Étape 1 — `GET /menu` lit Neon (remplace `menu.json` à l'exécution)

Ce qu'on fait

Ton back ne lit plus `menu.json` pour répondre à `/menu` : il va chercher les plats dans la BDD **Neon** (table `plates`). `menu.json` peut rester comme *seed* (pour remplir la BDD plus tard), mais **la source de vérité** devient Neon.

Outils

- Côté **back** : Node.js, Express, `pg` (pilote Postgres), `dotenv` (pour `.env`).
- Côté **DB** : Neon (ta base existe déjà, tables vides).

Où taper

- **Terminal WSL/Ubuntu** (ou terminal intégré VS Code), **dans le dossier `back/`**.

Commandes (commentées)

```
npm i pg dotenv
# installe le pilote Postgres (pg) et dotenv (charge .env)
```

Crée un fichier **`back/.env`** :

```
# Ta chaîne de connexion Neon (bouton "Copy" dans Neon).
# Ajoute ?sslmode=require si nécessaire.
DATABASE_URL=postgresql://USER:PASSWORD@HOST.neon.tech/DBNAME?sslmode=require
```

Ajoute dans **`back/.gitignore`** (si ce n'est pas déjà fait) :

```
node_modules
.env
```

> `.env` ne doit pas partir sur Git. `node_modules` non plus.

Code (chaque ligne expliquée)

Dans ton fichier serveur (celui où tu as `import express ...`), **tout en haut** ajoute :

```
import 'dotenv/config';                          // charge .env → process.env remplit (dont
import pkg from 'pg';                            // importe le paquet 'pg' (pilote Postgres)
const { Pool } = pkg;                            // extrait la classe Pool (gère des connexi

const pool = new Pool({                          // crée un "pool" de connexions vers Neon
  connectionString: process.env.DATABASE_URL  // lit l'URL de la BDD dans .env
  // pas besoin d'option ssl ici si ?sslmode=require est dans l'URL
});
```

Remplace ta route `/menu` par :

```
app.get("/menu", async (req, res) => {        // déclare la route GET /menu (fonction asy
  try {                                       // protège l'intérieur contre les erreurs
    const sql = `SELECT id, plate, description, image, price
                 FROM plates
                 ORDER BY id;`;               // requête SQL : prend les colonnes utiles,
    const { rows } = await pool.query(sql);   // exécute le SQL ; rows = tableau JS de li
    return res.json(rows);                    // renvoie le tableau au front en JSON
  } catch (err) {                             // en cas d'erreur (connexion, SQL, etc.)
    console.error("GET /menu error:", err);   // log côté serveur (utile pour toi)
    return res.status(500).json({ error: "DB error" }); // réponse d'erreur côté client
  }
});
```

Lancer le serveur

```
node script.js
# (ou le nom de ton fichier serveur : index.js, app.js…)
# vois "Serveur lancé…" dans le terminal
```

Vérifier

- Mets 1 plat dans Neon (dans l'éditeur SQL) puis recharge `http://localhost:3000/menu`.
- Ton front affiche alors ce que renvoie la BDD (plus le JSON local).

Étape 2 — `POST /clients` : le front envoie un nom, le back renvoie un client_id

> But : identifier le client côté back. On te donne deux routes possibles :
>
> - **Option A (simple, immédiate)** : stock **en mémoire** (pas encore en BDD).
> - **Option B (si tu as déjà une table `clients`)** : stock **en BDD**.

Front (où ? page menu)

Au chargement de la page menu, tu lis le prénom depuis `localStorage` et tu l'envoies au back.

```
const name = (localStorage.getItem("firstName") || "").trim();     // lit le prénom stock
if (!name) { window.location.href = "/index.html"; }               // si pas de prénom,

const res = await fetch("http://localhost:3000/clients", {         // appelle l'API /clie
  method: "POST",                                                  // méthode HTTP = POS
  headers: { "Content-Type": "application/json" },                 // dit au serveur : l
  body: JSON.stringify({ client_name: name })                      // transforme l'objet
});

if (!res.ok) {                                                     // si le serveur renv
  const err = await res.json().catch(() => ({}));                  // essaye de lire le
  console.error("POST /clients", err);                             // log dans la consol
}

const client = await res.json();                                   // lit la réponse JSO
sessionStorage.setItem("client_id", String(client.id));            // mémorise l'id clie
```

Pourquoi le header `Content-Type: application/json` ? Il **déclare le format du corps**. Grâce à ça, côté back, `app.use(express.json())` peut **parser** et remplir `req.body`. Sans ce header, `req.body` serait vide.

Back — Option A (mémoire, rapide)

Ajoute **avant** `app.listen(...)` :

```
const clients = [];                    // "table" en mémoire (tableau d'objets)
let nextClientId = 1;                  // compteur pour simuler des IDs

app.post("/clients", (req, res) => {   // route POST /clients
  const { client_name } = req.body;    // récupère le champ envoyé par le front (grâce à
  if (!client_name || !client_name.trim()) {
    return res.status(400).json({ error: "client_name requis" }); // valide
  }

  const name = client_name.trim();
  let existing = clients.find(c => c.client_name === name); // évite les doublons pendant
  if (existing) return res.json(existing);                  // si déjà existant → renvoi

  const client = { id: nextClientId++, client_name: name };  // crée un client avec id si
  clients.push(client);                                      // stocke en mémoire
  return res.status(201).json(client);                       // renvoie { id, client_name
});
```

Back — Option B (BDD Neon, si `clients` existe déjà)

```
app.post("/clients", async (req, res) => {                    // route POST /clients asyn
  const { client_name } = req.body;                           // lit le champ envoyé
  if (!client_name || !client_name.trim()) {
    return res.status(400).json({ error: "client_name requis" });
  }
  const name = client_name.trim();

  try {
    // essaie de retrouver un client existant
    const sel = await pool.query(
      "SELECT id, client_name FROM clients WHERE client_name = $1",
      [name]
    );                                                        // $1 = paramètre sécurisé (
    if (sel.rows.length) return res.json(sel.rows[0]);        // si trouvé → renvoie

    // sinon crée le client
    const ins = await pool.query(
      "INSERT INTO clients (client_name) VALUES ($1) RETURNING id, client_name",
      [name]
    );                                                        // INSERT + RETURNING pour r
    return res.status(201).json(ins.rows[0]);                 // renvoie { id, client_name
  } catch (err) {
    console.error("POST /clients error:", err);               // log serveur
    return res.status(500).json({ error: "DB error" });       // message client
  }
});
```

Étape 3 — `POST /orders` : créer une commande (statut initial = `en_attente`)

> But : au clic sur “Commander”, on envoie `plate_id` + `client_id` au back, qui crée une ligne dans la table `orders` (ou `commande` selon ton nommage) avec `status = 'en_attente'`. **Affichage détaillé de la commande** pourra arriver juste après (sur `commande.html`).

Front (où ? page menu, au clic sur un bouton)

```
// Quand tu crées le bouton, stocke l’id du plat dans un data-attribute :
btn.className = "order-button";                    // classe CSS
btn.dataset.plateId = String(plate.id);            // mémorise l’id du plat sur le bout

// Plus tard, pour chaque bouton :
btn.addEventListener("click", async (e) => {       // écoute le clic sur CE bouton
  e.preventDefault();                              // empêche un submit/navigation par

  const plate_id = Number(btn.dataset.plateId);    // récupère l’id du plat depuis le b
  const client_id = Number(sessionStorage.getItem("client_id")); // lit l'id client mémor

  const res = await fetch("http://localhost:3000/orders", {      // appelle l’API /orders
    method: "POST",                                              // on crée une commande
    headers: { "Content-Type": "application/json" },             // corps = JSON
    body: JSON.stringify({ client_id, plate_id })                // envoie les ids
  });

  const order = await res.json();                                // lit la commande créée
  sessionStorage.setItem("last_order_id", String(order.id));     // mémorise l'id de la c
  window.location.href = "/pages/commande.html";                 // va sur la page comman
});
```

Back — Option A (mémoire, rapide)

```
const orders = [];                          // "table" commandes en mémoire
let nextOrderId = 1;                        // compteur d'id pour commandes

app.post("/orders", (req, res) => {         // route POST /orders
  const { client_id, plate_id } = req.body;// récupère les ids envoyés
  if (!Number.isInteger(client_id)) {
    return res.status(400).json({ error: "client_id entier requis" });
  }
  if (!Number.isInteger(plate_id)) {
    return res.status(400).json({ error: "plate_id entier requis" });
  }

  // (facultatif) vérifie que le plat existe dans /menu (items DB) selon ton choix
  // (facultatif) vérifie que le client existe (si Option A de l'Étape 2)

  const order = {                           // construit la commande
    id: nextOrderId++,                      // id simulé
    client_id,                              // id client
    plate_id,                               // id plat
    status: "en_attente"                    // statut initial
  };
  orders.push(order);                       // stocke en mémoire
  return res.status(201).json(order);       // renvoie la commande
});
```

Back — Option B (BDD Neon, si `orders` existe déjà)

> Adapte **les noms de colonnes** à ton schéma (ex: `commande`, `status_id`, etc.). Exemple générique avec une colonne texte `status` :

```
app.post("/orders", async (req, res) => {                // route POST /orders asynchron
  const { client_id, plate_id } = req.body;              // récupère les ids envoyés
  if (!Number.isInteger(client_id)) {
    return res.status(400).json({ error: "client_id entier requis" });
  }
  if (!Number.isInteger(plate_id)) {
    return res.status(400).json({ error: "plate_id entier requis" });
  }

  try {
    const sql = `
      INSERT INTO orders (client_id, plate_id, status)
      VALUES ($1, $2, 'en_attente')
      RETURNING id, client_id, plate_id, status
    `;                                                   // INSERT + statut initial + ren
    const { rows } = await pool.query(sql, [client_id, plate_id]); // exécution paramétré
    return res.status(201).json(rows[0]);                // renvoie la commande créée
  } catch (err) {
    console.error("POST /orders error:", err);           // log serveur
    return res.status(500).json({ error: "DB error" });  // message client
  }
});
```

Règles d'or transversales (à garder en tête)

- **Toujours avant tes routes** :

```
app.use(cors({ origin: ["http://127.0.0.1:5500", "http://localhost:5500"] })); // aut
app.use(express.json());                                                        // par
```

- **Côté front** : pour envoyer du JSON → `headers: { "Content-Type": "application/json" }` + `body: JSON.stringify(...)`.
- **Côté back** : pour lire le JSON → `app.use(express.json())`, puis `req.body` dans la route.

Tu as maintenant le **chemin complet** :

1. **Étape 1** : `/menu` lit Neon → ton front affiche les plats de la BDD.
2. **Étape 2** : `/clients` reçoit le prénom → renvoie `client_id` (mémoire ou BDD).
3. **Étape 3** : `/orders` crée une commande (`en_attente`) avec `client_id` + `plate_id`.

Quand tu valides l'Étape 1, on enchaîne sur l'Étape 2 puis 3 tranquillement.