Menu → Panier (lclStrg) → Commande (API + DB)

Sommaire

1. **Charger le menu (`GET /api/menu`)**
   Qu'est-ce qu'on fait ? Le front réclame la liste des plats.
   Pourquoi ? Afficher ce qui vient de la base (source de vérité).

2. **Ajouter un plat au panier (`localStorage`)**
   Qu'est-ce qu'on fait ? Au clic "Commander", on stocke `{dishId, qty}` dans `localStorage.cart`.
   Pourquoi ? Laisser l'utilisateur composer sa commande, persistant entre pages.

3. **Continuer d'ajouter / revenir en arrière**
   Qu'est-ce qu'on fait ? On relit le panier et on ajoute/incrémente.
   Pourquoi ? Construire la commande de façon flexible côté client.

4. **Garantir un client (`POST /api/clients`)**
   Qu'est-ce qu'on fait ? Envoyer le `name` (depuis `localStorage.clientName`) pour obtenir `clientId`.
   Pourquoi ? Lier la commande à une personne en base.

5. **Valider la commande (`POST /api/orders`)**
   Qu'est-ce qu'on fait ? Envoyer `{ clientId, items }` (items dérivés du panier).
   Pourquoi ? Enregistrer en BD, renvoyer `orderId`, vider le panier, afficher un statut de confirmation.

Où ça s'exécute / qui parle à qui ?

- **Front (navigateur)** : affiche, gère le panier (`localStorage`), fait `fetch`.
- **Back (Express/Node)** : reçoit les requêtes `/api/...`, appelle la **DB**.
- **DB (Neon/PostgreSQL)** : stocke plats, clients, commandes.
- **Chaîne typique** : Front → Back (HTTP) → DB (SQL) → Back → Front (JSON).

Points d'attention

- `GET /api/menu` → **200 OK** + JSON.
- `POST /api/clients` → **201 Created** + `{id, name}` ; **400** si `name` manquant.
- `POST /api/orders` → **201 Created** + `{id}` ; **400** si `items` vide ou `clientId` manquant.
- Côté front : `if (!response.ok) throw ...` + message clair.
- Panier : toujours **lire** → **modifier** → **écrire** (JSON).

étapes détaillées

Étape 1 — Afficher le menu depuis la base

Qu'est-ce qu'on fait ? Le front demande la liste des plats au back, le back lit la DB, renvoie du JSON, le front affiche.
Pourquoi ? La source de vérité est la DB (Neon).
Où ? Back (Node/Express) sur **ta machine** ; DB chez **Neon** ; Front dans le **navigateur**.
Qui parle à qui ? Navigateur → (HTTP GET) → Express → (SQL) → Neon → Express → (JSON) → Navigateur.
Quand ? Au chargement de `menu.html`.

Back — connexion DB (Neon) — `back/models/db.js`

```
// S'exécute sur le SERVEUR (Node)
import { neon } from '@neondatabase/serverless';
export const sql = neon(process.env.DATABASE_URL); // .env: DATABASE_URL=...
```

Back — accès aux plats — `back/models/menu.model.js`

```
import { sql } from './db.js';

export async function listDishes() {
  return await sql`
    SELECT id, code, name, description, price_cents, image_url, category
    FROM dishes
    ORDER BY id ASC
  `;
}
```

Back — controller — `back/controllers/menu.controller.js`

```
import { listDishes } from '../models/menu.model.js';

export async function getMenu(req, res) {
  try {
    const rows = await listDishes();    // ⇔ DB
    res.status(200).json(rows);         // 200 OK + JSON
  } catch (e) {
    res.status(500).json({ message: 'Menu error' });
  }
}
```

Back — route — back/routes/menu.routes.js

```
import { Router } from 'express';
import { getMenu } from '../controllers/menu.controller.js';
const r = Router();
r.get('/', getMenu); // GET /api/menu
export default r;
```

Back — point d'entrée — back/server.js

```
import 'dotenv/config';
import express from 'express';
import cors from 'cors';

import menuRoutes from './routes/menu.routes.js';

const app = express();
app.use(cors({ origin: ['http://127.0.0.1:5500', 'http://localhost:5500'] }));
app.use(express.json());

app.use('/api/menu', menuRoutes);

app.listen(process.env.PORT || 3000, () =>
  console.log('API http://localhost:3000'));
```

Front — demander + afficher — front/js/menu-page.js

```
const API = 'http://localhost:3000/api'; // back Express

async function getMenu() {
  const res = await fetch(`${API}/menu`, {
    method: 'GET',
    headers: { 'Accept': 'application/json' }
  });
  if (!res.ok) {
    const info = await res.json().catch(() => ({}));
    throw new Error(info.message || `HTTP ${res.status}`);
  }
  return await res.json(); // tableau de plats
}

function renderMenu(dishes) {
  const list = document.querySelector('#menu-list');
  list.innerHTML = '';
  for (const d of dishes) {
    const card = document.createElement('div');
    card.className = 'dish';
    card.innerHTML = `
      <img src="${d.image_url || ''}" alt="${d.name}">
      <h3>${d.name}</h3>
      <p>${(d.price_cents/100).toFixed(2)} €</p>
      <button data-dishid="${d.id}" class="btn-order">Commander</button>
    `;
    list.appendChild(card);
  }
}

(async function bootstrapMenuPage() {
  try {
    const dishes = await getMenu();
    renderMenu(dishes);
  } catch (e) {
    alert('Impossible de charger le menu.');
  }
})();
```

Étape 2 — “Commander” = ajouter au panier (`localStorage`)

Qu’est-ce qu’on fait ? Au clic “Commander”, on enregistre le plat dans un panier (`localStorage.cart`).
Pourquoi ? Cumuler plusieurs plats, revenir en arrière, persister entre pages.
Où ? Navigateur uniquement.
Qui parle à qui ? Aucune requête réseau : lecture/écriture `localStorage`.
Quand ? À chaque clic “Commander”.

Front — utilitaires panier — `front/js/menu-page.js` (complément)

```
function readCart() {
  const raw = localStorage.getItem('cart');
  return raw ? JSON.parse(raw) : []; // ex: [{dishId: 12, qty: 2}, ...]
}

function writeCart(cart) {
  localStorage.setItem('cart', JSON.stringify(cart));
}

function addToCart(dishId, qty = 1) {
  const cart = readCart();
  const line = cart.find(l => l.dishId === dishId);
  if (line) { line.qty += qty; }
  else { cart.push({ dishId, qty }); }
  writeCart(cart);
}

// Délégation d'événements sur les boutons "Commander"
document.addEventListener('click', (e) => {
  const btn = e.target.closest('.btn-order');
  if (!btn) return;
  const dishId = Number(btn.dataset.dishid);
  addToCart(dishId, 1);
  btn.textContent = 'Ajouté ✓';
  setTimeout(() => (btn.textContent = 'Commander'), 800);
});
```

Astuce compteur (optionnel) :

```
function cartCount() {
  return readCart().reduce((acc, l) => acc + l.qty, 0);
}
function updateCartBadge() {
  const badge = document.querySelector('#cart-count');
  if (badge) badge.textContent = cartCount();
}
window.addEventListener('DOMContentLoaded', updateCartBadge);
```

Étape 3 — Garantir un client (`POST /api/clients`) puis valider la commande (`POST /api/orders`)

1. Obtenir un `clientId` en envoyant `{ name }` (depuis `localStorage.clientName`).
2. Envoyer `{ clientId, items }` (dérivés du panier) pour créer l'**order** + ses lignes.

Pourquoi ? Persister la commande en base, produire un `orderId`, vider le panier.
Où ? Front (navigateur) → Back (Express) → DB (Neon).
Qui parle à qui ? Front → `/api/clients` → DB → Front, puis Front → `/api/orders` → DB → Front.
Quand ? Au clic "Valider la commande".

Front — helpers POST JSON — `front/js/menu-page.js`

```
async function postJson(path, payload) {
  const res = await fetch(`${API}${path}`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json', 'Accept': 'application/json'
},
    body: JSON.stringify(payload)
  });
  const data = await res.json().catch(() => ({}));
  if (!res.ok) throw new Error(data.message || `HTTP ${res.status}`);
  return data;
}

async function ensureClientId() {
  const name = localStorage.getItem('clientName');
  if (!name) throw new Error('Nom client manquant.');
  const client = await postJson('/clients', { name });
  return client.id; // { id, name }
}

function buildItemsFromCart() {
  const cart = readCart();
  return cart.map(l => ({ dishId: l.dishId, qty: l.qty }));
}

async function submitOrder() {
  const clientId = await ensureClientId();
  const items = buildItemsFromCart();
  if (items.length === 0) { alert('Panier vide.'); return; }

  const order = await postJson('/orders', { clientId, items }); // { id }
  localStorage.removeItem('cart');
  alert(`Commande #${order.id} confirmée !`);
}

document.querySelector('#btn-checkout')?.addEventListener('click', () => {
  submitOrder().catch(() => alert('Erreur commande.'));
});
```

Back — routes / controllers / models

back/routes/clients.routes.js

```
import { Router } from 'express';
import { postClient } from '../controllers/clients.controller.js';
const r = Router();
r.post('/', postClient); // POST /api/clients
export default r;
```

back/controllers/clients.controller.js

```
import { createOrGetClientByName } from '../models/clients.model.js';

export async function postClient(req, res) {
  try {
    const { name } = req.body || {};
    if (!name) return res.status(400).json({ message: 'name required' });
    const client = await createOrGetClientByName(name);
    res.status(201).json(client); // { id, name }
  } catch {
    res.status(500).json({ message: 'client error' });
  }
}
```

back/models/clients.model.js

```
import { sql } from './db.js';

export async function createOrGetClientByName(name) {
  const inserted = await sql`
    INSERT INTO clients (name) VALUES (${name})
    ON CONFLICT DO NOTHING
    RETURNING id, name
  `;
  if (inserted.length) return inserted[0];

  const existing = await sql`
    SELECT id, name FROM clients WHERE name = ${name} LIMIT 1
  `;
  return existing[0] || null;
}
```

back/routes/orders.routes.js

```
import { Router } from 'express';
import { postOrder } from '../controllers/orders.controller.js';
const r = Router();
r.post('/', postOrder); // POST /api/orders
export default r;
```

back/controllers/orders.controller.js

```
import { createOrder } from '../models/orders.model.js';

export async function postOrder(req, res) {
  try {
    const { clientId, items } = req.body || {};
    if (!clientId || !Array.isArray(items) || items.length === 0) {
      return res.status(400).json({ message: 'clientId and items required' });
    }
    const order = await createOrder(clientId, items);
    res.status(201).json(order); // { id }
  } catch {
    res.status(500).json({ message: 'order error' });
  }
}
```

back/models/orders.model.js

```
import { sql } from './db.js';

export async function createOrder(clientId, items /* [{dishId, qty}] */) {
  try {
    await sql`BEGIN`;
    const created = await sql`
      INSERT INTO orders (client_id) VALUES (${clientId})
      RETURNING id
    `;
    const orderId = created[0].id;

    for (const it of items) {
      await sql`
        INSERT INTO order_items (order_id, dish_id, qty)
        VALUES (${orderId}, ${it.dishId}, ${it.qty})
      `;
    }
    await sql`COMMIT`;
    return { id: orderId };
  } catch (e) {
    await sql`ROLLBACK`;
    throw e;
  }
}
```

back/server.js (si on ajoute toutes les routes)

```
import 'dotenv/config';
import express from 'express';
import cors from 'cors';

import menuRoutes from './routes/menu.routes.js';
import clientsRoutes from './routes/clients.routes.js';
import ordersRoutes from './routes/orders.routes.js';

const app = express();
app.use(cors({ origin: ['http://127.0.0.1:5500', 'http://localhost:5500'] }));
app.use(express.json());

app.use('/api/menu', menuRoutes);
app.use('/api/clients', clientsRoutes);
app.use('/api/orders', ordersRoutes);

app.listen(process.env.PORT || 3000, () =>
  console.log('API http://localhost:3000'));
```

Dictionnaire (mini-glossaire appliqué)

- **payload** : le **corps** d'une requête HTTP (contenu envoyé), ex. `{ name: "Iris" }`.
- **endpoint** : URL précise d'une API (ex. `/api/orders`).
- `fetch(url, options)` : API du navigateur pour lancer des requêtes HTTP **asynchrones**.
- `response.ok` : booléen `true` si le code HTTP est **2xx** (succès).
- `!` : opérateur de **négation** logique (inverse un booléen).
- `response.status` **/** `response.statusText` : code/texte HTTP (ex. `404`, `"Not Found"`).
- `response.json()` : lit et décode la réponse en **objet JS**.
- `JSON.stringify(obj)` : encode un objet JS en **texte JSON** (pour `body`).
- `try/catch` : capture les erreurs (réseau ou `throw`).
- `throw new Error(msg)` : propage une erreur avec **message**.
- **CORS** : politique qui autorise (ou non) une **origine** front à appeler ton API.
- **controller** : code back qui gère la requête (valide, appelle la DB, renvoie).
- **model** : code back d'accès aux données (SQL vers la DB).
- **localStorage** : stockage clé/valeur **persistant** dans le navigateur.
- **status HTTP** : `200` OK, `201` Created, `400` Bad Request, `404` Not Found, `500` Internal Server Error.