

Cahier des charges – API NestJS

Projet : “Chez Nest-Or”, la pizzeria

Contexte

“Chez Nest-Or, le Nest plus ultra de la pizza”.

Cette petite PME fait appel à vous pour développer une API interne pour gérer sa carte :

- les pizzas
- les boissons
- les desserts
- les commandes
- des menus avec réduction

L'objectif est de concevoir une **API REST modulaire, propre et évolutive** avec NestJS, intégrant une logique métier cohérente.

1. Contraintes techniques

Le projet doit :

- respecter l'architecture NestJS (modules, controllers, services)
 - utiliser des DTO avec validation (`class-validator`)
 - activer la validation globale
 - gérer correctement les erreurs HTTP
 - séparer clairement logique HTTP et logique métier
 - fonctionner en mémoire
-

2. Architecture attendue

Organisation par domaine :

```
src/
  ├── pizzas/
  ├── drinks/
  ├── desserts/
  ├── orders/
  ├── menu/
  └── app.module.ts
```

Chaque domaine doit contenir :

- module
 - controller
 - service
 - dossier dto/
-

3. Module Pizzas

Modèle Pizza

- `id: number`
 - `name: string`
 - `price: number`
 - `ingredients: string[]`
 - `available: boolean`
-

Endpoints obligatoires

Méthode	Route
GET	/pizzas
GET	/pizzas/:id
POST	/pizzas
PUT	/pizzas/:id (remplacement complet)

`DELETE /pizzas/:id`

Le `PUT` doit remplacer entièrement la ressource.

Validation

- `name` : string, min 3 caractères
 - `price` : number strictement positif
 - `ingredients` : tableau non vide
 - `available` : boolean
-

4. Module Drinks

Modèle Drink

- `id`: number
 - `name`: string
 - `price`: number
 - `size`: string
 - `withAlcohol`: boolean
 - `available`: boolean
-

Endpoints obligatoires

Méthode	Route
---------	-------

GET	/drinks
-----	---------

GET	/drinks/:id
-----	-------------

POST	/drinks
------	---------

PUT	/drinks/:id
-----	-------------

DELETE	/drinks/:id
--------	-------------

5. Module Desserts

Modèle Dessert

- `id: number`
- `name: string`
- `price: number`
- `available: boolean`

CRUD simple identique aux boissons.

6. Module Orders

Modèle Order

- `id: number`
 - `pizzas: number[]`
 - `drinks: number[]`
 - `desserts: number[]`
 - `totalPrice: number` (calculé automatiquement)
 - `processed: boolean`
 - `createdAt: Date`
-

Endpoints obligatoires

Méthode	Route
GET	/orders
GET	/orders/:id
POST	/orders
PUT	/orders/:id
PATCH	/orders/:id/processed
DELETE	/orders/:id

Contraintes métier

Lors de la création ou modification d'une commande :

- toutes les ressources doivent exister
- aucune ressource indisponible
- `totalPrice` doit être recalculé automatiquement
- une erreur 400 ou 404 doit être levée si nécessaire

Le PATCH `/orders/:id/processed` doit modifier uniquement le champ `processed`.

7. Module Menu (logique métier)

Le module `menu` ne stocke pas de données.

Il contient uniquement une logique de calcul tarifaire.

Règle métier

Si une commande contient au moins :

- 1 pizza
- 1 boisson **sans alcool**
- 1 dessert

Alors une réduction de **10%** est appliquée à la somme des 3 éléments de la commande.

Les boissons avec alcool (`withAlcohol: true`) ne peuvent pas entrer dans un menu promotionnel.

La logique de calcul doit être centralisée dans un service dédié.

8. Gestion d'erreurs

- 404 si ressource inexiste
 - 400 si validation invalide
 - utilisation correcte des exceptions NestJS
 - validation globale activée
-

9. Livrable

Repository Git contenant :

- le code source
 - un README expliquant :
 - l'architecture
 - les endpoints
 - la logique du menu
-

Pour aller plus loin (optionnel)

1. Stockage NoSQL en fichiers JSON

Remplacer le stockage en mémoire par :

- fichiers JSON (`pizzas.json`, `orders.json`, etc.)
- lecture/écriture via `fs`
- simulation d'un stockage NoSQL simple

Objectif : comprendre la persistance sans base relationnelle.

2. Recherche par ingrédients

Ajouter :

- possibilité de filtrer les pizzas par ingrédient
Exemple :
`GET /pizzas?ingredient=tomato`

Ou recherche multiple :

`GET /pizzas?ingredients=tomato,mozzarella`

Cela implique :

- enrichir la logique de filtrage
- gérer proprement les query params
- normaliser les chaînes (lowercase, trim...)

3. Interface Front (libre)

Créer une interface web minimaliste permettant :

- afficher les pizzas
- créer une commande
- voir si la réduction menu est appliquée
- marquer une commande comme traitée

Technologie libre (HTML, framework JS, etc.).

Objectif : comprendre la communication front ↔ API REST.

4. Améliorations techniques

- Interceptor pour uniformiser les réponses
- Exception filter personnalisé
- Tests unitaires
- Logger personnalisé
- Swagger