {  
2   "email": "Koman@gmail.fr",  
3   "password": "Homme"  
4 }
```

- POST http://localhost:3000/register**: Headers (9), Body (raw JSON):

```
1 {  
2   "name": "Koman",  
3   "email": "Koman@gmail.fr",  
4   "password": "Homme",  
5   "role": "student"  
6 }
```

- Body (JSON) Preview** for the login response (Message: Connexion réussie, Token generated):

```
1 {  
2   "message": "Connexion réussie",  
3   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
4       eyJ1c2VySWQiOjMsImVtYWlsIjoiS29tYW5AZ21haWwuZnIiLCJyb2x1  
5       mcq3bGD-In4XWfTZAQ1nTJF7wSsMEhPIJR0lVdzVlIA",  
6   "user": {  
7     "id": 3,  
8     "email": "Koman@gmail.fr",  
9     "name": "Koman",  
10    "role": "student"  
11  }  
12 }
```

- Body (JSON) Preview** for the register response (Message: Compte créé avec succès, Token generated):

```
1 {  
2   "message": "Compte créé avec succès",  
3   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
4       eyJ1c2VySWQiOjMsImVtYWlsIjoiS29tYW5AZ21haWwuZnIiLCJyb2x1  
5       -4QGkDA-JtaM2BVDBjgUqgPm-FCGajlImG8HImQ3Lyk",  
6   "user": {  
7     "id": 3,  
8     "email": "Koman@gmail.fr",  
9     "name": "Koman",  
10    "role": "student"  
11  }  
12 }
```

# • PRINCIPALES DIFFICULTÉS

## GÉNÉRATION DES ROUTES

- Chemins et verbes parfois incohérents au début (singulier/pluriel, GET /commande vs GET /commandes/:id).
- Doublons/chevauchements de handlers lors des merges.

## CONFLITS GIT

Conflits sur main.js après ajouts simultanés de routes.  
Adoption d'un mini-workflow : pull avant push, branches "feature/\*", PR courte et revue rapide.

## VARIATIONS D'ENVIRONNEMENT ENTRE POSTES

différences d'environnement entre postes (versions de Node, variables d'environnement mal définies) provoquant des erreurs locales non reproductibles immédiatement sur le dépôt CI.

Solution : standardisation via .env.

## FORMAT ET VALIDATION DES IDENTIFIANTS

incohérences temporaires dans le format des id entre plusieurs endpoints (string UUID vs numeric id) provoquant des 404/400 pendant les tests.

# CE QUE NOUS AVONS APPRIS

## Créer un serveur Node.js minimal

Nous avons appris à initialiser un projet Node.js et à créer un serveur Express capable de gérer plusieurs routes. Cette étape nous a permis de comprendre la structure d'une API REST et l'organisation du code backend.

## Configurer Prisma et une base SQL

La mise en place de Prisma avec une base PostgreSQL nous a fait découvrir la modélisation des données et la gestion des relations entre entités.

## Travailler en équipe sur GitHub

Le travail collaboratif sur GitHub nous a appris à gérer les commits, résoudre des conflits de merge et maintenir un code propre et cohérent. Cette expérience nous a permis de mettre en pratique les bonnes pratiques de versionnage en équipe.

## Utiliser un outil de test d'API

Grâce à Thunder Client et Postman, nous avons pu tester et valider nos routes rapidement. Ces outils nous ont aidés à vérifier les réponses du serveur, à détecter les erreurs, et à améliorer la fiabilité de notre API.

