"Pressure changes everything. Pressure. Some people, you squeeze them, they focus. Others fold. Can you summon your talent at will? Can you deliver on a deadline? Can you sleep at night?" - AL PACINO

## MVC ?

Pas exactement ! Ce n'est pas un vrai MVC complet.
Cependant on garde des parties du modèle :

- Controllers : logique métier
- Services ( models ) : gestion des données

Dans ce cas, les Views sont les réponses JSON

## L'idée générale du projet

C'est une API basique Node.js sans Express; le but c'est d'apprendre comment :

- Créer un serveur HTTP natif
- Comprendre le routage manuel
- Utiliser les modules vu en cours + bonus
- Apprendre la structure d'une appli backend

Cette première partie constitue la fondation du grand projet SmartInventory, qu'on va par la suite transformer en une vrai API en utilisant Express, MongoDB ..
Pour faire simple on reste sur la lecture seule ( donc pas d'écriture ni db )

## Ce que vous devez retenir :

- Comment node gère les requêtes
- La séparation du code :
  server/router/controller/services
- Filtrage et Formattage d'une réponse format JSON
- Comment générer les logs avec EventEmitter
- Comment lire les JSON files sans bloquer le thread

POUR RÉSUMER :
Il faut comprendre la **logique backend** avant d'utiliser le framework Express

## Ce quoi doit faire votre solution

1. **Démarre un serveur HTTP sur le port 3000 par défaut**
2. **Répond à des routes ( products, orders, health )**
3. **Lit des fichier JSON ( data/products.json et data/orders.json )**
4. **Renvoie les données dans ces fichiers**
5. **Apllique des filtres sur les requêtes (? category=tools&minPrice=10&maxPrice=100 etc.).**
6. **Logge les requêtes avec un EventEmitter (request:received, response:sent).**
7. **Gère les erreur HTTP ( 400 si valide, 404 pour introuvables, 500 erreur interne )**
8. **Bonus : une route qui signe, compresse et exporte les produits**



# Feuille de route

Un "Pas à Pas" pour éviter les blocages et éclaircir le process à suivre.

## L'organisation

- server.js → crée le serveur HTTP
- router.js → choisit le bon contrôleur selon l'URL
- controllers/ → logique par ressource (produits, commandes)
- services/ → lecture et filtrage des données
- utils/ → outils partagés (sendJson, logger, parseQuery)
- data/ → fichiers JSON statiques

## Le flow complet :

Ce qui se passe

**1- Le serveur démarre (server.js) :**

Node crée un serveur HTTP et attend des requêtes.

2- **Un client envoie une requête (ex: GET /api/ products).**

Le serveur la reçoit et la transmet au routeur.

**3- Le routeur (router.js) regarde l'URL et choisit le bon contrôleur.**

Analogie : Le **serveur (le serveur du resto)** lit la commande et l'envoie au bon **cuisinier**.

4- **Le contrôleur (controllers/…) lit les paramètres (query, id, etc.) et demande les données au service.**

Analogie :
Le **cuisinier (controller)** dit au **commis (service)** d'aller chercher les ingrédients.

**5 - Le service (services/…) lit les fichiers JSON dans data/, filtre et renvoie les résultats.**

Analogie :
Le **commis (service)** ouvre le frigo (les fichiers JSON) et prépare les bons ingrédients.

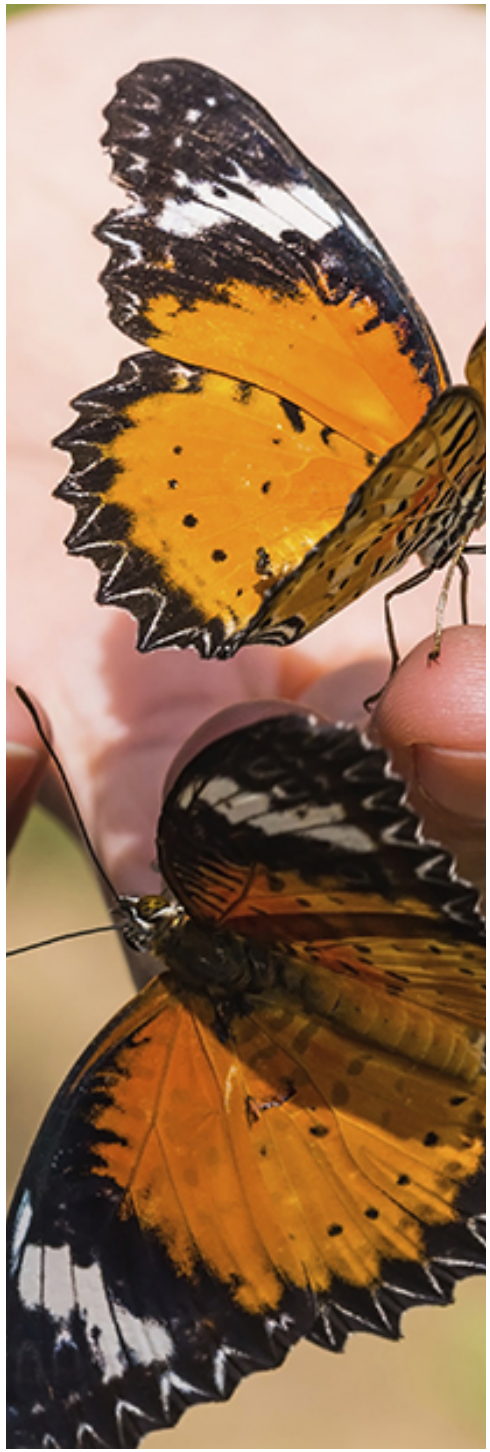
6- **Le contrôleur** formate la réponse et la renvoie au client sous forme JSON.

Analogie :
Le **cuisinier (controller)** dresse le plat (JSON) et le serveur le livre au client.

7- **Le logger (`utils/logger.js`)** enregistre la requête et la réponse dans la console.

Analogie :
Le **responsable du restaurant** note : "commande reçue / plat envoyé".

# Par où commencer (ordre logique)



0) Préparer l'environnement

Créez un dossier de projet et un dépôt Git.
Ajoute un package.json minimal comme décrit sur l'énoncé
Créez l'arborescence des dossiers

1) Créer / peupler la **data** (la "mini base de données")

des JSONs valides

2) Poser les **utils** indispensables

**src/utils/sendJson.js :** centraliser la réponse JSON (status + headers).
**src/utils/parseQuery.js :** helpers toNumber, toBool, parseDate.
**src/utils/logger.js :** EventEmitter avec deux événements : request:received, response:sent

Checkpoint : Vous pouvez require() ces utils sans erreur depuis un fichier test.

3) Écrire le **serveur** minimal

src/server.js : créer un serveur HTTP, définir le header JSON par défaut, émettre les logs via le logger.
src/router.js (temporaire) : renvoyer un 404 par défaut.
Tester : npm run dev —> curl -i http://localhost:3000/unknown —> ( 404 )

4) Ajouter une **route "health"** (la plus simple)

Dans router.js :

- parse l'URL (url.parse(req.url, true)),
- si GET /health → renvoyer { status:"ok", uptime, timestamp }.

Tester : curl http://localhost:3000/health. —> ( JSON avec status, uptime, timestamp )

Ce test est un Hello world, Félicitations !

5) Mettre en place le routage paramétré

- Dans router.js, ajoute un mini matcher pour /api/products/:id, etc.
  (Split par “/”, comparer segments, capturer :id.)
- Garde une table de routes lisible (évite un énorme if spaghetti).

Checkpoint : match("/api/products/101", "/api/products/:id") renvoie { id:"101" }.

6) Créer les services (accès data)

- src/services/productsService.js
  - ➡ Lire data/products.json (lecture au démarrage ou lazy + cache mémoire).
  - ➡ Exposer fonctions : search({...}), byId(id), bySku(sku).
  - ➡ Gérer filtres : q, category, minPrice, maxPrice, inStock, page, limit, (optionnel sort, order).

- src/services/ordersService.js
  - ➡ Lire data/orders.json.
  - ➡ Exposer search({status, from, to, page, limit}), byId(id), byOrderNumber(orderNumber).

Pièges: (
Valide les types (nombres, booléens, dates ISO).
Si lecture/parse JSON échoue → lève une erreur (le contrôleur renverra **500**
).

7) Créer les **controllers** (logique métier + réponses HTTP)

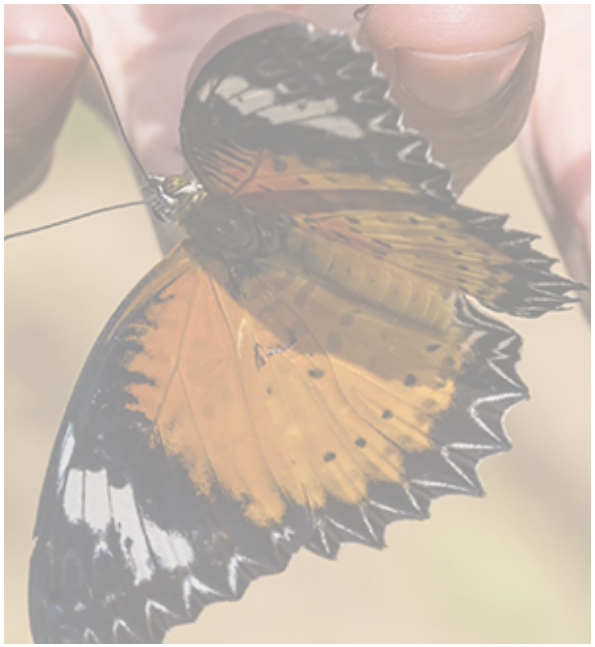
- src/controllers/productsController.js
  - ➡ listProducts(req,res,query) → utilise productsService.search, applique validation simple (minPrice<=maxPrice), renvoie { data, total, page, pages }.
  - ➡ getProductById(req,res,id) → 404 si absent.
  - ➡ getProductBySku(req,res,sku) → 404 si absent.
- src/controllers/ordersController.js
  - ➡ listOrders(req,res,query) → dates invalides → **400**.
  - ➡ getOrderById, getOrderByNumber.

Tester progressivement ( curl sur les produit sur l’énoncé ) —> JSON correct —> 404 & 400 justes

8) Brancher le router vers les controllers

- Dans router.js, route GET /api/products, /api/products/:id, /api/products/sku/:sku.
- Idem pour orders.
- Garde un catch-all 404 à la fin.

Tester ( Toutes les routes de la spec répondent, Les erreurs 400/404/500 sont renvoyées proprement (JSON). )



9) Ajouter le logging événementiel propre

- Vérifie que logger.js émet bien request:received et response:sent.
- Log concis : [ISO] → GET /api/products puis [ISO] ← 200 /api/products?....

Checkpoint: Les logs s'affichent à chaque requête, sans bruit excessif.

10) (Optionnel) BONUS : export compressé + signé

Route GET /api/export.gz :

- lit les produits,
- zlib.gzip du buffer JSON,
- si HMAC_SECRET défini → X-Signature: <hex hmac sha256> (module crypto),
- headers : Content-Type: application/gzip, Content-Disposition: attachment; filename="products.json.gz".

Tester : .env : HMAC_SECRET=secret —> npm run dev. —> curl -i http://localhost:3000/api/export.gz -o products.json.gz. —> X-Signature + fichier gzip valide

11) Rédiger un README minimal

Comment démarrer (npm run dev), variables d'env (PORT, HMAC_SECRET), endpoints et exemples curl, limites (lecture seule), bonus.

12) Démo

---

Là c'est terminé